TR82-002

PARALLEL SOLUTION OF ELLIPTIC PARTIAL DIFFERENTIAL EQUATIONS ON A TREE MACHINE

by

Roy Peredo Pargas

A dissertation submitted to the faculty of the University of North Carolina at Chapel Hill in partial fulfillment of the requirements for the degree of Doctor of Philosophy in Computer Science.

Chapel Hill

1982

Approved by:

Adviser: Gyula A. Magó

Reader: Donald F. Stanat

Reader: Bharadwaj Jayaraman

ROY PEREDO PARGAS. Parallel Solution of Elliptic Partial Differential Equations on a Tree Machine. (Under the direction of DR. GYULA A. MAGO.)

ABSTRACT

The numerical solution of elliptic partial differential equations (pde's) can often be reduced to the solution of other relatively simple problems, such as solving tridiagonal systems of equations and low-order recurrence relations. This thesis describes elegant and efficient tree machine algorithms for solving a large class of these simpler problems, and then uses these algorithms to obtain numerical solutions of elliptic partial pde's using methods of finite differences.

The tree machine model on which this work is based contains data only in the leaf cells of a complete binary tree of processors; one leaf cell typically holds all information pertinent to one point of the rectangular mesh of points used by the method of finite differences. An algorithm is described for communication among leaf cells using shortest paths; other algorithms are exhibited that find the first n terms of the solution to several classes of recurrence expressions in $O(\log n)$ time.

The communication and recurrence expression tree algorithms are used to describe algorithms to solve $(n \times n)$ tridiagonal linear systems of equations. A number of direct methods are shown to require $O(\log n)$ time, whereas other direct methods require $O((\log n)^2)$ time. Iterative methods are shown to require $O(\log n)$ time per iteration. The asymptotic complexity of both direct and iterative methods implemented on sequential, vector, array, and tree processors are compared.

The tridiagonal linear system solvers and the communication algorithms are used to describe algorithms to solve $(n^2 \times n^2)$ block-tridiagonal linear systems iteratively. Both point iterative and block iterative methods are shown to require $O(n)$ time per iteration. Alternating direction implicit methods require $O(n \log n)$ time per iteration. The asymptotic complexity of methods implemented on sequential, vector, array, and tree processors are again compared.

DEDICATION

To Maris, Rica, and Rebecca,
who made the bad times good
and the good times better.

ACKNOWLEDGEMENTS

A dissertation is never the product of one person alone. There are several people to whom I wish to express my gratitude.

I thank Dr. Gyula A. Magó, my adviser, who always managed to point me in the right direction whenever I got stuck. Besides suggesting the original research topic, he provided many hours of guidance and carefully read the numerous drafts of this dissertation.

I also thank the other members of my committee, Dr. Donald F. Stanat, Dr. Bharadwaj Jayaraman, Dr. Steven M. Pizer, and Dr. Douglas G. Kelly. Don Stanat, in particular, has been a close friend as well as an adviser.

I am very grateful to all my friends, especially to Anne Presnell and Ed Jones. Anne developed the communication algorithm, Atomic Rotate Left, which proved extremely useful. Ed was always willing to listen, and helped improve many of the algorithms.

Finally, I wish to thank my family, especially my mother and parents-in-law for many years of encouragement and support.

This work was supported in part by the National Science Foundation under Grants MCS78-02778 and MCS80-04206.

TABLE OF CONTENTS *page*

LIST OF FIGURES

Figure

CHAPTER 1. INTRODUCTION

The formulation of mathematical models in engineering and the physical sciences often involves partial differential equations (pde's). Rice *et al.* [Rice79] give as examples models to perform

> numerical weather prediction ..., the simulation of nuclear reactors and fusion reactors, the analysis of the structural properties of aircraft and bridges, the simulation of blood flow in the human body, the computation of air flow about an aircraft or aerospace vehicle, the propagation of noise through the atmosphere, and the simulation of petroleum reservoirs.

The numerical solution of partial differential equations often reduces to a problem of solving very large linear systems of equations in which the coefficient matrix is sparse and of a special structure (e.g. tridiagonal, block-tridiagonal). The sheer number of operations required (some applications are estimated to require at least 10^{18} operations, or about 10^7 hours on a CDC STAR-100) has motivated the search for faster methods of computing.

During the last fifteen years, much attention has been on the design and implementation of parallel algorithms to solve pde's on array or vector processors such as the ILLIAC-IV, the CDC STAR-100, and the CRAY-1. One approach has been to implement already existing algorithms originally intended for sequential machines on a parallel processor. This almost always requires a modification of the algorithm in order to introduce parallelism best suited to the machine's particular capabilities. One often hears of "vectorizing" a sequential FORTRAN program in order for it to run efficiently on a vector processor. Another approach has been to design new parallel algorithms specially suited for a particular

machine. At times, the algorithms produced, although efficient on a parallel processor, are inefficient on a sequential processor[1]. The lesson one immediately learns when studying parallel algorithms is that the requirements for efficiency are different from those for sequential computers. One rather difficult problem for the designer of parallel algorithms is communication among the processing elements of a parallel machine.

The parallel processor we study in this dissertation is a tree machine, i.e., a network of processors (nodes, cells) interconnected to form a binary tree [Mag679a, Toll81, Brow79]. In the design proposed by Mag6, hereafter referred to as *MM*, the nodes (of which there are two types) are small. For example, they might contain a bit-serial ALU, bit-serial communication between nodes, a few dozen registers, and a small memory to hold dynamically loaded microprograms. *MM* is a *small-grain* system with possibly several hundred thousand nodes. (Here, *granularity* is defined as the size of the largest repeated element [SuDD81].) This is in contrast to *large-grain* tree machines whose nodes are von Neumann processors, possibly numbering in the hundreds or thousands. (Systolic arrays [Kung79] are probably the best known examples of small grain systems.)

The problem we investigate in this dissertation is *the implementation on a tree machine of numerical methods to solve second-order elliptic partial differential equations using finite differences*. In particular, we seek to answer three questions.

---

[1]For example, recursive doubling [Ston73a, Ston75] is a parallel algorithm that solves an $(n \times n)$ tridiagonal linear system of equations (a basic subproblem of pde solvers) in $O(\log n)$ time on an array processor. On a sequential processor, recursive doubling takes $O(n \log n)$ time, compared to $O(n)$ time required by Gaussian elimination.

(1) Can solutions to elliptic partial differential equations be implemented efficiently on a tree machine?

(2) How does the tree machine implementation compare with implementations on other high performance machines, such as vector computers?

(3) What conclusions regarding tree machine programming do these implementations provide?

No attempt is made to design new elliptic pde solvers. All methods mentioned in this dissertation have been designed and analyzed previously.

To answer the above questions, we present a description of a simple abstract special-purpose tree machine and several classes of algorithms designed to run on it. The dissertation is organized as follows.

Chapter 2. Preliminaries

- Overview of partial differential equations and the method of finite differences.
- A description of the tree machine and its component cells.
- The algorithmic language used to specify some of the algorithms.
- Analysis of algorithms.

Chapter 3. Tree machine algorithms: algorithms that form the basic building blocks of tridiagonal and block-tridiagonal linear system solvers

- $O(\log n)$ algorithms that solve low-order linear recurrences and recurrence expressions, such as continued and partial fractions.

- Tree communication algorithms, that is, techniques that allow efficient communication among the leaf cells of the tree. Efficient communication among the processors is essential for an efficient implementation of parallel algorithms. We present algorithms that cyclically shift a vector of elements stored in the leaf cells a distance k to the left or right in $O(k)$ time if $k > \log n$, where n is the number of leaf cells in the tree, and $O(\log n)$ otherwise.

Chapter 4. Tridiagonal linear system solvers: tree machine implementations of direct and iterative parallel methods to solve tridiagonal linear systems

- Direct methods include Gaussian elimination, the Thomas algorithm, LU decomposition, a method using second-order linear recurrences (similar to Stone's recursive doubling [KoSt73], [Ston75]), cyclic reduction and the Buneman variant of cyclic reduction. On a sequential computer, all of the methods require $O(n)$ time. On a tree machine, cyclic reduction and the Buneman variant require $O(\log n)^2$ time; Gaussian elimination, the Thomas algorithm, LU decomposition, and the recursive doubling variant all require $O(\log n)$ time.

- Iterative methods include the Jacobi method, Jacobi over-relaxation, the Gauss-Seidel method, successive over-relaxation, red-black successive over-relaxation, and the iterative analog of LU decomposition (developed by Traub [Trau73]). All require $O(n)$ time on a sequential computer and $O(\log n)$ time on a tree machine per iteration.

- Comparison of results obtained with those for parallel and vector processors

Chapter 5. Iterative block-tridiagonal linear system solvers

- Tree machine implementations of point iterative, block iterative, and alternating direction implicit (ADI) methods to solve an $(n \times n)$ block-tridiagonal linear system. The point iterative methods studied are the Jacobi method, Jacobi over-relaxation, Gauss-Seidel, successive over-relaxation (SOR), and a variant of SOR, red-black successive over-relaxation. On a sequential computer, these methods require $O(n^2)$ time per iteration. Tree machine algorithms require $O(n)$ time per iteration. The block iterative methods studied are block Jacobi, block Jacobi over-relaxation, block Gauss-Seidel, and block SOR; all require $O(n)$ time per iteration. ADI methods studied require $O(n \log n)$ time per iteration.

- Detailed analysis of the time required by the Jacobi method

Chapter 6. Conclusions

CHAPTER 2. PRELIMINARIES

A. The Second-Order Partial Differential Equation

The general second-order partial differential equation (pde) in two dimensions

$$a\frac{\partial^2 z}{\partial x^2} + b\frac{\partial^2 z}{\partial x\,\partial y} + c\frac{\partial^2 z}{\partial y^2} + F\left(\frac{\partial z}{\partial x}, \frac{\partial z}{\partial y}, x, y\right) = 0 \qquad (2.1)$$

may be classified on the basis of the expression $b^2 - 4ac$ as follows:

elliptic	if $b^2 - 4ac < 0$
parabolic	if $b^2 - 4ac = 0$
hyperbolic	if $b^2 - 4ac > 0$.

Members of each class can be transformed, possibly with a change of variables, into a canonical form:

elliptic	$\dfrac{\partial^2 z}{\partial \xi^2} + \dfrac{\partial^2 z}{\partial \eta^2} = G$
parabolic	$\dfrac{\partial^2 z}{\partial \xi^2} = G$ (2.2)
hyperbolic	$\dfrac{\partial^2 z}{\partial \xi^2} - \dfrac{\partial^2 z}{\partial \eta^2} = G$

where $G = G(\xi, \eta, z, \partial z / \partial \xi, \partial z / \partial \eta)$. This dissertation deals exclusively with elliptic equations, of which the most commonly studied ones are Laplace's equation (2.3) and Poisson's equation (2.4).

$$\frac{\partial^2 z}{\partial x^2} + \frac{\partial^2 z}{\partial y^2} = 0 \tag{2.3}$$

$$\frac{\partial^2 z}{\partial x^2} + \frac{\partial^2 z}{\partial y^2} = \text{constant} . \tag{2.4}$$

In this dissertation, we will investigate how one may solve problems involving these equations on a tree machine.

B. The Method of Finite Differences

Problems involving second-order elliptic pde's are equilibrium problems. Given a region R, bounded by a curve C on which the function z is defined (the boundary conditions), and given that z satisfies Laplace's or Poisson's equation in R, the objective is to determine the value of z at any point in R. The method of finite differences is a widely used numerical method for solving this problem. The basic strategy is to approximate the differential equation by a difference equation and to solve the difference equation.

Consider Laplace's equation (2.3). Let R be a rectangular region and C its perimeter. Laying a rectangular mesh with n rows, m columns, and equal spacing h on the region (Figure 2.1), we want an approximation of the function z at the interior mesh points. Once this is determined, approximations at other points in the region may be obtained through interpolation.

One approximation is to replace the second derivatives in (2.3) with the centered second differences, so that for $z = z_i$,

$$\frac{\partial^2 z}{\partial x^2} \approx (z_{i-1} - 2z_i + z_{i+1})/h^2 \tag{2.5}$$

and

$$\frac{\partial^2 z}{\partial y^2} \approx (z_{i-m} - 2z_i + z_{i+m})/h^2 . \tag{2.6}$$

Laplace's equation is therefore approximated by

$$\frac{\partial^2 z}{\partial x^2} + \frac{\partial^2 z}{\partial y^2} = 0 \approx (z_{i-m} + z_{i-1} - 4z_i + z_{i+1} + z_{i+m})/h^2 \tag{2.7}$$

which gives

$$0 \approx z_{i-m} + z_{i-1} - 4z_i + z_{i+1} + z_{i+m} \tag{2.8}$$

indicating that one way to represent each point z_i is by a linear equation. The object is to solve the ith equation for z_i. This method, sometimes called the *direct* method, uses equation (2.8) and transforms the problem of approximating the z values at the $(n-2)(m-2)$ interior points to one of solving $(n-2)(m-2)$ linear equations in as many unknowns. The coefficient matrix of this linear system is block-tridiagonal in structure. The example of Figure 2.1 would have a coefficient matrix as shown in Figure 2.2.

Equation (2.8) may also be expressed as

$$z_i \approx (z_{i-1} + z_{i+1} + z_{i-m} + z_{i+m})/4 \tag{2.9}$$

suggesting that, if we know (or can approximate) the values of an interior point's four closest neighbors, we may iteratively improve the value at the point by replacing it with the average of its four closest neighbors. This is called the

iterative method. After assigning an initial value to each of the interior mesh points, the method iteratively improves the approximation by replacing each point with a weighted average of its four closest neighbors, as specified in equation (2.9). One pass through the mesh points constitutes one iteration. We may iterate as many times as desired, until some criterion for convergence has been satisfied. This method has been shown to have $O(h^2)$ convergence where h is the distance between two neighboring mesh points [Ames77].

The iterative approach involves a simple computation, in the simplest case nothing more than the averaging of four values. Higher-order approximations of Laplace's equation require using more points in the approximation but the basic operation remains the taking of weighted averages. Moreover, there is a great amount of parallel activity possible: theoretically, we may compute the ith approximation of all interior mesh points simultaneously. On a parallel processor, it may be possible to perform one iteration (modify all points) in as little time as it takes to modify one point. While this operation would appear to be trivial on the ILLIAC-IV whose processing elements are interconnected to form a rectangular mesh, the solution on a tree machine is far from obvious.

We will investigate the implementation of iterative methods of solving block-tridiagonal linear systems on a tree machine. (A more detailed discussion of the method of finite differences applied to elliptic pde's is given by Forsythe and Wasow [FoWa60] and Ames [Ames77].)

C. The Tree Machine (TM)

1. Previous Work

Magó [Mag679a], [Mag680] has proposed a cellular computer, here referred to as *MM*, organized as a binary tree of processors, that allows simultaneous evaluation of expressions stored in the leaf cells of the tree. It directly executes functional programming languages, a class of languages developed by Backus [Back78], in which the expression of parallelism is natural. Tolle [Toll81] has proposed a similar tree-structured cellular computer with more powerful, but more complex, cells. In both designs, processors contained in the tree cells are capable of independent operation, thus providing the potential for parallel computation. Williams [Will81] studied parallel associative searching algorithms and presented several techniques to predict and analyze the amount of time and storage required by the algorithms if run on *MM*. Koster [Kost77] and Magó, Stanat, and Koster [MaSK81] developed a method for obtaining upper and lower bounds of the execution time of programs run on *MM*. Their analysis carefully accounts for communication and storage management costs. Parallel algorithms for tree machines have also been developed by Browning [Brow79] for a variety of applications, including sorting, matrix multiplication, and the color cost problem, and by Bentley and Kung [BeKu79] for searching problems. Leiserson [Leis79] studied systolic trees and how to maintain a priority queue on one.

2. Overview of TM

In this section, we describe *TM*, a special-purpose tree network of processors similar to, but of a much simpler structure and less powerful than, the

general-purpose machines proposed by Magó and Tolle. *TM* is a binary tree network of processing elements in which the branches are two-way communication links (Figure 2.3). Leaf and nonleaf processing elements are called L cells and T cells respectively. Attached to the root cell, functioning as the root cell's father, is a cell called Control (C cell).

When describing algorithms, cells are sometimes referred to by their level in the tree. The L cells are on level 0, the lowest level T cells are on level 1, the root T cell is on level $\log N$, and the C cell is on level $\log N + 1$, where N is the number of L cells in the tree. Two-way communication among the cells is conducted through the tree branches; a T cell may communicate with its father and two sons and an L cell may communicate with its father; a C cell communicates with the root T cell and with external storage, as explained below. An L cell may communicate with another L cell by sending information up the tree through the sending L cell's ancestor T cells and then back down again to the receiving L cell.

In principle, all cells operate asynchronously. However, the algorithms presented can be more easily understood if we view the operation as proceeding in synchronous upward and downward sweeps. We note, however, that this synchrony is not a necessary feature of *TM*.

An example of a task requiring a downward sweep is that of broadcasting information to all L cells. The C cell sends information to its son the root cell, which sends the information to its two sons, which send the information to their sons, and so on, until the information is simultaneously received by the L cells. An example of a task requiring an upward sweep is that of adding the values

stored in the L cells with the C cell receiving the sum.

3. The Microprogramming Language

As Hoare observed [Hoar78], a programming language for a machine with multiple, independent, asynchronous processing units must contain special statements not ordinarily found in languages for sequential computers. This is, in part, because the processing units must have a way of communicating and synchronizing with each other. The language he describes for multiple processors, CSP, includes statements such as parallel commands specifying possible concurrent execution of its components, and input and output commands used for communication between processors. Browning and Seitz [BrSe81] concur with Hoare and have written a compiler for TMPL, a language similar to Hoare's CSP, for the purpose of implementing algorithms on a tree machine. Programs presented in this dissertation will be written in an algorithmic language whose special features are described briefly below. Communication between tree cells will be handled by SEND and RECEIVE commands. The CASE command and the concurrent execution of statements are also explained. Figure 2.4 shows a few sample statements.

SEND and RECEIVE require cooperation between two cells. In order for a cell to execute a SEND statement, the receiving cell must execute a RECEIVE statement (Figures 2.4a and 2.4b). It may happen that a cell wishing to send (receive) data must wait until its partner is ready. A CASE statement allows a cell to wait for more than one other cell. Figure 2.4c shows a cell attempting to receive data from its left son (case c.1), receive data from its right son if n is

currently nonzero (case c.2), or send data to its father if n is nonnegative (case c.3). Only one of the cases will be executed and the choice may depend upon the value of n. If for example, n is currently 0, only cases c.1 and c.3 may be executed. If both the cell's left son and father are ready to communicate, the cell randomly chooses between them, executes the statement, and either increments n (left son was chosen) or sets n to 0 (father was chosen). If neither father nor left son is ready to communicate, the cell waits until one is ready. We allow only one SEND or RECEIVE statement for each condition of a case statement. Figure 2.4d shows concurrent execution. Data must be sent to both sons but the order of execution is not important; the first ready son is sent the data first. Had the statement been written

L.SEND(X); R.SEND(Y);

the right son may be unnecessarily delayed from receiving its data if the left son is not ready to execute an F.RECEIVE statement.

4. The Tree Cells

A cell (L, T, or C) consists of a small memory and a processor (Figure 2.5). The memory holds the control program, a single cell microprogram, and a small number of registers used to store data. At all times, a cell processor is under the exclusive control of either the control program or the cell microprogram. The control programs are shown in Figure 2.6.

The L cell control program (Figure 2.6a) instructs the processor to wait until a microprogram set, consisting of a T cell and an L cell microprogram,

arrives. When one does, the control program instructs the processor to save the L cell microprogram and then to execute the microprogram, i.e., control of the processor is transferred to the microprogram. The microprogram specifies (1) how much data to read, (2) in which variables to store the data, (3) how to process the data, and (4) what to send back to the father. The microprogram retains control of the processor until its execution is complete. At that time, control of the processor returns to the control program. Note that while the control program is executing, the L cell expects, and should only receive, a microprogram set. While the microprogram is executing, the L cell may only receive data.

The T cell control program (Figure 2.6b) instructs the processor to wait until a microprogram set arrives. When one does, the processor sends a copy to each of its sons while saving the T cell microprogram. It then begins executing its microprogram. Like the L cell, control of the T cell processor returns to the control program only after the microprogram is executed.

The C cell control program instructs the C cell processor to fetch the next set of L, T, and C cell microprograms from external storage. The processor then saves the C cell microprogram and sends the L and T cell microprograms (the microprogram set) to the root T cell. The C cell then executes its microprogram. After execution, control returns to the control program which proceeds to fetch the next set of microprograms from external storage.

A C cell, a T cell, and an L cell microprogram, taken collectively, may be viewed as a global operation to be performed by all of the cells of the machine operating in harmony, whereas a microprogram specifies the local operation

performed by an individual cell. The C cell's supplying all cells with their microprograms is analogous to the instruction-fetch cycle in conventional von Neumann machines. The L cells, upon receiving their microprograms, initiate the execution of the algorithm and cause an execution chain reaction to ripple through the tree. This continues until all cells have completely executed their microprograms. This is analogous to the execution cycle in von Neumann machines. The last statement typically executed by an L cell is an

```
F.SEND ("DONE");
```

statement. When a T cell has received a "DONE" signal from both its sons, the signal is propagated up the tree. The "DONE" signal reaching the C cell marks the end of execution of the collection of (C, T, and L) microprograms. The control program takes over the C cell processor (as has already occurred in the L and T cells) and the execution of the next set of microprograms is ready to begin. At this point, we say that the machine has gone through one *execution cycle*.

5. Example and Analysis of an Algorithm

This section gives a simple example of L, T, and C cell microprograms and presents an execution time analysis of the algorithm. The problem is the following. Let each L cell contain an integer. The object is to store in each L cell the number of L cells whose values exceed half the sum of the integers. A solution to this problem, called COUNT, proceeds as follows:

(1) L cells send their values (integers) to their fathers. T cells add values received from their sons and send the sum to their fathers. The C cell

receives the sum and sends it back down through tree to the L cells.

(2) Each L cell compares its value with the sum and sends up a "1" if its value exceeds half the sum. Otherwise it sends up a "0". The T cells add the values received from their sons and send the sum to their fathers. The C cell returns the value it receives through the tree to the L cells.

The L, T, and C cell microprograms are shown in Figures 2.7a-c.

Each algorithm presented in this dissertation will be followed by a time analysis. For simplicity, we assume that cells on each level of the tree (the L cells are on level 0, their father T cells are on level 1, and so on) operate synchronously. The design of the algorithms make this a reasonable assumption. We emphasize, however, that this is not an essential feature of either *TM* or *MM*. The analysis will be expressed as the number of parallel arithmetic operations (additions and multiplications) and the amount of communication time required. If all of the L cells must execute a multiplication, for example, we assume that all of the L cells take the same amount of time to do it. Their collective action is considered, therefore, as a single multiplication. Communication is measured in steps where one step is defined as the time required for one cell to send one unit of information (one number, one character) to an adjacent cell. A cell may simultaneously send and receive information from cells adjacent to it. A T cell may therefore send as many as three numbers to its father and sons and receive three numbers from its father and sons in one time unit or step. The analysis of the COUNT algorithm is shown in Figure 2.8.

6. Relationship between TM and MM

TM is a *model* of a tree machine. Its primary function is to serve as a vehicle for describing tree machine algorithms. We may implement the algorithms described in this dissertation by incorporating *TM* into *MM*, a general-purpose machine capable of executing programs written in an FFP language. *MM*, during a process called partitioning [Mag679a], identifies the innermost applications of an FFP expression and, for each, constructs a binary subtree among the T cells. During subsequent machine cycles, these component trees machines simultaneously reduce the innermost applications. After reduction, the new innermost applications are identified and the process is repeated. Each of these component tree machines may be considered an instance of *TM*.

The embedding of *TM* into *MM* would be fairly straightforward. There are only a few details that need to be explicitly stated. First, all of the algorithms described on *TM* require that the processors contained in the T cells be able to execute slightly more complex programs than are described by Mag6. Moreover, T cells in *TM* are supplied user-defined microprograms (in *MM*, only the L cells receive microprograms, T cell programs are built-in). In short, a T cell processor should have the processing power of an L cell processor. Since all T nodes execute the same microprogram, such capability would be easy to add to the design described by Mag6 [Mag679a].

Secondly, the component tree machines formed in *MM* after partitioning are seldom complete binary trees. All of the algorithms described on *TM*, therefore, should execute correctly on incomplete, as well as complete, binary trees. Care was taken to ensure that this, in fact, be the case.

Finally, there is the question of interrupts. In *MM*, the root T cell periodically issues an interrupt to perform storage management. This interrupt causes all component tree machines still in the reduction process to temporarily halt their operations. Storage management may move the contents of the L cells of some component tree machines. If so, these component tree machines must be rebuilt (i.e., the branches between the L cells and the T cells must be redefined) and any information stored in the T cells of these machines before the interrupt must be reconstructed. It is necessary, therefore, that *TM* algorithms be able to perform this reconstruction easily. Again, care was taken to ensure this fact. Alternatively, it would be possible to have *MM* mark certain component tree machines as "uninterruptible" (this would require a minor modification of the machine described by Mag6 [Mag679a]). The machine would then delay storage management until the specially marked tree machines had completely reduced their applications.

However, the problem of interrupts may be moot if an elliptic pde is to be solved. Such problems usually deal with a great many data. It is likely that the entire machine, *MM*, would be dedicated to this purpose. If so, then the elliptic pde solver could be allowed to execute uninterrupted.

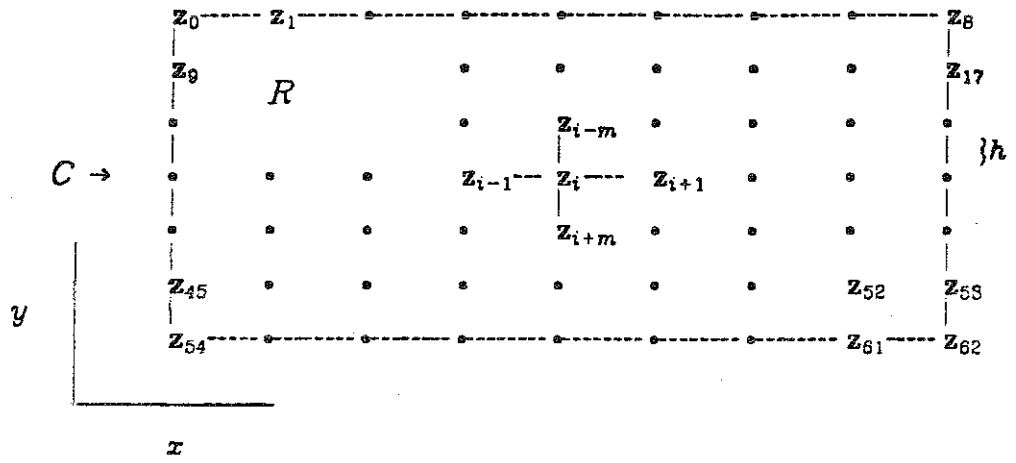


Figure 2.1 A rectangular mesh with n=7 rows and m=9 columns. The mesh points are labeled z_0 through z_{62} in row major order. Each interior point z_i (that is, each point not on the perimeter C) has as its four closest neighbors the points z_{i-m}, z_{i-1}, z_{i+1}, and z_{i+m}. Note that the corner points z_0, z_8, z_{54}, and z_{62} do not have any interior points as neighbors and will not participate directly in any computation. They are included to make the subscripting of points regular.

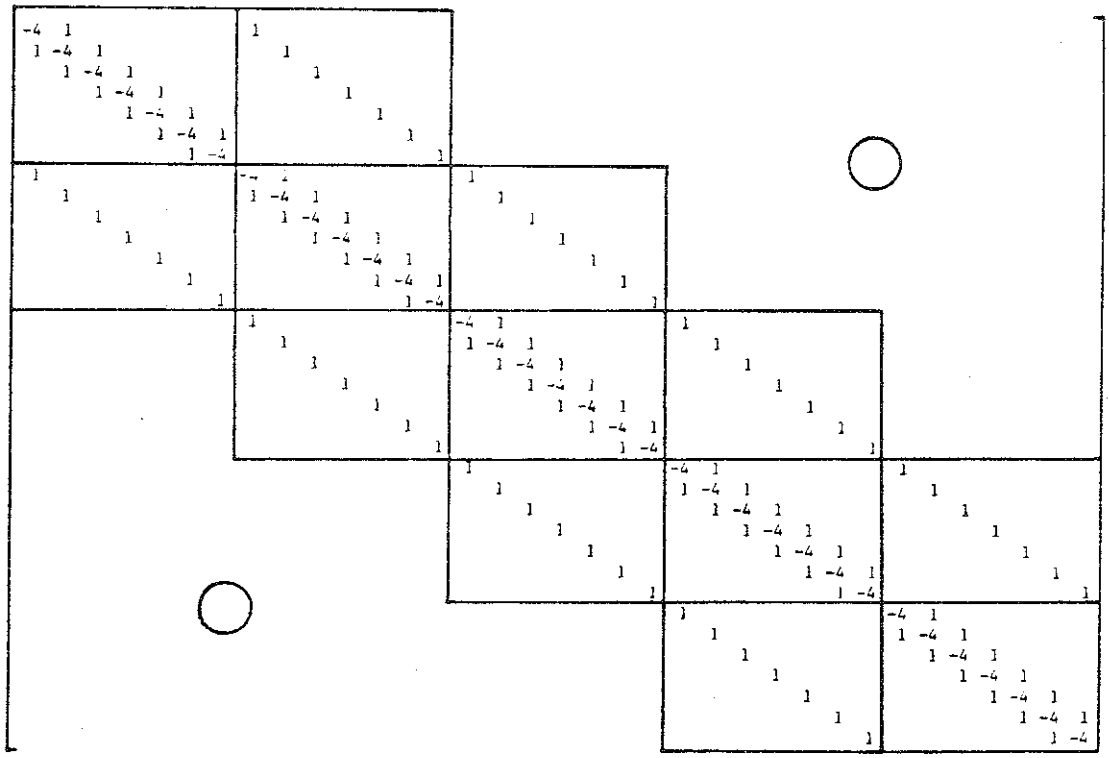


Figure 2.2 The coefficient matrix of the system of linear equations resulting from the example of Figure 2.1 is (35×35) and block-tridiagonal in structure. Each of the diagonal blocks is tridiagonal; each of the off-diagonal blocks is diagonal. The number of block equations is $n-2$=5 and the order of each block is $m-2$=7, where n and m are the number of rows and columns, respectively, of the original rectangular mesh.

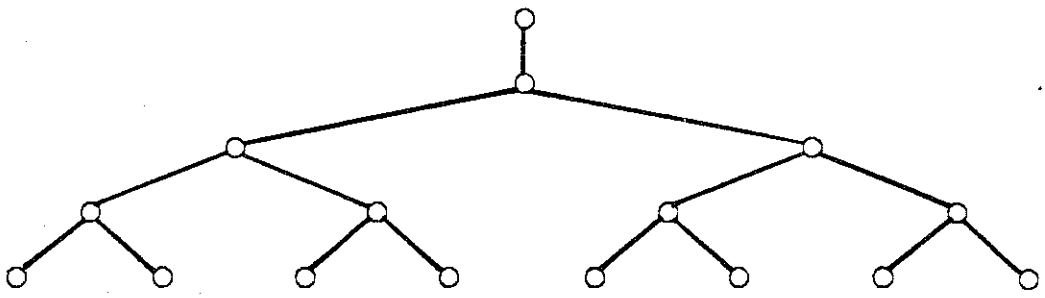


Figure 2.3 Basic structure of the tree machine. The top node is called the C cell, interior nodes are called I cells, and the leaf nodes are called L cells.

(a) **comment** Send contents of X and Y to right son.
R.send(X,Y);

(b) **comment** Receive information from father and store in Y and Z.
F.receive(Y,Z);

(c) **comment** Case statement. Communicate with first available cell.
begin
(c.1) case L.receive(X): n:=n+1
(c.2) case n≠0 & R.receive(Y): n:=n−1
(c.3) case n≥0 & F.send(Z): n:=0
end;

(d) **comment** Concurrent execution. Send contents of X and Y to sons.
L.send(X), R.send(Y);

Figure 2.4 Sample statements.

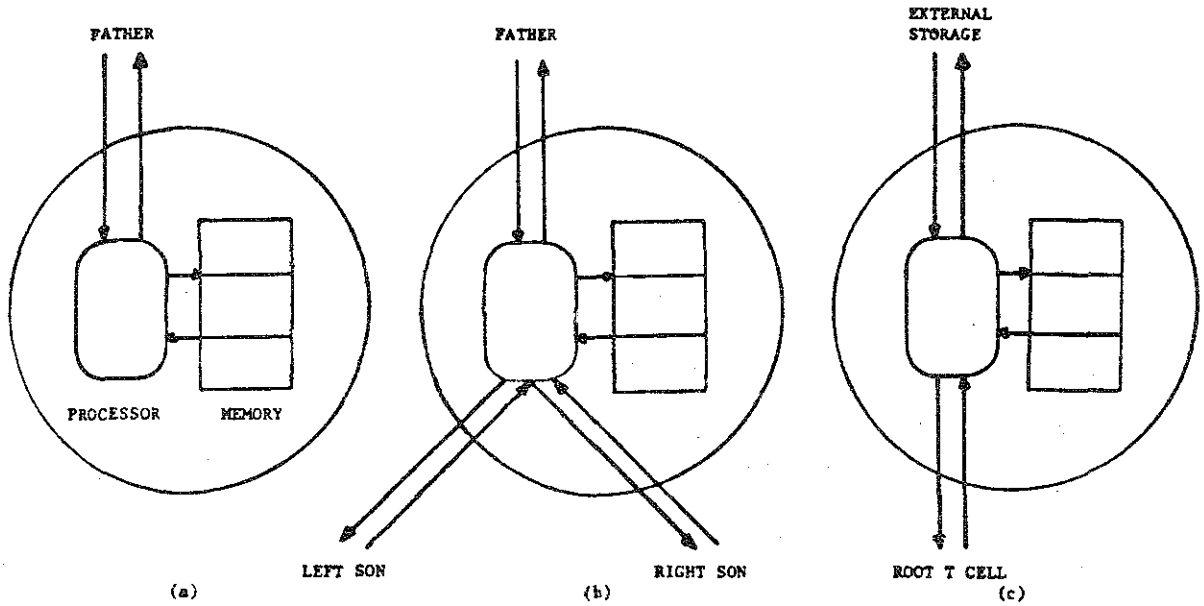


Figure 2.5 A cell (L, T, or C) contains a processor and a memory, divided into three compartments. (a) An L cell communicates only with its father. (b) A T cell communicates with its father and both sons. (c) A C cell communicates with its son, the root T cell, and with external storage from which it obtains the microprograms which it sends down to the T and L cells.

(a) L-CONTROL:

```
begin
        F.receive (microprogram set).
        Pick out and store L cell microprogram in memory.
        Execute L cell microprogram.
end;
```

(b) T-CONTROL:

```
begin
        F.receive (microprogram set).
        Send microprogram to each son, while storing T cell
          microprogram in memory.
        Execute T cell microprogram.
end;
```

C-CONTROL:

```
begin
        Fetch L, T, and C cell microprograms from external storage.
        Store C cell microprogram in memory.
        Send (microprogram set).
        Execute C cell microprogram.
end;
```

Figure 2.6 Control programs for (a) L cells, (b) T cells, (c) C cell.

(a) L-COUNT:

```
begin
    comment L cells send up VALUE and receive SUM of values
    F.send (VALUE);    F.receive (SUM);

    comment Compare SUM with twice VALUE.  Send result to father
    if 2*VALUE > SUM
        then F.send (1)    else F.send (0);

    comment NUM is the number of L cells whose values exceed
    half their SUM
    F.receive (NUM);

    comment Signal end of algorithm.
    F.send ("DONE")
end;
```

(b) T-COUNT:

```
begin
    comment L cells sent up VALUEs, send sum to C cell
    L.receive (LVAL), R.receive (RVAL);   F.send (LVAL + RVAL);

    comment C cell sent sum of values down, propagate to L cells
    F.receive (FVAL);   L.send (FVAL), R.send (FVAL);

    comment L cells sent up 1s and 0s, send up sum to C cell
    L.receive (LVAL), R.receive (RVAL);   F.send (LVAL + RVAL);

    comment C cell sent sum of 1s and 0s, propagate to L cells
    F.receive (FVAL);   L.send (FVAL), R.send (FVAL);

    comment Propagate "DONE" signal.
    L.receive (LSIGNAL), R.receive (RSIGNAL);
    if LSIGNAL=RSIGNAL="DONE"
        then F.send ("DONE")
        else F.send ("ERROR")
end;
```

(c) C-COUNT:

```
begin
    comment Receive and return the sum of VALUEs
    receive (SUM); send (SUM);

    comment Receive and return the NUM of selected L cells
    receive (SUM); send (SUM);

    comment Receive "DONE" signal.
    F.receive (SIGNAL);
    if SIGNAL≠"DONE" then ERROR
end;
```

Figure 2.7 L, T, and C cell microprograms for example algorithm.

COUNT								
Step	Swps	Comm. Time	Parallel Operations					
			L cells		T cells		C cell	
			+	×	+	×	+	×
1	1	$2\log n + 2$	0	0	$\log n$	0	0	0
2	1	$2\log n + 2$	0	1	$\log n$	0	0	0
Total	2	$4\log n + 4$	0	1	$2\log n$	0	0	0

Figure 2.8 Analysis of the COUNT program of Figure 2.7. Step (1) requires each cell to send one number to its father during the upward sweep and each cell to send one number to its son(s) during the downward sweep. The total communication time from L cells to C cell and back is $2(\log n + 1)$ units or steps. During the upward sweep, each T cell must perform one addition. As there are $\log n$ levels of T cells, there are a total of $\log n$ parallel additions performed. Step (2) is analyzed similarly. Note that a sequential algorithm would have required $2n$ additions, 1 division, and $2n$ array references (to be compared with the number of communication steps in the tree machine algorithm).

CHAPTER 3. Basic Tree Algorithms

A. Introduction

The solution of tridiagonal and block-tridiagonal linear systems can be decomposed into problems of solving low-order recurrence expressions. On a tree machine, such tridiagonal and block-tridiagonal system solvers require the L cells to communicate in certain special patterns. The purpose of this chapter is to present the tree machine solutions to these recurrence and communication problems. The resulting algorithms form the basic building blocks of the algorithms presented in Chapters 4 and 5.

In Section B, we present a general method for obtaining the first n terms of recurrence expressions on tree machines in O(log n) time. Section C presents ROTLA, an O(log n) communication algorithm developed by H.A. Presnell. ROTLA efficiently executes the FFP primitive ROTL [Back78] when applied to a vector of atomic elements on a tree machine. Section D presents a general communication technique, GDCA, which allows the L cells to communicate efficiently in a varied number of patterns. The communication time for GDCA depends on the pattern; whenever possible, the time is less than linear in the number of L cells participating. Both ROTLA and GDCA have been presented in a previous paper [PrPa81].

B. Composition and Substitution

1. Overview

Composition and Substitution is a method that enables one to solve a class of problems in a single sweep up and down the tree. This class includes homogeneous and inhomogeneous linear recurrences with variable or constant coefficients, continued and partial fractions, and recurrences of the form

$$\begin{aligned} x_0 &= a_0 \\ x_i &= \frac{a_i + b_i x_{i-1}}{c_i + d_i x_{i-1}} \qquad i = 1,2,\ldots,n-1 . \end{aligned} \tag{3.1}$$

We describe three tree machine algorithms, LR1, FRACTION, and LR2, which solve first-order linear recurrences, recurrences of the form (3.1), and second-order linear recurrences, respectively, on a tree machine, each in $O(\log n)$ execution time. (By *solve*, we mean compute the first n terms, given n.) We then show how Composition and Substitution can be extended to solve higher-order recurrence expressions.

Much study has gone into the parallel solution of linear recurrence expressions. In a paper on the parallel solution of tridiagonal linear systems, Stone [Ston73a] introduced a method called recursive doubling, which allows the user to solve linear recurrences of all orders in $O(\log n)$ time on a parallel processor of the ILLIAC-IV type. The method was generalized by Kogge and Stone [KoSt73] and by Kogge [Kogg74] who described a broad class of functions which enjoy special composition properties and for which the method is applicable. Kogge [Kogg73] described how to pipeline the method to obtain the maximal computational rate.

Papers dealing with the relationship between computation time and number of processors [ChKu75, Chen76, ChSa75, HyKu77, ChKS78 and Gajs81] have

presented bounds on the number of processors required to minimize the time to solve first-order linear recurrences and bounds on the time required to solve the problem given a fixed number of processors. Except for the algorithm described by Gajski [Gajs81], the algorithms were designed for an idealized p-processor machine on which there is no contention for memory (to obtain either instructions or data), any number of processors and memories may be used at any time, and communication among processors involves no delay. Hyafil and Kung [HyKu77] established that, even with an idealized parallel processor, the best speedup[1] one may obtain when solving first-order linear recurrences is $(2/3)p + 1/3$. In a related work [Kung76], Kung established that "many nonlinear recurrences can be sped up by at most a constant factor, no matter how many processors are used."

Two general approaches to the problem have emerged: one approach reorders the arithmetic operations required to solve the linear recurrence and distributes them among the available processors in order to minimize computation time [ChKu75, Chen76, ChSa75, ChKS78]; the other uses function composition systematically to reduce the dependencies among the variables of the linear recurrence [Ston73a, KoSt73, Kogg73, Kogg74, Gajs81]. The algorithms described in the next section use the latter approach.

2. Parallel Solution of Recurrence Expressions

This section describes how properties of recurrence expressions may be exploited to solve such expressions in parallel on a tree machine. As an example, we present the tree machine algorithm LR1 which determines the first n

[1]Speedup is defined as $S_p = T_1/T_p$ where T_1 and T_p are the amounts of time required to solve a problem on a sequential processor and a p-processor machine, respectively.

values of a first-order linear recurrence. Some of the material presented here was previously studied by Kogge [Kogg74] who described algorithms to solve recurrences on an SIMD-type parallel processor.

Consider the first-order linear recurrence

$$\begin{aligned} \mathbf{x}_0 &= a_0 \\ \mathbf{x}_1 &= a_1 + b_1\mathbf{x}_0 \\ &\cdots \\ \mathbf{x}_{n-1} &= a_{n-1}+b_{n-1}\mathbf{x}_{n-2} \end{aligned} \tag{3.2}$$

The objective is to compute the values $\mathbf{x}_i$, $0 \le i \le n-1$. To provide a uniformity which will simplify the tree machine algorithm, we modify (3.2) by defining

$$\mathbf{x}_0 = a_0 + b_0\mathbf{x}_{-1} \tag{3.3}$$

where $b_0=0$ and $\mathbf{x}_{-1}$ is a dummy variable. Each equation of (3.2) is now a function of one variable. Next, we define $\mathbf{X}_{i,j}$ to be the coefficients of the equation expressing $\mathbf{x}_i$ as a function of $\mathbf{x}_j$ $(i \ge j)$, i.e., if

$$\mathbf{x}_i = a + b\,\mathbf{x}_j \tag{3.4}$$

then

$$\mathbf{X}_{i,j} = (a,b) \tag{3.5}$$

We may now express (3.2) in the more general form

$$\mathbf{x}_i = f(\mathbf{X}_{i,i-1}, \mathbf{x}_{i-1}), \quad 0 \le i \le n-1 \tag{3.6}$$

where $\mathbf{X}_{i,i-1} = (a_i, b_i)$. Equation (3.6) may be expanded as follows

$$
\begin{aligned}
\mathbf{x}_i &= f^{i}(\mathbf{X}_{i,i-1}, \mathbf{x}_{i-1}) \\
&= f(\mathbf{X}_{i,i-1}, f(\mathbf{X}_{i-1,i-2}, \mathbf{x}_{i-2})) \\
&= f(\mathbf{X}_{i,i-1}, f(\mathbf{X}_{i-1,i-2}, f(\mathbf{X}_{i-2,i-3}, \mathbf{x}_{i-3}))) \\
&= \quad \cdots \\
&= f(\mathbf{X}_{i,i-1}, f(\mathbf{X}_{i-1,i-2}, f(\mathbf{X}_{i-2,i-3}, \ldots, f(\mathbf{X}_{0,-1}, \mathbf{x}_{-1})\ldots)))
\end{aligned}
\tag{3.7}
$$

This corresponds to the nested equation

$$
\mathbf{x}_i = a_i + b_i(a_{i-1} + b_{i-1}(a_{i-2} + b_{i-2}(\cdots (a_0 + b_0\mathbf{x}_{-1}) \cdots))) \tag{3.8}
$$

which suggests the order of operations executed by a sequential algorithm. However, we will exhibit a function g such that, for a given f and for all i,j,k, $-1 \le k < j < i \le n-1$,

$$
f(\mathbf{X}_{i,j}, \mathbf{x}_j) = f(\mathbf{X}_{i,j}, f(\mathbf{X}_{j,k}, \mathbf{x}_k)) = f(g(\mathbf{X}_{i,j}, \mathbf{X}_{j,k}), \mathbf{x}_k) = f(\mathbf{X}_{i,k}, \mathbf{x}_k) \tag{3.9}
$$

Using g, we then we show that a more efficient, parallel solution is possible. To determine g for first-order linear recurrences, consider two linear equations

$$
\begin{aligned}
\mathbf{x}_i &= a + b\,\mathbf{x}_j = f(\mathbf{X}_{i,j}, \mathbf{x}_j) \\
\mathbf{x}_j &= a' + b'\mathbf{x}_k = f(\mathbf{X}_{j,k}, \mathbf{x}_k)
\end{aligned}
\tag{3.10}
$$

Substituting the second equation into the first, we obtain

$$
\begin{aligned}
\mathbf{x}_i &= f(\mathbf{X}_{i,j}, f(\mathbf{X}_{j,k}, \mathbf{x}_k)) \\
&= a + b(a' + b'\mathbf{x}_k) \\
&= (a + ba') + bb'\mathbf{x}_k \\
&= f(g(\mathbf{X}_{i,j}, \mathbf{X}_{j,k}), \mathbf{x}_k).
\end{aligned}
\tag{3.11}
$$

Therefore, a function g that satisfies equation (3.9) for a first-order linear recurrence f is

$$
g(\mathbf{X}_{i,j}, \mathbf{X}_{j,k}) = \mathbf{X}_{i,k} = (a + ba', bb'). \tag{3.12}
$$

We call g a *composition* function because it takes the coefficients of two equations of a recurrence expression and performs the operations required to compose them. We make a few observations regarding the functions f and g.

Lemma 3.1

Given a recurrence expression $\mathbf{x}_i = f(\mathbf{X}_{i,i-1}, \mathbf{x}_{i-1})$, $0 \le i \le n-1$, and a function g that satisfies equation (3.9), each variable $\mathbf{x}_i$ can be expressed as a linear function of any $\mathbf{x}_j$, $-1 \le j < i \le n-1$. I.e., given $j < i$, we can find the coefficients $\mathbf{X}_{i,j}$ such that $\mathbf{x}_i = f(\mathbf{X}_{i,j}, \mathbf{x}_j)$

Proof

For $j < i$, the $(i-j)$th line of equation (3.7) gives

$$\begin{aligned} \mathbf{x}_i &= f(\mathbf{X}_{i,i-1}, f(\mathbf{X}_{i-1,i-2}, f(\mathbf{X}_{i-2,i-3}, \ldots, f(\mathbf{X}_{j+1,j}, \mathbf{x}_j)\ldots))) \\ &= f(g(\mathbf{X}_{i,i-1}, g(\mathbf{X}_{i-1,i-2}, \\ &\qquad g(\mathbf{X}_{i-2,i-3}, \ldots, g(\mathbf{X}_{j+2,j+1}, \mathbf{X}_{j+1,j})\ldots))), \mathbf{x}_j) \\ &= f(\mathbf{X}_{i,j}, \mathbf{x}_j) \end{aligned} \tag{3.13}$$

by repeated use of the property of g described by equation (3.9). ∎

Lemma 3.2

If $\mathbf{x}_i$ is expressed in terms of the dummy variable $\mathbf{x}_{-1}$ (as is $\mathbf{x}_0$ initially) then $\mathbf{x}_i$ is solved.

Proof

Equation (3.8) expresses $\mathbf{x}_i$ as a linear function of $\mathbf{x}_{-1}$. Expanding (3.8) and substituting $b_0 = 0$, we obtain

$$\begin{aligned} \mathbf{x}_i &= (a_i + b_i a_{i-1} + b_i b_{i-1} a_{i-2} + \cdots + a_0 \prod_{j=1}^{i} b_j) + (\prod_{j=0}^{i} b_j)\mathbf{x}_{-1} \\ &= (a_i + b_i a_{i-1} + \cdots + a_0 \prod_{j=1}^{i} b_j) \end{aligned} \tag{3.14}$$

i.e., $\mathbf{x}_i$ is expressed in terms of the given coefficients b_i and parameter a_0. The value of $\mathbf{x}_i$ is determined. ∎

Lemma 3.3

Let $\mathbf{x}_i = f(\mathbf{X}_{i,i-1}, \mathbf{x}_{i-1})$, $0 \le i \le n-1$, be a recurrence expression and g a function that satisfies

$$f(\mathbf{X}_{i,j}, f(\mathbf{X}_{j,k}, \mathbf{x}_k)) = f(g(\mathbf{X}_{i,j}, \mathbf{X}_{j,k}), \mathbf{x}_k) \tag{3.15}$$

where $0 \le k < j < i \le n-1$, then, for $-1 \le l < k < j < i \le n-1$,

$$f(g(\mathbf{X}_{i,j}, g(\mathbf{X}_{j,k}, \mathbf{X}_{k,l}))) = f(g(g(\mathbf{X}_{i,j}, \mathbf{X}_{j,k}), \mathbf{X}_{k,l})). \tag{3.16}$$

and we say that *g is associative under f.*

Proof

From Lemma 3.1 and equation (3.7) we can show that

$$\mathbf{x}_i = f(\mathbf{X}_{i,j}, f(\mathbf{X}_{j,k}, f(\mathbf{X}_{k,l}, \mathbf{x}_l))). \tag{3.17}$$

Applying (3.15) twice, we obtain

$$\begin{aligned} \mathbf{x}_i &= f(\mathbf{X}_{i,j}, f(g(\mathbf{X}_{j,k}, \mathbf{X}_{k,l}), \mathbf{x}_l)) \\ &= f(g(\mathbf{X}_{i,j}, g(\mathbf{X}_{j,k}, \mathbf{X}_{k,l})), \mathbf{x}_l) \end{aligned} \tag{3.18}$$

Applying (3.15) twice to (3.17) in a different order, we obtain

$$\begin{aligned} \mathbf{x}_i &= f(g(\mathbf{X}_{i,j}, \mathbf{X}_{j,k}), f(\mathbf{X}_{k,l}, \mathbf{x}_l)) \\ &= f(g(g(\mathbf{X}_{i,j}, \mathbf{X}_{j,k}), \mathbf{X}_{k,l}), \mathbf{x}_l) \end{aligned} \tag{3.19}$$

Hence, g is associative under f. ■

Lemma 3.3 provides the key to developing parallel solutions of recurrences. It allows for the regrouping of the operations required by equation (3.7). For example, to solve the following equation for $\mathbf{x}_3$

$$\mathbf{x}_3 = f(\mathbf{X}_{3,2}, f(\mathbf{X}_{2,1}, f(\mathbf{X}_{1,0} f(\mathbf{X}_0, \mathbf{x}_{-1})))) \tag{3.20}$$

we may apply equation (3.9) and Lemma 3.3 to transform equation (3.20) into

$$\mathbf{x}_3 = f(g(g(\mathbf{X}_{3,2}, \mathbf{X}_{2,1}), g(\mathbf{X}_{1,0}, \mathbf{X}_{0,-1})), \mathbf{x}_{-1}) \tag{3.21}$$

suggesting a parallel solution: simultaneously evaluate $\mathbf{X}_{3,1} = g(\mathbf{X}_{3,2}, \mathbf{X}_{2,1})$ and $\mathbf{X}_{1,-1} = g(\mathbf{X}_{1,0}, \mathbf{X}_{0,-1})$ and then evaluate $\mathbf{X}_{3,-1} = g(\mathbf{X}_{3,1}, \mathbf{X}_{1,-1})$. The first component of the pair $\mathbf{X}_{3,-1}$ is the value of $\mathbf{x}_3$.

The Tree Machine Algorithm: LR1

On a sequential computer, the solution of (3.2) requires $O(n)$ time. We now describe the tree machine algorithm, LR1, which solves equation (3.2) in a single upward and downward sweep. LR1 requires each tree cell to perform a constant amount of computation. Because there are $O(\log n)$ levels of tree cells, the algorithm executes in $O(\log n)$ time. Let $n = 2^p$ for p a nonnegative integer. This choice of n is purely for ease of presentation, since LR1 works for any positive value of n as shown later. Let L_i be the ith occupied L cell counting from the right. (For this example, all L cells are occupied.) We store the coefficients of the ith equation, i.e., $\mathbf{X}_{i,i-1}$, in L_i. Figure 3.1 shows the initial values stored in

the L cells of the tree machine for n=8. Recall that $\mathbf{X}_{0,-1}$, stored in the rightmost L cell, is the pair $(a_0,0)$.

The L cells start the upward (composition) sweep by sending their coefficients to their fathers. A T cell (Figure 3.2) receives $\mathbf{X}_{i,j} = (a_L,b_L)$ and $\mathbf{X}_{j,k} = (a_R,b_R)$ from its left and right sons, respectively. The T cell composes $\mathbf{X}_{i,j}$ with $\mathbf{X}_{j,k}$ and sends the result,

$$\mathbf{X}_{i,k} = g(\mathbf{X}_{i,j}, \mathbf{X}_{j,k}) = (a_L + b_L a_R,\ b_L b_R) \tag{3.22}$$

to its father. The T cell is unaware of the identity of the coefficients it receives from its sons. Every T cell simply receives a pair of coefficients from each of its sons, operates on them in the manner described, saves $\mathbf{X}_{j,k}$ (the coefficients received from its right son), and sends $\mathbf{X}_{i,k}$ (the coefficients produced) to its father.

The upward sweep ends when the C cell receives $\mathbf{X}_{n-1,-1}$ from the root T cell. From Lemma 3.2, we know that the value of $\mathbf{x}_{n-1}$ has been determined. During the downward sweep, we can compute the remaining $\mathbf{x}_i$'s.

The downward (substitution) sweep begins when the C cell sends to the root T cell the pair $(x, 0)$. The first component is the value of $\mathbf{x}_{n-1}$; the second is the value of the dummy variable $\mathbf{x}_{-1}$. In general, the T cell that received $\mathbf{X}_{i,j}$ and $\mathbf{X}_{j,k}$ during the upward sweep, receives the pair $(\mathbf{x}_i,\mathbf{x}_k)$ during the downward sweep (Figure 3.3). The first component is the solution of the leftmost L cell in its left subtree. The second component is used to obtain $\mathbf{x}_j$, the solution of the leftmost L cell in its right subtree. The T cell computes $\mathbf{x}_j$ by substituting $\mathbf{x}_k$ in the equation represented by $\mathbf{X}_{j,k}$, i.e.,

$$\mathbf{x}_j = a_R + b_R \mathbf{x}_k. \tag{3.23}$$

It then sends $(\mathbf{x}_i, \mathbf{x}_j)$ and $(\mathbf{x}_j, \mathbf{x}_k)$ to its left and right sons respectively.

The downward sweep ends with L_i receiving $(\mathbf{x}_i, \mathbf{x}_{i-1})$. The L cell saves $\mathbf{x}_i$ and may or may not save $\mathbf{x}_{i-1}$ (in some applications, the L cell uses $\mathbf{x}_{i-1}$ in a later computation). The first n terms of the recurrence relation have now been found. Figures 3.4 and 3.5 show the full upward and downward sweeps for $n=8$.

We prove the correctness of LR1 for $n=2^p$, p a positive integer, with the following lemmas. The lemmas actually prove a stronger statement: that composition and substitution correctly solve a general recurrence expression $\mathbf{x}_i = f(\mathbf{X}_{i,i-1}, \mathbf{x}_{i-1})$, $0 \le i \le 2^p - 1$ for which a composition function g can be found. We assume that L_i ($0 \le i \le n-1$, counting from the right) initially contains $\mathbf{X}_{i,i-1}$.

Lemma 3.4

Let $\mathbf{x}_i = f(\mathbf{X}_{i,i-1}, \mathbf{x}_{i-1})$, $0 \le i \le 2^p - 1$, be a recurrence expression for which a composition function g is known. Let T be a T cell. Let T_1 and T_2 be its left and right sons, L_1 and L_2 the leftmost and rightmost L cells in its left subtree, and L_3 and L_4 its leftmost and rightmost L cells in its right subtree. Let L_1, L_2, L_3, and L_4 initially contain $\mathbf{X}_{i,i-1}$, $\mathbf{X}_{j+1,j}$, $\mathbf{X}_{j,j-1}$, and $\mathbf{X}_{k+1,k}$, respectively. Then, during the upward sweep, T receives $\mathbf{X}_{i,j}$ from its left son and $\mathbf{X}_{j,k}$ from its right son and sends $\mathbf{X}_{i,k}$ to its father.

Proof

Proof by induction on the level number of the T cells. (T cells that are fathers of L cells are on level 1, their fathers are on level 2, ..., the root T cell is on level $\log N$ where N is the number of L cells in the tree.) ∎

Lemma 3.4 proves the correctness of the upward (composition) sweep. From Lemma 3.4, we conclude that the root T cell sends the coefficient set $\mathbf{X}_{n-1,-1}$ to its father, the C cell. The C cell is then able to determine the solution of $\mathbf{x}_{n-1}$. Lemma 3.5 proves the correctness of the downward (substitution) sweep and shows that the L cells receive the correct values.

Lemma 3.5

If a T cell received $\mathbf{X}_{i,j}$ and $\mathbf{X}_{j,k}$ from its sons during the upward sweep, then, during the downward sweep, T will receive $\mathbf{x}_i$ and $\mathbf{x}_k$ from its father.

It then uses x_k to solve for x_j and sends (x_i, x_j) to its left son and (x_j, x_k) to its right son.

Proof

Proof by induction on the level number of the T cells. (Start with the root T cell and proceed downward.) ∎

Lemma 3.6

LR1 is correct.

Proof

Follows directly from Lemmas 3.4 and 3.5. ∎

The analysis of LR1 is shown in Figure 3.6. During the upward sweep, each L cell and each T cell sends two values to their fathers. Because there are $\log n$ levels of T cells, the total communication time required during the upward sweep is $2(\log n + 1)$ units or steps, where one unit is the time required for a cell to send one value to an adjacent cell. Similarly, during the downward sweep, the C cell and T cells send two values to each of their sons. The time required during the downward sweep is also $2(\log n + 1)$ steps. Only the T cells perform arithmetic operations. A T cell performs one addition and two multiplications (3.22) during the upward sweep and one addition and one multiplication (3.23) during the downward sweep. Thus, LR1 requires a total of $2\log n$ parallel additions and $3\log n$ parallel multiplications for both sweeps.

We may use a similar algorithm to solve the first-order backward recurrence relation

$$\begin{aligned} x_{n-1} &= a_{n-1} \\ x_{n-2} &= a_{n-2} + b_{n-2}x_{n-1} \\ &\cdots \\ x_0 &= a_0 + b_0 x_1 . \end{aligned} \tag{3.24}$$

Called BLR1, it has the same time complexity as LR1.

Extensions

We end this section by describing variations of the basic LR1 algorithm. Similar variations of FRACTION and LR2 are also possible.

Empty L cells. If the desired number (n) of terms of the recurrence relation is not a power of 2, we must use a tree with 2^p L cells where $p = \lceil \log n \rceil$. Some of the L cells will be empty and will not participate productively. When distributing the initial coefficient pairs among the L cells, $\mathbf{X}_{0,-1}$ is stored in the rightmost occupied L cell, $\mathbf{X}_{1,0}$ is stored in the next occupied L cell to the left, and so on. Surprisingly, we do not need to modify the T cell algorithm described above. We must describe, however, what the empty L cells are to do.

During the upward sweep, while occupied L cells send up their coefficient pairs, empty L cells send up the pair (0, 1). This has the effect of defining the "equation" stored in an empty L cell as identical to the equation stored in the first occupied L cell to its right. A T cell does not know whether the data it receives is from an empty or an occupied L cell. It simply performs the two multiplications and one addition described in Figure 3.2 during the upward sweep. If the pair of values from one of its sons came from an empty L cell, a T cell effectively sends the other son's values to its father. Similarly, during the downward sweep, a T cell blindly performs the multiplication and addition described in Figure 3.3. L_i still receives the values $\mathbf{x}_i$ and $\mathbf{x}_{i-1}$; empty L cells ignore the values they receive. Figure 3.7 shows the execution of LR1 for n=5 on a tree with 8 L cells.

Solving several independent recurrences simultaneously. The algorithm is, in fact, more powerful than described in the previous paragraph. If there are enough L cells to accommodate two or more linear recurrences (with one L cell

containing at most one equation of one linear recurrence) we may solve all of the linear recurrences simultaneously.

Let two or more linear recurrences be stored in disjoint segments of the L array. Each equation of each recurrence has a pair of coefficients. We distribute the coefficient pairs as follows. Starting with the first equation of the first recurrence, we distribute the coefficients pairs of the first recurrence one pair to an L cell. After storing the coefficient pair of the last equation of the first recurrence, we continue with the coefficient pair of the first equation of the second recurrence, and so on. We may consider the entire initial set of coefficient pairs to be the terms of one large linear recurrence and apply LR1 to all of the L cells. Because the first coefficient pair of each recurrence is of the form $(a, 0)$, we are sure that the terms of one linear recurrence will not be affected by the terms of another. We may therefore store as many linear recurrences as the L cells can hold, apply LR1, and in a single sweep, solve all recurrences simultaneously.

3. Quotients of Linear Recurrences

Consider recurrence expressions of the form

$$\begin{aligned} \mathbf{x}_0 &= a_0 \\ \mathbf{x}_i &= \frac{a_i + b_i \mathbf{x}_{i-1}}{c_i + d_i \mathbf{x}_{i-1}} \qquad i = 1,2,\ldots,n-1 . \end{aligned} \tag{3.25}$$

where c_i and d_i are not both 0. As with first-order linear recurrences, we let

$$\mathbf{x}_0 = \frac{a_0 + b_0 \mathbf{x}_{-1}}{c_0 + d_0 \mathbf{x}_{-1}} \tag{3.26}$$

where $b_0 = d_0 = 0$, and $c_0 = 1$. Moreover, if

$$\mathbf{x}_i = \frac{a + b\,\mathbf{x}_j}{c + d\,\mathbf{x}_j} \tag{3.27}$$

we say that $\mathbf{X}_{i,j} = (a, b, c, d)$. Therefore we may express (3.25) as

$$\mathbf{x}_i = f(\mathbf{X}_{i,i-1}, \mathbf{x}_{i-1}) = \frac{a_i + b_i \mathbf{x}_{i-1}}{c_i + d_i \mathbf{x}_{i-1}}, \quad 0 \le i \le n-1 \tag{3.28}$$

where $\mathbf{X}_{i,i-1} = (a_i, b_i, c_i, d_i)$.

To determine the corresponding composition function g, we observe that if

$$\mathbf{x}_i = \frac{a + b\,\mathbf{x}_j}{c + d\,\mathbf{x}_j} \tag{3.29}$$

and

$$\mathbf{x}_j = \frac{a' + b'\mathbf{x}_k}{c' + d'\mathbf{x}_k} \tag{3.30}$$

then

$$\mathbf{x}_i = \frac{a + b\dfrac{a' + b'\mathbf{x}_k}{c' + d'\mathbf{x}_k}}{c + d\dfrac{a' + b'\mathbf{x}_k}{c' + d'\mathbf{x}_k}} \tag{3.31}$$

$$= \frac{a(c' + d'\mathbf{x}_k) + b(a' + b'\mathbf{x}_k)}{c(c' + d'\mathbf{x}_k) + d(a' + b'\mathbf{x}_k)}$$

$$= \frac{(ac' + ba') + (ad' + bb')\mathbf{x}_k}{(cc' + da') + (cd' + db')\mathbf{x}_k}$$

The desired function is

$$g(\mathbf{X}_{i,j}, \mathbf{X}_{j,k}) = \mathbf{X}_{i,k} = (ac' + ba', ad' + bb', cc' + da', cd' + db'). \tag{3.32}$$

Equation (3.28) describes a recurrence expression whose composition function is defined in (3.32). Using Lemmas 3.4 and 3.5, we can develop a tree

algorithm, which we call FRACTION, to solve (3.28) in one upward and downward sweep through the tree. FRACTION determines the first n terms of (3.28). For ease of presentation, assume that $n = 2^p$. L_i, counting from the right, initially contains the quadruple $\mathbf{X}_{i,i-1} = (a_i, b_i, c_i, d_i)$; the rightmost L cell contains $\mathbf{X}_{0,-1} = (a_0, 0, 1, 0)$.

Composition starts with the L cells sending $\mathbf{X}_{i,i-1}$. A T cell receives

$$\mathbf{X}_{i,j} = (a_L, b_L, c_L, d_L) \tag{3.33}$$

from its left son, and

$$\mathbf{X}_{j,k} = (a_R, b_R, c_R, d_R) \tag{3.34}$$

from its right son. As described by equation (3.32), the T cell computes and sends

$$\mathbf{X}_{i,k} = (a_L c_R + b_L a_R,\ a_L d_R + b_L b_R,\ c_L c_R + d_L a_R,\ c_L d_R + d_L b_R) \tag{3.35}$$

to its father. The upward sweep ends with the C cell receiving the quadruple $(a, 0, c, 0)$ where a/c is the solution of $\mathbf{x}_{n-1}$, the leftmost L cell in the tree. This ends the composition sweep.

The C cell divides a by c and returns the pair $(a/c, 0)$ to the root T cell starting the substitution sweep. As in LR1, the second component is the value of the dummy variable $\mathbf{x}_{-1}$ and the first component is the value of $\mathbf{x}_{n-1}$. The T cell that received $\mathbf{X}_{i,j}$ and $\mathbf{X}_{j,k}$ from its left and right sons during the upward sweep receives the pair $(\mathbf{x}_i, \mathbf{x}_k)$ from its father. It determines $\mathbf{x}_j$ by substituting $\mathbf{x}_k$ into the equation represented by $\mathbf{X}_{j,k}$, i.e.

$$\mathbf{x}_j = \frac{a_R + b_R \mathbf{x}_k}{c_R + d_R \mathbf{x}_k} \tag{3.36}$$

and sends $(\mathbf{x}_i, \mathbf{x}_j)$ to its left son and $(\mathbf{x}_j, \mathbf{x}_k)$ to its right son. L_i receives the pair $(\mathbf{x}_i, \mathbf{x}_{i-1})$. Using Lemmas 3.4 and 3.5, we prove the correctness of the tree algorithm FRACTION.

Lemma 3.7
FRACTION is correct.

Proof
Follows from Lemmas 3.4 and 3.5. ∎

The analysis of FRACTION is summarized in Figure 3.8. During the upward sweep, L cells and T cells send a quadruple to their fathers, requiring a total of $4\log n + 4$ communication steps. Each T cell also performs 4 additions and 8 multiplications. The C cell performs one division. During the downward sweep, the C cell and the T cells send a pair of values to each of their sons, requiring a total of $2\log n + 2$ communication steps. Each T cell performs 2 additions and 3 multiplications or divisions.

Extensions

The variations to LR1 presented in the previous section may also be applied to FRACTION in a similar manner. To accommodate empty L cells, we program empty L cells to send the quadruple (0, 1, 1, 0) to their fathers during the upward sweep. This has the effect of defining the "equation" stored in an empty L cell as identical to the first nonempty L cell to its right. Because of the associativity property enjoyed by the composition functions, this has no effect on the nonempty L cells. A T cell need not know whether the information it receives came from an empty or a nonempty L cell.

4. Second- and Higher-Order Linear Recurrences

Composition and Substitution may be extended to higher-order recurrences. We present the tree algorithm LR2, the solution for the first n terms of a second-order linear recurrence; the generalization to third and higher-order recurrences is straightforward. Given

$$\begin{aligned} \mathbf{x}_0 &= a_0 + b_0\mathbf{x}_{-1} + c_0\mathbf{x}_{-2} \\ \mathbf{x}_1 &= a_1 + b_1\mathbf{x}_0 + c_1\mathbf{x}_{-1} \\ \mathbf{x}_2 &= a_2 + b_2\mathbf{x}_1 + c_2\mathbf{x}_0 \\ &\cdots \\ \mathbf{x}_{n-1} &= a_{n-1} + b_{n-1}\mathbf{x}_{n-2} + c_{n-1}\mathbf{x}_{n-3} \end{aligned} \tag{3.37}$$

where $\mathbf{x}_{-1}$ and $\mathbf{x}_{-2}$ are dummy variables and $b_0=c_0=c_1=0$, we want to solve for $\mathbf{x}_i$, $0\le i\le n-1$. Equation (3.37) is of the form

$$\mathbf{x}_i = f\,(\mathbf{X}_{i,i-1,i-2},\ \mathbf{x}_{i-1},\ \mathbf{x}_{i-2})\ . \tag{3.38}$$

In order to find a composition function g which satisfies equation (3.7) we use a change of variables. Define

$$\mathbf{y}_i = \begin{bmatrix} \mathbf{x}_i \\ \mathbf{x}_{i-1} \end{bmatrix} \quad A_i = \begin{bmatrix} a_i \\ 0 \end{bmatrix} \quad B_i = \begin{bmatrix} b_i & c_i \\ 1 & 0 \end{bmatrix} \quad \text{and} \quad \mathbf{X}_{i,i-1} = (A_i,\ B_i) \tag{3.39}$$

for $0\le i\le n-1$. Equation (3.37) may therefore be expressed as

$$\mathbf{y}_i = A_i + B_i\mathbf{y}_{i-1} = f\,(\mathbf{X}_{i,i-1},\ \mathbf{y}_{i-1}) \tag{3.40}$$

which is a first-order linear recurrence in y. We can now use the method of Composition and Substitution, with scalar addition and multiplication replaced by vector addition and matrix multiplication, provided we can find a suitable composition function g. Given

$$\begin{aligned} \mathbf{y}_i &= A + B\mathbf{y}_j = f\,(\mathbf{X}_{i,j},\ \mathbf{y}_j) \\ \mathbf{y}_j &= A' + B'\mathbf{y}_k = f\,(\mathbf{X}_{j,k},\ \mathbf{y}_k) \end{aligned} \tag{3.41}$$

where $\mathbf{X}_{i,j} = (A, B)$ and $\mathbf{X}_{j,k} = (A', B')$, we obtain

$$\mathbf{y}_i = A + B(A' + B'\mathbf{y}_k) = (A + BA') + BB'\mathbf{y}_k = f(\mathbf{X}_{i,k}, \mathbf{y}_k) \qquad (3.42)$$

which gives

$$g(\mathbf{X}_{i,j}, \mathbf{X}_{j,k}) = \mathbf{X}_{i,k} = (A + BA', BB'). \qquad (3.43)$$

Initially, L_i (counting from the right) contains the coefficients of the ith equation: (a_i, b_i, c_i). The two rightmost occupied L cells contain $(a_1, b_1, 0)$ and $(a_0, 0, 0)$. The L cells start the upward (composition) sweep: L_i sends the coefficients $\mathbf{X}_{i,i-1} = (A_i, B_i)$. A T cell receives $\mathbf{X}_{i,j} = (A, B)$ from its left son and $\mathbf{X}_{j,k} = (A', B')$ from its right son and applies the function g (equation (3.43)). It sends the result $\mathbf{X}_{i,k}$ to its father (Figure 3.9a). The upward sweep ends when the C cell receives $\mathbf{X}_{n-1,-1} = (A_{n-1}, B_{n-1})$. From Lemma 3.2, we know that $B_{n-1} = 0$ and $\mathbf{y}_{n-1} = A_{n-1}$. The equation stored in the leftmost L cell has been solved.

The downward (substitution) sweep determines the rest of the solutions and sends them down to the proper L cells (Figure 3.9b). The C cell starts by returning the pair of solutions $(\mathbf{y}_{n-1}, \mathbf{y}_{-1})$ to the root T cell. The second component is the vector $(0, 0)^T$, the "solution" of the dummy variable $\mathbf{y}_{-1}$. A T cell that received $\mathbf{X}_{i,j}$ and $\mathbf{X}_{j,k}$ from its left and right sons during the upward sweep receives the pair $(\mathbf{y}_i, \mathbf{y}_k)$ from its father. The value $\mathbf{y}_k$ is used to compute the solution $\mathbf{y}_j$ using the coefficients $\mathbf{X}_{j,k} = (A', B')$ retained during the upward sweep:

$$\mathbf{y}_j = A' + B'\mathbf{y}_k . \qquad (3.44)$$

The T cell then sends the pair $(\mathbf{y}_i, \mathbf{y}_j)$ to its left son and $(\mathbf{y}_j, \mathbf{y}_k)$ to its right son.

L_i receives the pair $(\mathbf{y}_i, \mathbf{y}_{i-1})$. In effect, L_i receives the solutions $\mathbf{x}_i$, $\mathbf{x}_{i-1}$, and $\mathbf{x}_{i-2}$. This ends the downward sweep.

A proof similar to that of Lemma 3.6 suffices to show the following.

Lemma 3.8
LR2 is correct. ■

The analysis of LR2 is shown in Figure 3.10. During the upward sweep, each T cell receives 6 values, the components of the pair (A, B), from each son, and sends 6 values to its father. The number of communication steps is therefore $6\log n + 6$. During the downward sweep, each T cell receives 4 values, the components of the solution pair $(\mathbf{y}_i, \mathbf{y}_k)$ from its father and sends 4 values to each of its sons. The number of communication steps is $4\log n + 4$. Only the T cells perform arithmetic operations. During the upward sweep, each T cell must apply the function g (equation (3.43)). This requires 12 multiplications and 8 additions. As there are $\log n$ levels of T cells, a total of $12\log n$ parallel multiplications and $8\log n$ parallel additions are required. During the downward sweep, each T cell must solve equation (3.44). This requires a total of $4\log n$ multiplications and $4\log n$ additions.

Extensions

If some of the L cells are empty, we program an empty L cell to send up the pair (A, B) where

$$A_i = \begin{bmatrix} 0 \\ 0 \end{bmatrix} \text{ and } B_i = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}. \tag{3.45}$$

The effect is to define the "equation" stored in the empty L cell as identical to the equation stored in the first nonempty L cell to its right. Because of the

associativity property of g, this has no effect on the solution process.

Extending the method of Composition and Substitution to higher-order linear recurrences is straightforward. To solve a kth-order linear recurrence, we transform the original equations into a linear recurrence with matrix coefficients as we did in equation (3.39). We then use a tree algorithm analogous to LR1. The resulting coefficients, however, will include $(k \times k)$ matrices and the T cell operations will involve multiplying these matrices. Each T cell, therefore, will need $O(k^2)$ storage and will perform $O(k^3)$ arithmetic operations. In order to solve the tridiagonal and block-tridiagonal linear systems described in this dissertation, we need to solve only first- and second-order linear recurrences, which, as has been shown above, can be done with cells with very modest storage capacity.

C. Atomic Rotate Left: ROTLA

Backus [Back78] describes the functions rotate left (ROTL) and rotate right (ROTR) which circularly rotate a vector of elements to the left and to the right, respectively. ROTL is defined as follows:

$$\text{ROTL} : \mathbf{x} \equiv \mathbf{x} = \langle \mathbf{x}_0 \rangle \longrightarrow \langle \mathbf{x}_0 \rangle;$$
$$\mathbf{x} = \langle \mathbf{x}_0, \mathbf{x}_1, \cdots, \mathbf{x}_{n-1} \rangle \ \& \ n \geq 2 \longrightarrow \langle \mathbf{x}_1, \cdots, \mathbf{x}_{n-1}, \mathbf{x}_0 \rangle .$$

We describe three possible ways of implementing ROTL on a tree machine. Let the vector elements be distributed among the L cells, with the L_i (counting from the left) containing $\mathbf{x}_i$. The first implementation sends $\mathbf{x}_0$ up to the C cell and down to the right of the the L cell that contains $\mathbf{x}_{n-1}$, as shown in Figure 3.11. Only one value must travel up to the C cell and back down again; thus, the

time required is $2\log n + 2$ steps. This technique is simple but it requires an empty L cell on the right of $\mathbf{x}_{n-1}$. The second implementation avoids this problem by sending all of the values up to the C cell and broadcasting them back down. L_i is programmed to receive the value $\mathbf{x}_{i+1 \pmod n}$ when it arrives. This method always works, not only for ROTL, but for an arbitrary permutation. The disadvantage is that $(2\log n + 2) + (n-1)$, or $O(n)$, steps are needed.

A third implementation is described by Presnell [PrPa81]. This tree algorithm, called ROTLA, enables L_i to receive $\mathbf{x}_{i+1 \pmod n}$ (avoiding the storage management problem of the first implementation) but requires only $2\log n + 2$ steps. One drawback is that ROTLA works only if the elements are atoms (single numbers or characters), or a small fixed-length vector of atoms, such as a pair of atoms representing a complex number whereas the function ROTL allows the $\mathbf{x}_i$'s to be arbitrary sequences. The analogous algorithm ROTRA implements the function rotate right (ROTR) with the same restrictions.

After some use, it became apparent that ROTLA could also be used to provide a means of *communication* among the L cells. Because the L cell that initially contains $\mathbf{x}_i$ receives $\mathbf{x}_{i+1 \pmod n}$ this algorithm may be used for vector operations that need to combine $\mathbf{x}_i$ and $\mathbf{x}_{i+1 \pmod n}$. As a communication tool, ROTLA gains power if some of the L cells can be masked from the operation at appropriate times. This allows the user to specify different subsets of L cells and to have members of a subset communicate exclusively among themselves. This potential use of ROTLA is the primary reason for including it in this dissertation.

ROTLA is an example of a permutation that can be implemented on a tree machine in O(log n) time. This is not true of all permutations. For example, if the tree is full (all of the L cells are occupied), the time required to reverse the order of the elements of a sequence is linear. Reversal and other permutations on a tree machine were studied by Tolle and Siddall [ToSi81].

We now describe ROTLA. We are given $\mathbf{x}=<\mathbf{x}_0,\mathbf{x}_1,\cdots,\mathbf{x}_{n-1}>$ and wish to obtain $\mathbf{x}=<\mathbf{x}_1,\cdots,\mathbf{x}_{n-1},\mathbf{x}_0>$. Some of the L cells may be empty. L_i (counting from the left) initially contains $\mathbf{x}_i$.

The L cells begin the upward sweep by sending up their x-values; empty L cells send up the distinguished symbol ϕ. Every T cell receives a value from each of its sons and sends one of the values to its father as specified by the following code:

```
L.receive(LVAL), R.receive(RVAL);
if LVAL≠φ then F.send(LVAL) else F.send(RVAL)
```

The second line of code states that the T cell should send the value received from the left son provided it is not the empty symbol ϕ. Otherwise, it should send the value received from the right son. The upward sweep ends when the C cell receives a value from the root T cell; this value is $\mathbf{x}_0$, the leftmost vector element. Figure 3.12 shows the upward sweep for ROTLA with n=5 on an eight-L-cell tree machine.

The downward sweep begins when the C cell returns $\mathbf{x}_0$ to the root T cell. Each T cell receives one value from its father and sends one value to each of its sons as specified by the following code. LVAL and RVAL, the values received from the T cell's left and right sons during the upward sweep, were stored in the T cell

for use during the downward sweep.

```
F.receive(FVAL);

if LVAL≠φ & RVAL≠φ
   then begin L.send(RVAL), R.send(FVAL); end

else if LVAL=φ & RVAL≠φ
   then begin L.send(φ), R.send(FVAL); end

else if LVAL≠φ & RVAL=φ
   then begin L.send(FVAL), R.send(φ); end

else if LVAL=φ & RVAL=φ
   then begin L.send(φ), R.send(φ); end
```

The T cell returns ϕ to the son that sent up ϕ during the upward sweep. If neither son sent up ϕ, the T cell sends RVAL to the left son and FVAL (just received from the father) to the right son. Figure 3.13 shows the downward sweep for ROTLA. The following theorem precisely describes what happens when ROTLA is executed.

Theorem: (Presnell [PrPa81]) Let the initial configuration of a tree machine M be such that the elements of an n-vector $\mathbf{x}=<\mathbf{x}_0, \mathbf{x}_1, \cdots, \mathbf{x}_{n-1}>$ are stored in the L cells with at most one element per cell. After executing the ROTLA algorithm, the L cell initially containing $\mathbf{x}_i$ contains $\mathbf{x}_{(i+1) \bmod n}$, for $i=0, 1, \ldots, n-1$. Furthermore, if $i \neq n-1$, $\mathbf{x}_{(i+1) \bmod n}$ is routed to its destination along the shortest path through the tree. $\mathbf{x}_0$ is routed up to the C cell and down to its destination.

To analyze ROTLA, note that during the upward sweep, the L cells and the T cells each send exactly one value to their fathers. If we assume that the L cells simultaneously send up their values and that T cells on one row simultaneously send up their values, the C cell receives a value from the root T cell cell after $\log n + 1$ steps. Similarly, during the downward sweep, the C cell and the T cells send one value to each of their sons. If we assume that a T cell can send one value to each of its sons simultaneously, then the L cells receive values after $\log n + 1$ steps. We see, therefore, that ROTLA requires a total of $2\log n + 2$, or

$O(\log n)$, parallel communication steps. The analysis of ROTLA is summarized in Figure 3.14.

If $\mathbf{x}_i$ is not atomic, but rather a vector of length k, we can use the same technique but simply consider the k elements to be a single data item if the k elements can be stored in a single L cell. Because the T cells can pipeline their operations, the total time required is $2\log n + 2 + (k-1)$, which is still $O(\log n)$ if $k \ll \log n$.

D. General Data Communication Algorithm: GDCA

1. Description

ROTLA circularly shifts the values of the occupied L cells a distance of 1 to the left. When constructing algorithms for a tree machine, the need for other data communication techniques quickly arises. For this purpose, we describe GDCA. We define a *communication pattern* to be a pair $<\alpha, \beta>$, where α and β are the mappings of data items onto the L array before and after "communication," where a mapping is a 1-1 function from a set of values $\{\mathbf{x}_0, \mathbf{x}_1, \ldots, \mathbf{x}_{n-1}\}$ onto the set of L cells $\{L_0, L_1, \ldots, L_{n-1}\}$. Before communication, each occupied L cell must contain a value, its index, and the value of its index after communication. This means that only communication patterns in which β is easily computable in terms of α are of practical interest, using information such as the index of the L cell, the number of values, row and column numbers associated with the values (for data obtained from a matrix), etc. For example, we may want to shift the L cell values a distance $k>1$ (Figure 3.15a). (We could do this with k applications of ROTLA but GDCA is more efficient.) As another example, we may want L_i

and L_{i+1} to exchange values (i=0,2,...,n−2; n=even) (Figure 3.15b). As a third example, we may want to "shuffle" the data in the L cells (Figure 3.15c).

GDCA is not the only way to effect a communication pattern. We could, for example, send the L cell values up to the C cell and broadcast them back down. Each L cell sends its value to its father; each T cell receives one or more values from its sons and sends them up to its father. All n values are received by the C cell which broadcasts each value down as it is received. Each L cell receives all n values but is programmed to save only one of them. (A more detailed discussion is given by Magó [Magó79a].) If we define a *step* as the time it takes for one cell to send a value to an adjacent cell, the C cell would receive the first of the n values after h+1 steps, where h stands for the height of the tree (from root to leaf). It immediately sends this value down and after another h+1 steps, the L cells receive the first value. (Note that while values are being broadcast down, the rest of the values continue to rise to the C cell.) After another n−1 steps, therefore, the L cells receive the last of the values. The total time required is

$$T(n) = n + 2h + 1$$

steps, i.e., $O(n)$ time.

Our aim here is to perform communication in less than $O(n)$ time, whenever possible. This can be achieved only if fewer than the total number of the values have to go through the C cell. Whenever this is the case, the mechanism we devise (GDCA) routes each element along the shortest path from source to destination. Each L cell value travels up the tree only as far as it has to, i.e., until it reaches the lowest common ancestor of the source and destination L cells.

The basic idea of this mechanism is that each T cell constructs a 4-register directory (LLOW, LHIGH, RLOW, RHIGH) containing the sequence numbers of the leftmost occupied L cells in its left and right subtrees (LLOW and RLOW), and the rightmost occupied L cells in its left and right subtrees (LHIGH and RHIGH). If all of the L cells of the left subtree are empty, then LLOW and LHIGH are ϕ. The same is true for the right subtree. In a sense, the machine determines α (what is where in the machine before communication). β is determined as follows. When an L cell wants to send information, it determines the sequence number of the recipient L cell (DEST#) and sends the pair (DEST#, VALUE) to its father T cell. A T cell compares DEST# with the information in its directory to determine whether to send the pair (DEST#, VALUE) further up the tree or down to one of the sons. L cells communicate by sending the pair (DEST#, VALUE). The T cells route the data left, right, or up as soon as the data is received. This algorithm sends an L value only as far up the tree as necessary, thus making most efficient use of the tree branches. The maximum distance a value must travel from one L cell to another is twice the height of the tree.

Constructing the directories is straightforward and can be done in a single upward and downward sweep. In the upward sweep, empty L cells send a "0" to their fathers, occupied L cells send a "1". A T cell receives values from its left and right sons into LNUM and RNUM and sends LNUM+RNUM to its father. LNUM and RNUM contain the number of occupied L cells in the T cell's left and right subtrees. The C cell receives from its son a value n which equals the total number of occupied L cells in the tree as shown in the example of Figure 3.16a.

The C cell starts the downward sweep by returning n and $n-1$ to its son, the root T cell. Each T cell constructs its directory using the algorithm shown in

Figure 3.17. At the end of the downward sweep, every L cell receives n and its sequence number from its father; an empty L cell receives n and ϕ, as shown in the example of Figure 3.16b. The analysis of the initialization sweep for GDCA is shown in Figure 3.18.

The L cells are responsible for computing DEST#. We assume a unique destination for each L cell value. For example, if we want to shift the L cell values a distance k circularly to the left (Figure 3.15a), an L cell must have

n	total number of occupied L cells
$seq\#$	sequence number
k	shift distance.

With these, the L cell computes

DEST# := mod($seq\#-k,n$).

To compute the DEST# needed for an exchange (Figure 3.15b), the L cells would execute

if mod($seq\#$,2)=0 **then** DEST#:=$seq\#+1$ **else** DEST#:=$seq\#-1$;

To implement a "shuffle" pattern (Figure 3.15c), the L cells would execute

if $seq\#<n/2$ **then** DEST#:=$2\times seq\#$ **else** DEST#:=$2\times seq\#-n+1$;

Transposing a square matrix with n elements requires each L cell to send its value to the following destination

$row := \sqrt{n}$;
j := mod($seq\#,row$);
$i := (seq\#-j)/row$;
DEST# := $j\times row + i$;

In these and many other important cases, the functions required to compute DEST# are simple. They use $seq\#$, n, and possibly other information characterizing the data (such as row or column length, if the data represents matrix elements).

The time required to perform GDCA, however, would depend on what the communication pattern is. Some patterns would still require O(n) time. For example, reversal of the L cell elements (for an n L cell tree machine, sending the contents L_i to L_{n-i-1}, i= 0, 1, ..., $n-1$) would require all of the L cell elements to go through the root T cell, i.e., $O(n)$ execution time. The "shuffle" pattern (Figure 3.15c) and transposing a square matrix with n elements also require $O(n)$ time. Some patterns, however, are sublinear, such as the k-shift and exchange patterns shown in Figures 3.15a and 3.15b.

In this dissertation, we shall often make use of GDCA, especially to perform k-shift. The next section presents estimates of the execution time of GDCA when performing a k-shift.

2. Execution Time of K-shift

First, we develop an upper bound of the time required by GDCA to perform a circular k-shift to the *left*. The analysis of a k-shift to the *right* is similar. Let $N = 2^p$ be the number of L cells and n the number of occupied L cells (i.e., the number of L cells participating in the shift). We analyze only values of $k \leq \lceil n/2 \rceil$. (A k-shift to the left, where $k > \lceil n/2 \rceil$, is equivalent to an $(n-k)$-shift to the right, where $n-k \leq \lceil n/2 \rceil$. By symmetry, the analysis of an $(n-k)$-shift to the right is identical to a k-shift to the left.) Define the *level* of a cell to be the length of the shortest path from the cell to any leaf. Thus, the L cells are on level 0, their father T cells are on level 1, ..., and the root T cell is on level p.

The T cell needs a *policy* to handle possible contention among its father and sons. Contention occurs when, for example, a T cell receives a DEST# from each of its sons, and both DEST#'s must be sent to the father; the T cell must decide

which to send first. A T cell policy explicitly states which of its father and sons has highest, middle, and least priority. Consider the two policies, P1 and P2:

P1: FATHER > LEFTSON > RIGHTSON

P2: FATHER > RIGHTSON > LEFTSON .

P1 says that, in the event of contention, the father has priority over the left son which has priority over the right son. P2 says that the father has priority over the right son which has priority over the left son. Assume that the T cells adopt policy P1. The analysis using policy P2 is similar.

We observe what happens in individual subtrees as (DEST#, VALUE) pairs are sent up the tree. Let T be an arbitrary T cell. Let n_L and n_R be the number of occupied L cells in its left and right subtrees, respectively. There are three cases.

(1) $k \geq n_L + n_R$ (Figure 3.19a). Because the shift distance is greater than the number of occupied L cells in both subtrees, all of the $n_L + n_R$ pairs must be sent to the father of the T cell, i.e., shifted out of this subtree. Because T is using policy P1, pairs from the left son are sent before pairs from the right son. There is contention in this T cell (pairs from the right son must wait), but no *unnecessary* delay.

(2) $k \leq n_L$ (Figure 3.19b). Because the shift distance is less than the number of occupied L cells in T's *left* subtree, only the pairs from the leftmost k occupied L cells of T's left subtree must be sent to the father. The destinations of all other occupied L cells (in T's subtrees) are in T's subtrees, and are sent *down* by T as soon as they are received. There is no contention; streams flow through the channels unencumbered. The root T cell is a spe-

cial case: none of the values it receives is sent to its father (C cell). The root T cell receives at most k (DEST#, VALUE) pairs from its left son, all of which must be sent to its right son. It also receives at most k (DEST#, VALUE) pairs from its right son, all of which must be sent to its left son. A maximum of $2k$ numbers (i.e., k pairs) move from one to the other subtree of the root. There is no contention as the two streams are moving in opposite directions. The first DEST# reaches the root after p steps, and reaches the destination L cell after p more steps. The rest of the stream reach the destination L cells after another $2k-1$ steps. The total time for these pairs to rise to the root and reach their destinations is therefore $2p + 2k - 1$ steps.

(3) $n_L < k < n_L + n_R$ (Figure 3.19c). Here, all n_L of the pairs from T's left son must be sent to T's father. The first $k - n_L$ pairs from T's right son must *also* be sent to T's father. The trailing $n_L + n_R - k$ pairs from T's right son must be sent to T's left son; *however, these pairs are unnecessarily delayed*. Because T is using policy P1, the trailing $n_L + n_R - k$ pairs from T's right son are prevented from moving toward their destinations because they must wait for the $k - n_L$ pairs (which must be sent to T's father) to clear. If T is on level h, the total time required for all pairs in T's subtrees to reach their destinations is at least

$$2h + 2(n_L + n_R) - 1 \; < \; 2p + 2(2k) - 1 \, . \qquad (3.46)$$

We observe that on any path from the root to a leaf, there is at most one such T. Thus, this delay is not compounded. (For a full tree, i.e., when all L cells are occupied, all such T cells have the same level number.)

We use the right hand side of (3.46), added to the time required to build the directories ($2p + 3$, as shown in Figure 3.18), to bound the total time, $T(k)$, required for a GDCA k-shift, i.e.,

$$T(k) = (2p + 3) + 2p + 2(2k) - 1 = 4p + 4k + 2 \qquad (3.47)$$

or $O(k)$ time. If $k=0$ (L cell values are not communicated), the only cost would be the time required to initialize the T cell directories:

$$T(0) = T(n) = (2p + 3) \qquad (3.48)$$

Note that the problem described in case (3) would not have occurred had T used policy P2 instead (Figure 3.19d). This is because the pairs (sent by T's right son) that need to be sent to T's left son are received by T first. Hence, they are not delayed.

A program was written to simulate a tree machine executing the GDCA k-shift pattern. The program was written in PL/C and run on an IBM 360/75. The object was to count the number of communication time units needed to execute k-shift. The assumptions of the program were as follows.

(1) A cell could communicate with its father or sons in one time unit, where communication means sending or receiving one numeric value. A T cell could therefore send as many as three (possibly different) values and receive three other values in one time unit. An L cell would need at least two time units to send a (DEST#, VALUE) pair to its father.

(2) If a cell receives a value in one time unit, the cell has to wait at least until the next time unit before sending that value.

(3) All cells operated synchronously.

(4) The direction of the shift was to the left. A k-shift would cause L_i to receive the value of $L_{(i+k) \bmod n}$, where n is the number of occupied L cells.

The program had four parameters:

(1) the number of L cells in the tree machine,

(2) the number of occupied L cells,

(3) the shift distance k, and

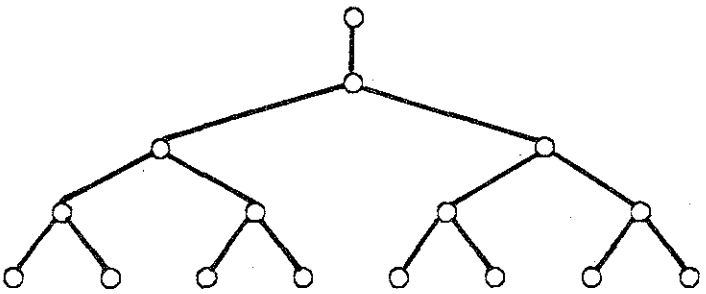
(4) the T cell policy (both P1 and P2 were tested).

For a given simulation run, if the number of occupied L cells was less than the total number of L cells, the empty cells were selected at random, using the random number generator described by Coveyou and Macpherson [CoMa67].

Figures 3.20-22 summarize the results obtained for a 16-, 32-, and 64-L-cell tree machine where n is the number of occupied L cells and k is the shift distance. Entries under P1 and P2 are the number of steps produced by the simulation. $T(k)$ is the bound defined by equation (3.47).

All results produced by the simulation were bounded by $T(k)$, using either T cell policy P1 or P2. It is interesting to note, however, that for values of $k \leq \lceil n/2 \rceil$ and using policy P1, the time required for GDCA did not increase uniformly with k (as they did with policy P2). Increases were in large amounts, followed by plateaus (see, for example, Figure 3.22, $N = n = 64$, $k = 17, \ldots, 31$). The values of k which cause these sudden rises correspond with those identified as potential problems in case (3). Note also that for $k > \lceil n/2 \rceil$ a similar problem occurs when policy P2 is used. In summary, the results of the simulation (Fig-

ures 3.20-22) show that best results are obtained if the T cells use policy P2 when $k \leq \lceil n/2 \rceil$ and policy P1 otherwise.



Initial:	$X_{7,6}$	$X_{6,5}$	$X_{5,4}$	$X_{4,3}$	$X_{3,2}$	$X_{2,1}$	$X_{1,0}$	$X_{0,-1}$
Final:	x_7	x_6	x_5	x_4	x_3	x_2	x_1	x_0
	x_6	x_5	x_4	x_3	x_2	x_1	x_0	x_{-1}

Figure 3.1 Initial and final contents of $n=8$ L cells executing LR1. The ith L cell, counting from the right, initially contains the pair of values $(X_{i,i-1})=(a_i,b_i)$, $0 \le i \le n-1$. At the end of the downward sweep, the ith L cell contains the solution components x_i and x_{i-1}.

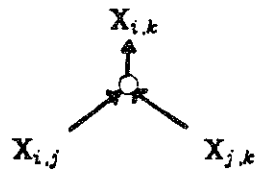


Figure 3.2 A composition step in LR1. During the upward sweep, a T cell receives a_L and b_L from its left son and a_R and b_R from its right son. The T cell stores the values a_L and b_L and sends the values $a_L+b_L a_R$ and $b_L b_R$ to its father, as described in equation (3.22).

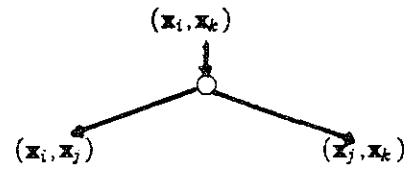


Figure 3.3 A substitution step in LR1. During the downward sweep, a T cell receives (x_i,x_k) from its father. It computes $x_j=a_R + b_R x_k$ and sends (x_i,x_j) to its left son and (x_j,x_k) to its right son. Note that a_R and b_R were stored during the upward sweep.

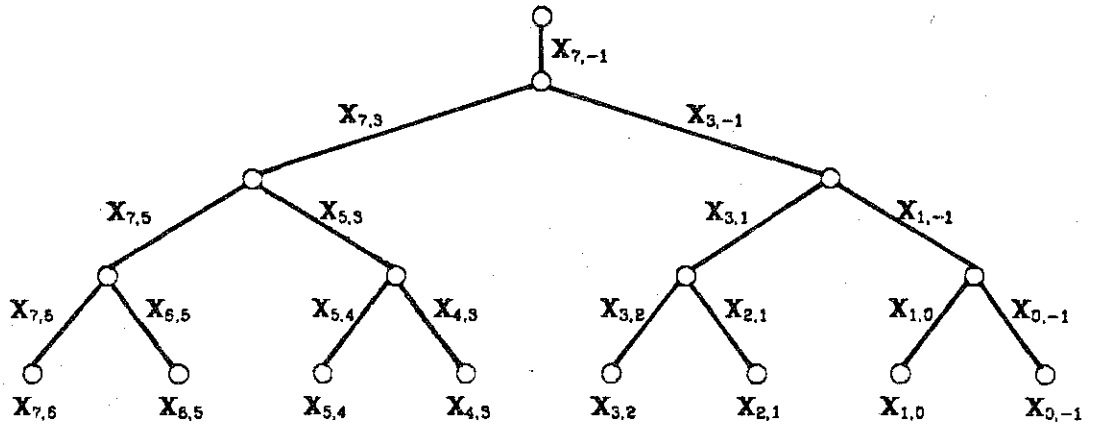


Figure 3.4 Values communicated during the upward (composition) sweep of LR1 on a fully occupied 8 L cell tree machine. Composition of two linear equations occurs at each of the T cells.

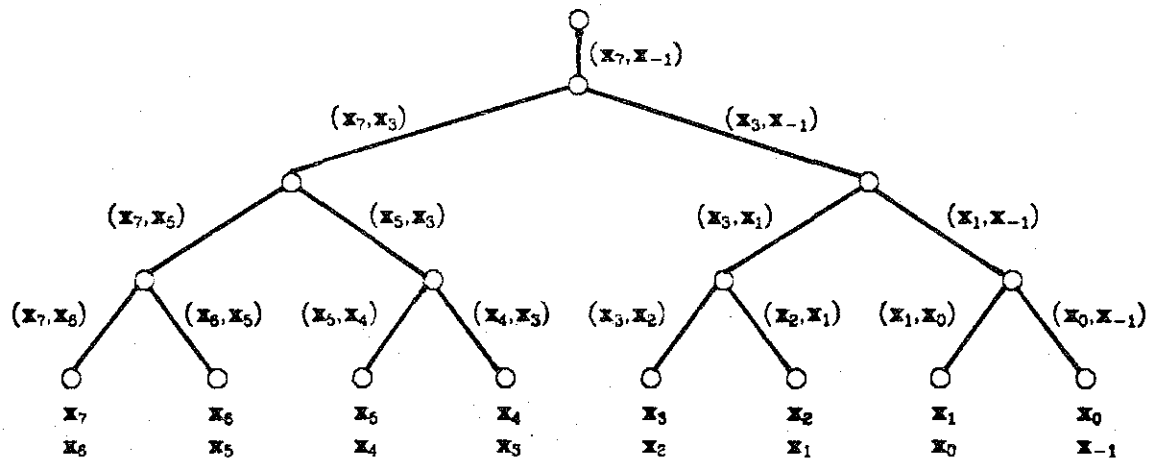
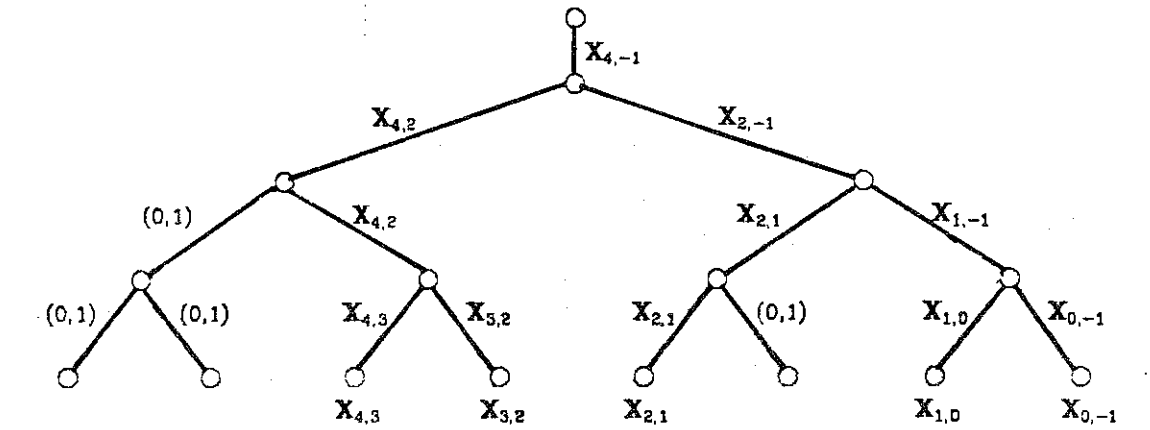


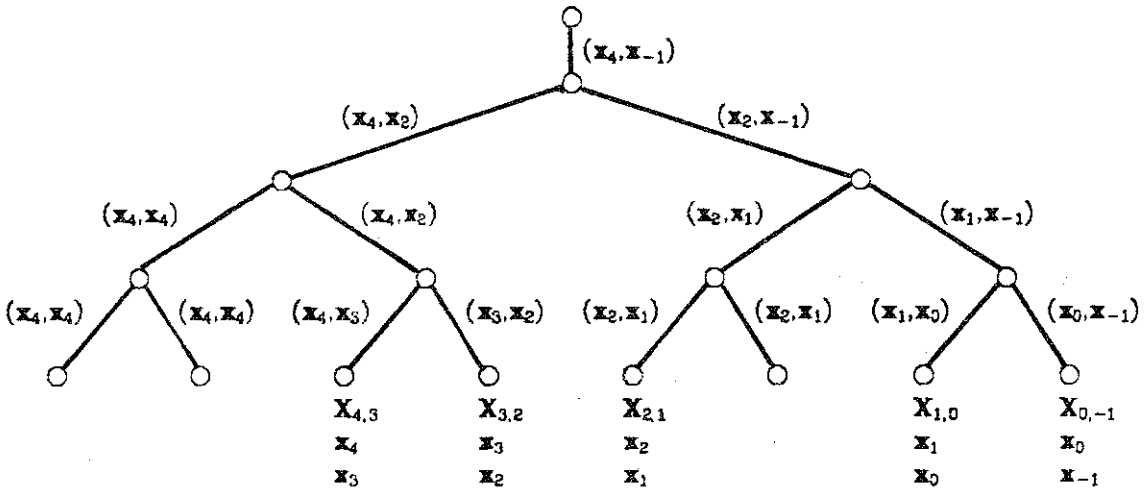
Figure 3.5 Values communicated during the downward (substitution) sweep of LR1. Evaluation of a new x value occurs at each of the T cells. Note that the ith L cell, counting from the right, receives the solutions x_i and x_{i-1}.

LR1							
Sweep	Comm. Steps	Parallel Operations					
		L cells		T cells		C cell	
		+	×	+	×	+	×
up	$2\log n + 2$	0	0	$\log n$	$2\log n$	0	0
down	$2\log n + 2$	0	0	$\log n$	$\log n$	0	0
Total	$4\log n + 4$	0	0	$2\log n$	$3\log n$	0	0

Figure 3.6 Analysis of LR1. During the upward sweep, each level of cells sends a pair of values to the next higher level. Because data from the L cells must pass through $\log n + 1$ levels to reach the C cell, the upward sweep requires $2\log n + 2$ communication steps. Each T cell performs two multiplications and one addition. Because there are $\log n$ levels of T cells, the upward sweep executes $2\log n$ parallel multiplications and $\log n$ parallel additions. No arithmetic operations are executed by the C cell or the L cells. The downward sweep is analyzed similarly.



(a)



(b)

Figure 3.7 Upward and downward sweeps of LR1 for n=5 on an 8 L cell tree machine. (a) Empty L cells send up the pair (0,1) while occupied L cells send up the coefficient pair (a, b). Neither the T cells nor the C cell knows whether the data it receives came from an empty or an occupied L cell. (b) At the end of the downward sweep, an empty L cell ignores the values it receives.

FRACTION							
Sweep	Comm. Time	Parallel Operations					
		L cells		T cells		C cell	
		+	×	+	×	+	×
up	$4\log n + 4$	0	0	$4\log n$	$8\log n$	0	1
down	$2\log n + 2$	0	0	$2\log n$	$3\log n$	0	0
Total	$6\log n + 6$	0	0	$6\log n$	$11\log n$	0	1

Figure 3.8 Analysis of FRACTION. During the upward sweep, each cell must send 4 values a, b, c, and d to its father, requiring a total of $4(\log n + 1)$ communication steps. Each of the T cells must substitute the equation received from the left son into the equation received from the right, requiring 8 multiplications and 4 additions, as shown in equation (3.13). A total of $8\log n$ parallel multiplications and $4\log n$ parallel additions are executed. The C cell must execute a single division. During the downward sweep, every cell sends 2 values to each of its sons, requiring a total of $2(\log n + 1)$ communication steps. Each T cell must solve a quotient of two linear equations, requiring 3 multiplications or divisions and 2 additions.

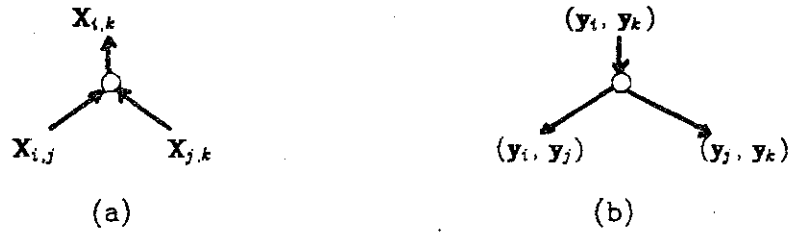


Figure 3.9 Composition and substitution applied to LR2. (a) During the upward (composition) sweep, a T cell receives $\mathbf{X}_{i,j} = (A, B)$ from its left son and $\mathbf{X}_{j,k} = (A', B')$ from its right son. The T cell applies the function g (equation (3.28i)) and sends the result $\mathbf{X}_{i,k} = (A + BB', BB')$ to its father. (b) During the downward (substitution) sweep, the T cell receives the pair of solutions $(\mathbf{y}_i, \mathbf{y}_k)$ from its father. It uses $\mathbf{y}_k$ to compute $\mathbf{y}_j$ using equation (3.34) sends the pair $(\mathbf{y}_i, \mathbf{y}_j)$ to its left son and the pair $(\mathbf{y}_j, \mathbf{y}_k)$ to its right son.

LR2							
Sweep	Comm. Time	Parallel Operations					
		L cells		T cells		C cell	
		+	×	+	×	+	×
up	$6\log n + 6$	0	0	$8\log n$	$12\log n$	0	0
down	$4\log n + 4$	0	0	$4\log n$	$4\log n$	0	0
Total	$10\log n + 10$	0	0	$12\log n$	$16\log n$	0	0

Figure 3.10 Analysis of LR2. During the upward sweep, a T cell receives 6 values from each son and sends 6 values to its father. The total communication time is therefore $6(\log n + 6)$ units. During the downward sweep, each T cell receives 4 values from its father and sends 4 values to each son. Only the T cells perform arithmetic operations. Each T cell evaluates (3.19) and (3.20) during the upward sweep, requiring 12 multiplications and 6 additions. It evaluates (3.21) and (3.22) during the downward sweep, requiring 4 multiplications and 4 additions.

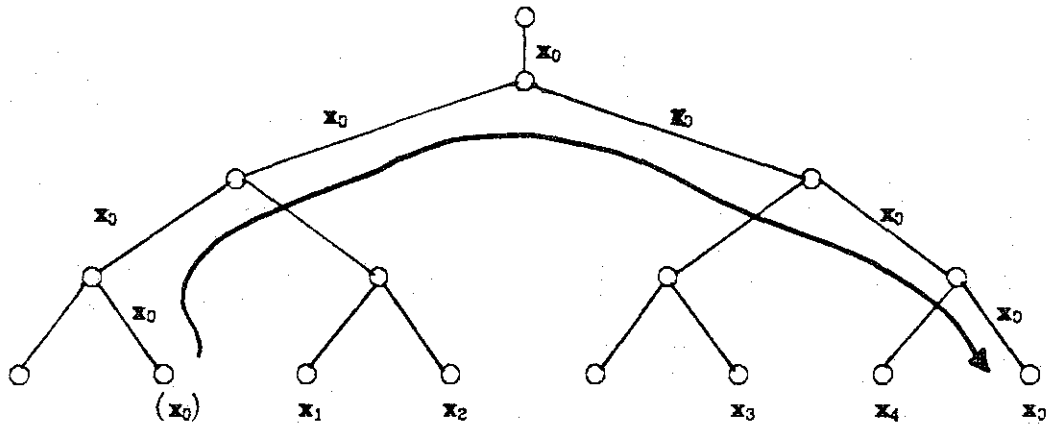


Figure 3.11 One implementation of ROTL. x_0 is sent up to the C cell and sent down to an empty L cell to the right of x_{n-1}.

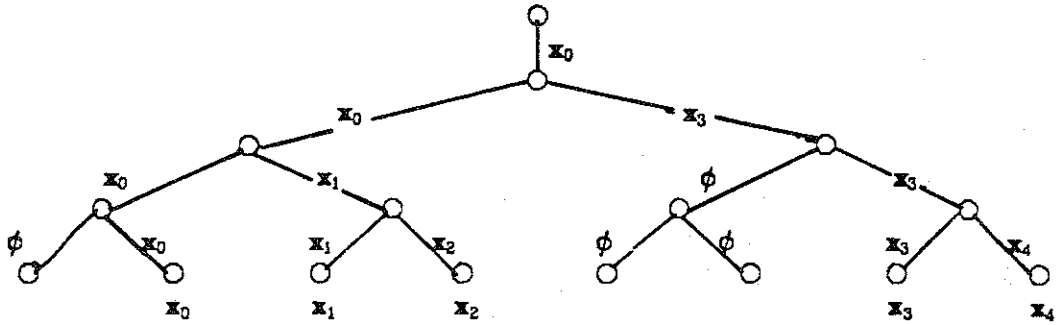


Figure 3.12 ROTLA upward sweep. The L cells initiate the upward sweep by sending the x values to their fathers. A T cell receives the values x_L and x_R from its left and right sons, saves x_R, and sends x_L to its father. The upward sweep ends when the C cell receives a value from the root T cell.

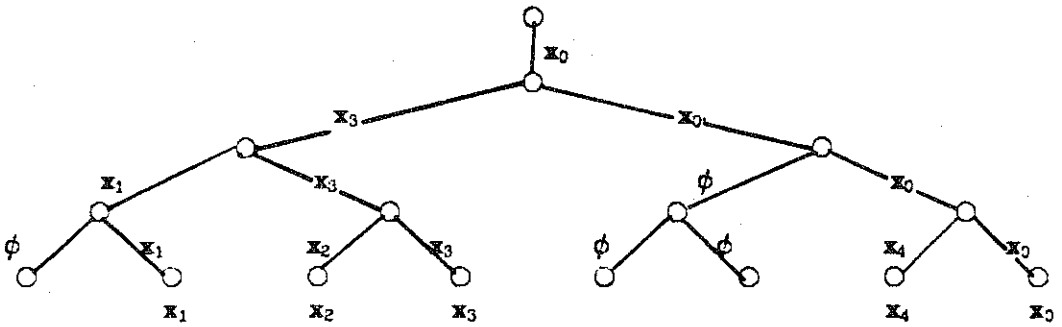


Figure 3.13 ROTLA downward sweep. The downward sweep starts when the C cell returns the value to its son. Each T cell receives a value from the father, sends it to the right son, and sends the value saved from the upward sweep (x_R) to the left son. The downward sweep ends when the L cells receive the new values.

ROTLA								
Sweep	Comm. Steps	Parallel Operations						
		L cells		T cells		C cell		
		+	×	+	×	+	×	
up	$\log n + 1$	0	0	0	0	0	0	
down	$\log n + 1$	0	0	0	0	0	0	
Total	$2\log n + 2$	0	0	0	0	0	0	

Figure 3.14 Analysis of ROTLA. Each cell sends one value to its father during the upward sweep. As there are $\log(n)+2$ levels of cells in the tree, the upward sweep requires $\log(n)+1$ parallel communication steps. Similarly, the downward sweep requires $\log(n)+1$ parallel communication steps. None of the cells execute any arithmetic operations.

L cells:	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
send:		x0		x1		x2	x3	x4		x5	x6		x7		x8	x9
receive:																
(a)		x3		x4		x5	x6	x7		x8	x9		x0		x1	x2
(b)		x1		x0		x3	x2	x5		x4	x7		x6		x9	x8
(c)		x0		x5		x1	x6	x2		x7	x3		x8		x4	x9

Figure 3.15 Examples of data communication among the L cells for n=10, #L cells=16. (a) L cell elements are shifted circularly to the left a distance k=3. (b) L cell with sequence numbers i and i+1 exchange values, i=even. (c) L cell elements are "shuffled".

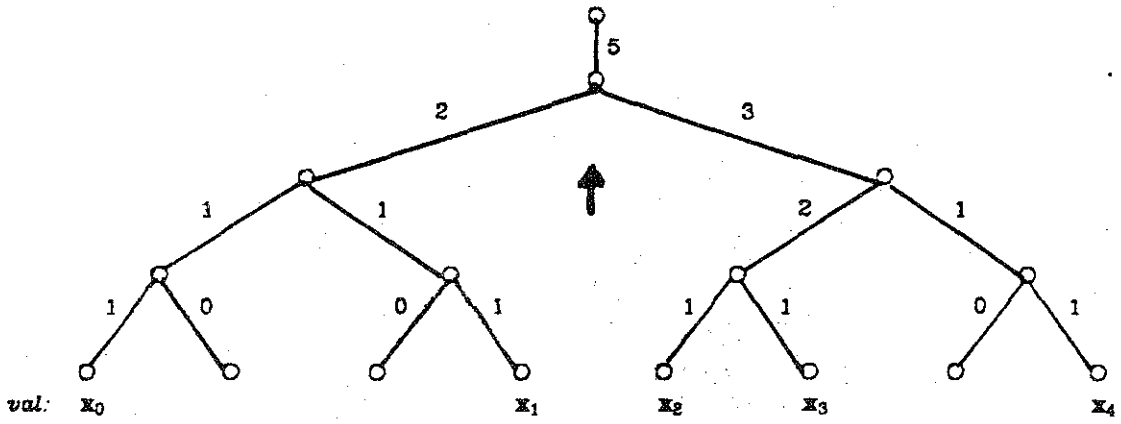


Figure 3.16a GDCA initialization upward sweep. Empty L cells send a "0", nonempty L cells send a "1". A T cell stores the value received from its left son (right son) in LNUM (RNUM). The value equals the number of nonempty L cells in the T cell's left (right) subtree. The C cell receives n, the total number of nonempty L cells in the tree.

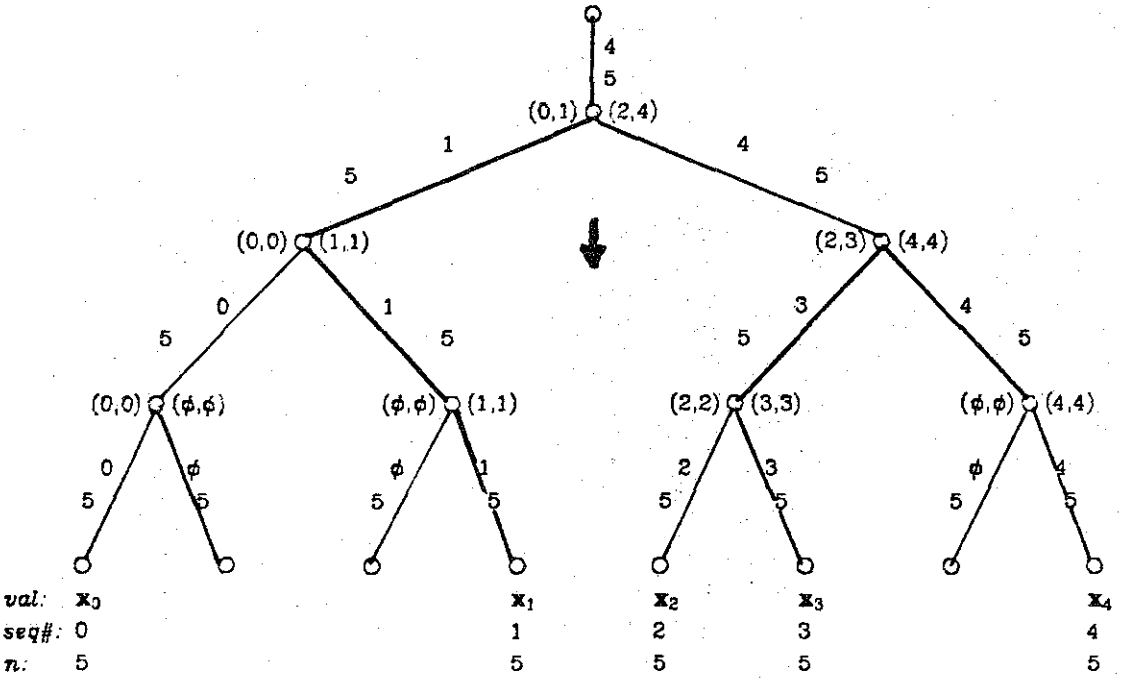


Figure 3.16b GDCA initialization downward sweep. Each T cell receives a value from its father and uses the value to determine (LLOW, LHIGH) and (RLOW, RHIGH). Note that the first value an L cell receives is n, the number of nonempty L cells. The second value is either ϕ or its sequence number.

```
F.receive(n,FVAL);                  receive two values from father
if RNUM≠0                           determine highest and lowest
  then begin                          sequence numbers in right
    RHIGH := FVAL;                    subtree
    RLOW := FVAL−RNUM+1
    end
  else RHIGH := RLOW := ϕ;

if LNUM≠0                           determine highest and lowest
  then begin                          sequence numbers in left
    LHIGH := FVAL−RNUM;               subtree
    LLOW := LHIGH−LNUM+1
    end
  else LHIGH := LLOW := ϕ;

L.send(n,LHIGH),                    send data to left and right sons
R.send(n,RHIGH);
```

Figure 3.17 Algorithm constructing the T cell directories for GDCA.

Constructing GDCA Directories							
Sweep	Comm. Steps	Parallel Operations					
		L cells		T cells		C cell	
		+	×	+	×	+	×
up	$\log(n)+1$	0	0	$\log(n)$	0	0	0
down	$\log(n)+2$	0	0	$5\log(n)$	0	0	0
Total	$2\log(n)+3$	0	0	$6\log(n)$	0	0	0

Figure 3.18 Analysis of GDCA directory construction. During the upward sweep, each L and T cell sends one value to its father, thus requiring a total of $\log(n)$ communication time units. During the downward sweep, the C and T cells send two values to each son. Because the T cells may pipeline communication (a T cell may send down the first value received before waiting for the second to arrive), the downward sweep requires a total of $\log(n)-2$ time units ($\log(n)+1$ plus one time unit to send the second value to the L cells). Without pipelining, the time required would be $2\log(n)+2$. Only the T cells perform arithmetic operations. Each T cell performs one addition during the upward sweep and a maximum of two additions and three multiplications during the downward sweep.

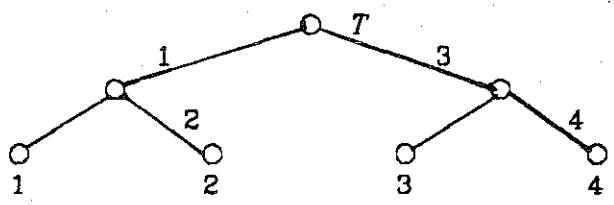


Figure 3.19a GDCA k-shift, case (1): $n_L = n_R = 2$, $k \geq 4$, where n_L and n_R are the number of occupied L cells in T's left and right subtrees, respectively, and k is the shift distance. Because the shift distance is greater than or equal to the sum of $n_L + n_R$, values 1, 2, 3, and 4 are all sent to the father of T. Because T has adopted policy P1, there is contention (values 3 and 4 must wait for 1 and 2 to be sent) but no unnecessary delay.

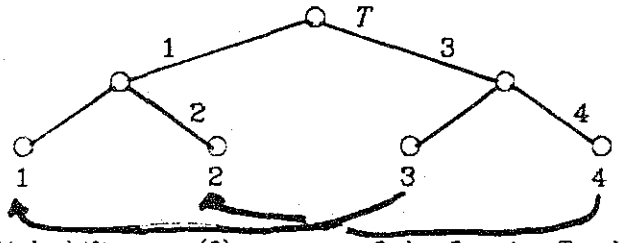


Figure 3.19b GDCA k-shift, case (2): $n_L = n_R = 2$, $k = 2$, using T cell policy P1. The arrows indicate the destinations of values 3 and 4. Because the shift distance is equal to n_L, the values 1 and 2 are sent by T to its father while 3 and 4 are sent to its left son. Because T has adopted policy P1, there is no contention and no delay.

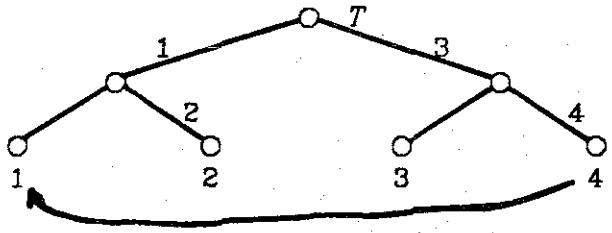


Figure 3.19c GDCA k-shift, case (3): $n_L = n_R = 2$, $k = 3$, using T cell policy P1. The arrow indicates the destination of value 4. Because $n_L < k < n_L + n_R$, the values 1, 2, and 3 are sent by T to its father. Value 4 should be sent to T's left son but must wait for value 3 to be sent up. Value 3, in turn, must wait for values 1 and 2 to be sent up. Value 4 is, therefore, unnecessarily delayed.

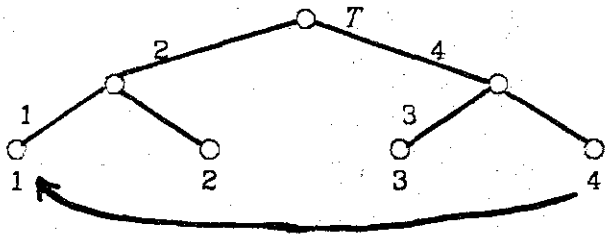


Figure 3.19d GDCA k-shift, case (3): $n_L = n_R = 2$, $k = 3$, using T cell policy P2. The arrow indicates the destination of value 4. Unlike in Figure 3.19c, value 4 is received by T ahead of value 3. Thus, 4 may be sent down to T's left son in the next time step while value 2 is sent to the father. In the same step, values 1 and 3 are received by T. In the next two time steps, values 3 and 1 (in that order) are sent by T to its father. Value 4 was not delayed.

Simulation Results (16 L cells)									
n	k	P1	P2	T(k)	n	k	P1	P2	T(k)
4	1	20	20	22	4	3	18	16	24
	2	22	22	26		4	11	11	11
5	1	20	20	22	5	4	20	20	22
	2	22	22	26		5	11	11	11
	3	22	22	30					
8	1	20	20	22	8	5	24	24	30
	2	22	22	26		6	22	22	26
	3	24	24	30		7	20	20	22
	4	24	26	34		8	11	11	11
9	1	20	20	22	9	6	24	24	30
	2	22	22	26		7	22	22	26
	3	26	24	30		8	20	20	22
	4	26	26	34		9	11	11	11
	5	26	26	38					
11	1	20	20	22	11	7	26	28	34
	2	22	22	26		8	24	24	30
	3	24	24	30		9	22	22	26
	4	30	26	34		10	20	20	22
	5	28	28	38		11	11	11	11
	6	26	28	42					
12	1	20	20	22	12	7	28	26	38
	2	22	22	26		8	26	30	34
	3	26	24	30		9	24	26	30
	4	28	26	34		10	22	22	26
	5	30	28	38		11	20	20	22
	6	30	30	42		12	11	11	11
14	1	20	20	22	14	8	30	32	42
	2	22	22	26		9	28	32	38
	3	24	24	30		10	26	28	34
	4	26	26	34		11	24	24	30
	5	30	28	38		12	22	22	26
	6	32	30	42		13	20	20	22
	7	32	32	46		14	11	11	11
16	1	20	20	22	16	9	32	32	46
	2	22	22	26		10	30	32	42
	3	24	24	30		11	28	32	38
	4	26	26	34		12	26	26	34
	5	32	28	38		13	24	24	30
	6	32	30	42		14	22	22	26
	7	32	32	46		15	20	20	22
	8	34	34	50		16	11	11	11

Figure 3.20

Simulation Results (32 L cells)									
n	k	P1	P2	T(k)	n	k	P1	P2	T(k)
4	1	24	22	26	4	3	24	24	26
	2	26	24	30		4	13	13	13
8	1	24	24	26	8	5	28	26	34
	2	26	26	30		6	26	26	30
	3	28	28	34		7	24	24	26
	4	30	28	38		8	13	13	13
12	1	24	24	26	12	7	32	30	42
	2	26	26	30		8	30	30	38
	3	30	28	34		9	28	28	34
	4	32	30	38		10	26	26	30
	5	34	32	42		11	24	24	26
	6	34	34	46		12	13	13	13
16	1	24	24	26	16	9	36	38	50
	2	26	26	30		10	34	30	46
	3	28	28	34		11	32	32	42
	4	36	30	38		12	30	34	38
	5	38	32	42		13	28	30	34
	6	38	34	46		14	26	26	30
	7	34	36	50		15	24	24	26
	8	34	36	54		16	13	13	13
23	1	24	24	26	32	13	42	44	62
	2	26	26	30		14	40	48	58
	3	28	28	34		15	38	44	54
	4	30	30	38		16	36	40	50
	5	32	32	42		17	34	42	48
	6	40	34	46		18	32	32	42
	7	40	36	50		19	30	30	38
	8	44	36	54		20	28	28	34
	9	46	40	58		21	26	26	30
	10	38	40	62		22	24	24	26
	11	44	42	66		23	13	13	13
	12	42	44	70					
32	1	24	24	26	32	17	52	52	82
	2	26	26	30		18	50	52	78
	3	28	28	34		19	48	52	74
	4	30	30	38		20	46	52	70
	5	34	32	42		21	44	52	66
	6	34	34	46		22	42	52	62
	7	36	36	50		23	40	52	58
	8	38	38	54		24	38	38	54
	9	52	40	58		25	36	36	50
	10	52	42	62		26	34	34	46
	11	52	44	66		27	32	34	42
	12	52	46	70		28	30	30	38
	13	52	48	74		29	28	28	34
	14	52	50	78		30	26	26	30
	15	52	52	82		31	24	24	26
	16	54	54	86		32	13	13	13

Figure 3.21

Simulation Results (64 L cells)									
n	k	P1	P2	T(k)	n	k	P1	P2	T(k)
16	1	28	28	30	16	9	38	42	54
	2	30	30	34		10	38	44	50
	3	32	32	38		11	36	36	46
	4	36	34	42		12	34	36	42
	5	38	36	46		13	32	32	38
	6	42	38	50		14	30	30	34
	7	44	40	54		15	28	28	30
	8	38	40	58		16	15	15	15
32	1	28	28	30	32	17	48	58	86
	2	30	30	34		18	54	60	82
	3	32	32	38		19	52	56	78
	4	34	34	42		20	50	58	74
	5	36	36	46		21	48	56	70
	6	42	38	50		22	46	52	66
	7	50	40	54		23	44	54	62
	8	52	42	58		24	42	54	58
	9	52	44	62		25	40	40	54
	10	52	46	66		26	38	40	50
	11	48	48	70		27	36	36	46
	12	64	50	74		28	34	34	42
	13	60	50	78		29	32	32	38
	14	58	52	82		30	30	30	34
	15	58	56	86		31	28	28	30
	16	56	56	90		32	15	15	15
48	1	28	28	30	48	25	72	74	118
	2	30	30	34		26	68	72	114
	3	32	32	38		27	68	72	110
	4	34	34	42		28	66	72	106
	5	36	36	46		29	64	72	102
	6	44	38	50		30	62	74	98
	7	46	40	54		31	60	74	94
	8	46	42	58		32	58	74	90
	9	48	44	62		33	56	74	86
	10	64	46	66		34	54	64	82
	11	72	48	70		35	52	68	78
	12	70	50	74		36	50	50	74
	13	70	50	78		37	48	64	70
	14	72	52	82		38	46	50	66
	15	80	56	86		39	44	68	62
	16	76	56	90		40	42	50	58
	17	76	60	94		41	40	46	54
	18	74	62	98		42	38	46	50
	19	82	64	102		43	36	36	46
	20	78	66	106		44	34	34	42
	21	76	68	110		45	32	32	38
	22	74	70	115		46	30	30	34
	23	76	72	118		47	28	28	30
	24	72	72	122		48	15	15	15

Figure 3.22 (continued next page)

Simulation Results (64 L cells)									
n	k	P1	P2	T(k)	n	k	P1	P2	T(k)
64	1	28	28	30	64	33	88	88	150
	2	30	30	34		34	86	88	146
	3	32	32	38		35	84	88	142
	4	34	34	42		36	82	88	138
	5	36	36	46		37	80	88	134
	6	38	38	50		38	78	88	130
	7	40	40	54		39	76	88	126
	8	54	42	58		40	74	88	122
	9	54	44	62		41	72	88	118
	10	54	46	66		42	70	88	114
	11	54	48	70		43	68	88	110
	12	54	50	74		44	66	88	106
	13	54	52	78		45	64	88	102
	14	54	54	82		46	62	88	98
	15	56	56	86		47	60	88	94
	16	58	58	90		48	58	88	90
	17	88	60	94		49	56	56	86
	18	88	62	98		50	54	54	82
	19	88	64	102		51	52	54	78
	20	88	66	106		52	50	54	74
	21	88	68	110		53	48	54	70
	22	88	70	114		54	46	54	66
	23	88	72	118		55	44	54	62
	24	88	74	122		56	42	42	58
	25	88	76	126		57	40	40	54
	26	88	78	130		58	38	38	50
	27	88	80	134		59	36	36	46
	28	88	82	138		60	34	34	42
	29	88	84	142		61	32	32	38
	30	88	86	146		62	30	30	34
	31	88	88	150		63	28	28	30
	32	90	90	154		64	15	15	15

Figure 3.22

CHAPTER 4. TRIDIAGONAL LINEAR SYSTEM SOLVERS

A. Overview

On a sequential computer, an $(n \times n)$ tridiagonal linear system of equations can be solved in $O(n)$ time. The traditional algorithms for solving such a system are LU decomposition and Gaussian elimination, or the more efficient Thomas algorithm [Ames77, Youn71]. During the last two decades, parallel algorithms to solve tridiagonal systems have been developed for computers such as the ILLIAC-IV, CDC STAR-100, and TI-ASC. In 1965, R. Hockney [Hock65] introduced cyclic reduction, a method for solving the block-tridiagonal linear system resulting from the five-point approximation of Poisson's equation on a square region (Chapter 2). Buneman proposed a slightly different version of cyclic reduction [Bune69, BuGN70] that was proven to be stable although requiring more arithmetic operations. Both are direct methods and can be implemented on a parallel processor in $O(\log\ n)$ steps, providing a great improvement over previous methods, both direct and iterative [Dorr70]. Moreover, both methods can be applied to tridiagonal linear systems as well [Hock70].

Stone [Ston73a] introduced a technique called recursive doubling which can solve linear recurrences on a parallel processor of the ILLIAC-IV type in $O(\log\ n)$ steps. Since the LU decomposition of a tridiagonal system can be transformed into a problem of solving linear recurrences, recursive doubling provided a third parallel tridiagonal system solver. Stone compared these three methods

[Ston75] and concluded that, for an ILLIAC-IV type parallel processor, cyclic reduction is preferable for the constant diagonal case whereas recursive doubling is more efficient when the diagonal has arbitrary values. His analysis did not include overhead cost such as data rearrangement. Lambiotte and Voigt [LaVo75] included overhead cost and machine timing formulas in their analyses of three direct methods (LU decomposition, recursive doubling, and cyclic reduction) and three iterative methods implemented on a vector processor, the CDC STAR-100. Among their conclusions: direct methods are superior to iterative methods, and for large systems cyclic reduction is the fastest. They also concluded that when implemented on a vector processor, recursive doubling required $O(n \log n)$ time, compared to the $O(n)$ time required by traditional algorithms on a sequential computer. This is because recursive doubling requires a total of $O(n \log n)$ operations and a vector processor gives only a constant speedup over sequential computers. Furthermore, Diamond [Diam75] has found recursive doubling to be unstable for certain systems of equations.

Other methods subsequently developed include a method by Sameh and Kuck [SaKu78] based on Givens' reduction of a matrix to triangular form, and a method by Swarztrauber [Swar79] based on Cramer's rule. Both methods run in $O(\log n)$ time on an n-processor machine. As noted by Ortega and Voigt in an excellent survey [OrVo77], these methods may have merit if the stability of cyclic reduction is in doubt.

Traditional *iterative* methods for solving tridiagonal systems include the Jacobi method, Jacobi over-relaxation (JOR), the Gauss-Seidel method, and successive over-relaxation (SOR). Lambiotte and Voigt [LaVo75] found that whereas the Jacobi method and JOR could be efficiently implemented on a vector

processor, neither the Gauss-Seidel method nor SOR could because of the sequential nature of the recurrence expressions produced. However, a variation of SOR called red-black SOR [Youn71], which treats the vector of elements as two independent halves, could be efficiently implemented. Traub [Trau73] introduced a method that transforms the recurrence equations of LU decomposition into iterations and showed that the rate of convergence is independent of the size of the matrix. Lambiotte and Voigt [LaVo75] implemented Traub's algorithm on the STAR-100 and an accelerated version of the algorithm is described by Heller, Stevenson and Traub [HeST76].

This chapter addresses the problem of implementing tridiagonal linear system solvers on tree machines. As some of the algorithms presented can be decomposed into problems of solving first- and second-order recurrence expressions, we will make frequent use of the tree algorithms LR1, LR2 and FRACTION, as well as the tree communication algorithms ROTLA and ROTRA, all described in Chapter 3.

Our aim is to solve the $(n \times n)$ tridiagonal linear system

$$\begin{bmatrix} b_0 & c_0 & & & 0 \\ a_1 & b_1 & c_1 & & \\ & a_2 & b_2 & c_2 & \\ & & \ddots & & c_{n-2} \\ 0 & & & a_{n-1} & b_{n-1} \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ \cdots \\ x_{n-1} \end{bmatrix} = \begin{bmatrix} r_0 \\ r_1 \\ r_2 \\ \cdots \\ r_{n-1} \end{bmatrix}. \qquad (4.1)$$

Section B presents three classes of direct solution methods. The first transforms the coefficient matrix into an upper bidiagonal matrix and solves the new system using back substitution. This class includes Gaussian elimination and the Thomas algorithm, algorithms that appear to be inherently sequential and, thus, have been thought to be unsuitable for other parallel processors

[Ston75, LaVo75, HeST76, JeHo79, OrVo77]. Both methods run in $O(n)$ time on a sequential computer and (as we show later) in $O(\log n)$ time on a tree machine. The second class of solutions uses LU decomposition, i.e., the coefficient matrix is decomposed into LU where L is lower bidiagonal and U is upper bidiagonal. The system is then solved using forward substitution and back substitution. Two examples are given, both of which take $O(n)$ time on a sequential computer and $O(\log n)$ time on a tree machine. The third class of algorithms is based on cyclic reduction, in one step transforming a tridiagonal system into one approximately half its size. Iterating $O(\log n)$ times, we are left with one equation in one unknown. We solve this equation and, in $O(\log n)$ more iterations, solve for the rest of the variables. This class includes cyclic reduction and Buneman's algorithm and requires $O(n)$ time on a sequential computer and $O((\log n)^2)$ time on a tree machine.

In section C, we investigate iterative solution methods, including the Jacobi method, JOR, the Gauss-Seidel method, SOR, red-black SOR, and an iterative analog to LU decomposition developed by Traub [Trau73]. One iteration of each of these methods requires $O(n)$ time on a sequential computer and $O(\log n)$ time on a tree machine.

In section D, we summarize our results and make comparisons with results obtained by Stone [Ston75] on a parallel processor and by Lambiotte and Voigt [LaVo75] on a vector computer.

B. Direct Methods

1. Traditionally Sequential Algorithms

In this section, we describe two algorithms, Gaussian elimination and the Thomas algorithm, originally designed for sequential processing. These algorithms have been considered unsuitable for efficient implementation on parallel processors such as vector computers [LaVo75] and array processors of the ILLIAC-IV type [Ston75]. We will see that they can be implemented efficiently on a tree machine.

Gaussian elimination and the Thomas algorithm transform (4.1) into an upper bidiagonal system and then determine the **x**-vector using back substitution. Both methods use first-order recurrence expressions to obtain the bidiagonal system and to perform back substitution.

Thomas Algorithm

The Thomas Algorithm transforms (4.1) into

$$\begin{bmatrix} 1 & e_0 & & & 0 \\ & 1 & e_1 & & \\ & & \ddots & \ddots & \\ & & & 1 & e_{n-2} \\ 0 & & & & 1 \end{bmatrix} \begin{bmatrix} \mathbf{x}_0 \\ \mathbf{x}_1 \\ \cdots \\ \mathbf{x}_{n-2} \\ \mathbf{x}_{n-1} \end{bmatrix} = \begin{bmatrix} \mathbf{y}_0 \\ \mathbf{y}_1 \\ \cdots \\ \mathbf{y}_{n-2} \\ \mathbf{y}_{n-1} \end{bmatrix} \tag{4.2}$$

where

$$
\begin{aligned}
e_0 &= c_0 / b_0 \\
e_i &= c_i / (b_i - a_i e_{i-1}) \\
&= \frac{c_i + 0\, e_{i-1}}{b_i - a_i e_{i-1}} \qquad i = 1,2,\ldots,n-1
\end{aligned}
\tag{4.3}
$$

and

$$
\begin{aligned}
\mathbf{y}_0 &= \mathbf{r}_0 / b_0 \\
\mathbf{y}_i &= \frac{\mathbf{r}_i - a_i \mathbf{y}_{i-1}}{b_i - a_i e_{i-1}} \qquad i = 1,2,\ldots n-1
\end{aligned}
\tag{4.4}
$$

Back substitution provides the desired vector $\mathbf{x}$:

$$
\begin{aligned}
\mathbf{x}_{n-1} &= \mathbf{y}_{n-1} \\
\mathbf{x}_i &= \mathbf{y}_i - e_i \mathbf{x}_{i+1} \qquad i = n-2,\ n-3,\ldots,0
\end{aligned}
\tag{4.5}
$$

To implement this algorithm on the tree machine, we store all information pertinent to the ith equation in the ith L cell (L_i). Each L cell, therefore, contains

a, b, c, r	coefficients of one equation
e, y	storage for intermediate values
x	solution of one equation
$seq\#$	L cell's sequence number, $0 \le seq\# \le n-1$
t_i	temporary

where, by definition, $a_0 = c_{n-1} = 0$. The tree algorithm proceeds as follows.

(1) Compute e_i according to (4.3) using FRACTION. L_i sends up the quadruple $(c_i, 0, b_i, -a_i)$ and receives (e_i, e_{i-1}).

(2) Compute $\mathbf{y}_i$ according to (4.4) using LR1. Note that all of the components of the denominator of (4.4) are known. The denominator may, therefore, be evaluated by the L cells beforehand: L_i computes $t_i := b_i - a_i e_{i-1}$. This reduces (4.4) to a first-order linear recurrence in $\mathbf{y}$, which is solved using LR1 with L_i sending up the pair $(\mathbf{r}_i / t_i, -a_i / t_i)$, and receiving $\mathbf{y}_{i-1}$ and $\mathbf{y}_i$.

(3) Compute $\mathbf{x}_i$ according to (4.5) using BLR1. L_i sends up the pair $(\mathbf{y}_i, -e_i)$ and receives $\mathbf{x}_{i-1}$ and $\mathbf{x}_i$.

The analysis of the Thomas algorithm is summarized in Figure 4.1. This algorithm requires $O(\log n)$ time to execute.

Gaussian Elimination

In Gaussian elimination, (4.1) is transformed into

$$\begin{bmatrix} d_0 & c_0 & & & 0 \\ & d_1 & c_1 & & \\ & & \ddots & & \\ & & & d_{n-2} & c_{n-2} \\ 0 & & & & d_{n-1} \end{bmatrix} \begin{bmatrix} \mathbf{x}_0 \\ \mathbf{x}_1 \\ \cdots \\ \mathbf{x}_{n-2} \\ \mathbf{x}_{n-1} \end{bmatrix} = \begin{bmatrix} \mathbf{y}_0 \\ \mathbf{y}_1 \\ \cdots \\ \mathbf{y}_{n-2} \\ \mathbf{y}_{n-1} \end{bmatrix} \tag{4.6}$$

where

$$\begin{aligned} d_0 &= b_0 \\ d_i &= b_i - a_i c_{i-1} / d_{i-1} \\ &= \frac{-a_i c_{i-1} + b_i d_{i-1}}{0 + 1 d_{i-1}} \qquad i = 1,2,\ldots,n-1 \end{aligned} \tag{4.7}$$

$$\begin{aligned} \mathbf{y}_0 &= \mathbf{r}_0 \\ \mathbf{y}_i &= \mathbf{r}_i - (a_i / d_{i-1})\mathbf{y}_{i-1} \qquad i = 1,2,\ldots,n-1 \end{aligned} \tag{4.8}$$

Back substitution provides the desired vector $\mathbf{x}$:

$$\begin{aligned} \mathbf{x}_{n-1} &= \mathbf{y}_{n-1} / d_{n-1} \\ \mathbf{x}_i &= (\mathbf{y}_i - c_i \mathbf{x}_{i+1}) / d_i \qquad i = n-2, n-3,\ldots,0 \end{aligned} \tag{4.9}$$

The tree algorithm is as follows.

(1) Use ROTRA to send c_{i-1} (contained in L_{i-1}) to L_i.

(2) Use FRACTION to determine d_i according to (4.7). L_i sends up the quadruple $(-a_i c_{i-1}, b_i, 0, 1)$ and receives (d_i, d_{i-1}).

(3) Use LR1 to determine $\mathbf{y}_i$ according to (4.8). L_i sends up the pair $(\mathbf{r}_i, -a_i / d_{i-1})$ and receives $\mathbf{y}_{i-1}$ and $\mathbf{y}_i$.

(4) Use BLR1 to determine $\mathbf{x}_i$ according to (4.9). L_i sends up the pair $(\mathbf{y}_i / d_i, -c_i / d_i)$ and receives $\mathbf{x}_i$ and $\mathbf{x}_{i+1}$.

The analysis of Gaussian elimination is summarized in Figure 4.2. This algorithm requires $O(\log n)$ time to execute.

2. LU Decomposition

The LU decomposition of the matrix A, if one exists, transforms A into a product of lower and upper bidiagonal matrices. Thus

$$A = LU = \begin{bmatrix} 1 & & & 0 \\ l_1 & 1 & & \\ & l_2 & 1 & \\ & & \ddots & \\ 0 & & l_{n-1} & 1 \end{bmatrix} \begin{bmatrix} u_0 & c_0 & & 0 \\ & u_1 & c_1 & \\ & & \ddots & \\ & & u_{n-2} & c_{n-2} \\ 0 & & & u_{n-1} \end{bmatrix}. \tag{4.10}$$

where

$$\begin{aligned} u_0 &= b_0 \\ l_i &= a_i / u_{i-1} \\ u_i &= b_i - l_i c_{i-1} \\ &= b_i - a_i c_{i-1} / u_{i-1} \qquad i = 1,2,\ldots,n-1 \end{aligned} \tag{4.11}$$

We rewrite (4.11) as

$$\begin{aligned} u_0 &= b_0 \\ u_i &= \frac{-a_i c_{i-1} + b_i u_{i-1}}{0 + 1 u_{i-1}} \qquad i = 1,2,\ldots,n-1 \end{aligned} \tag{4.12}$$

$$l_i = a_i / u_{i-1} \qquad i = 1,2,\ldots,n-1 \tag{4.13}$$

which expresses u_i as a quotient of two linear functions of u_{i-1}, and l_i as a function of u_i. We solve $LU\mathbf{x}=L\mathbf{y}=\mathbf{r}$ using forward substitution to obtain the intermediate vector $\mathbf{y}$,

$$\begin{aligned} \mathbf{y}_0 &= \mathbf{r}_0 \\ \mathbf{y}_i &= \mathbf{r}_i - l_i \mathbf{y}_{i-1} \qquad i=1,2,\ldots,n-1 \end{aligned} \tag{4.14}$$

and back substitution to obtain the desired vector $\mathbf{x}$:

$$\begin{aligned} \mathbf{x}_{n-1} &= \mathbf{y}_{n-1}/u_{n-1} \\ \mathbf{x}_i &= (\mathbf{y}_i - c_i \mathbf{x}_{i+1})/u_i \quad i=n-2,\, n-3,\ldots,0 \end{aligned} \tag{4.15}$$

In the tree implementation of this algorithm, each L cell holds the following variables:

a,b,c,r	coefficients of one equation
l,u,y	storage for intermediate values
x	solution of one equation
$seq\#$	L cell's sequence number, $0 \le seq\# \le n-1$

where, by definition, $a_0=c_{n-1}=0$. The algorithm proceeds as follows.

(1) Use ROTRA to send c_{i-1} (contained in L_{i-1}) to L_i.

(2) Compute u_i according to (4.12). Use FRACTION with L_i sending up the quadruple $(-a_i c_{i-1}, b_i, 0, 1)$ and receiving u_{i-1} and u_i.

(3) L_i computes l_i according to (4.13) using u_{i-1} obtained in step (2).

(4) Compute $\mathbf{y}_i$ according to (4.14). Use LR1 with L_i sending up the pair $(\mathbf{r}_i, -l_i)$ and receiving $\mathbf{y}_{i-1}$ and $\mathbf{y}_i$.

(5) Compute $\mathbf{x}_i$ according to (4.15). Use BLR1 with L_i sending up the pair $(\mathbf{y}_i/u_i, -c_i/u_i)$ and receiving $\mathbf{x}_{i-1}$ and $\mathbf{x}_i$.

The analysis of LU decomposition is summarized in Figure 4.3. This algorithm requires $O(\log n)$ time to execute.

A Method Using Second-Order Linear Recurrences

A basic principle of the theory of continued fractions [Wall48] provides another way to obtain u_i (4.12). First solve the second order linear recurrence

$$\begin{aligned} q_{-1} &= 1 \\ q_0 &= b_0 \\ q_i &= b_i q_{i-1} - a_i c_{i-1} q_{i-2} \qquad i = 1,\ldots,n-1 \end{aligned} \tag{4.16}$$

and then evaluate

$$u_i = q_i / q_{i-1} \qquad i = 0,1,\ldots,n-1. \tag{4.17}$$

We can prove (4.17) by induction.

(Basis)

For $i=0$, $u_0 = b_0 = b_0/1 = q_0/q_{-1}$.

(Induction)

Assuming that (4.17) is true for $i \geq 0$, we want to show that it is also true for $(i+1)$, i.e., $u_{i+1} = q_{i+1}/q_i$. But

$$\begin{aligned} u_{i+1} &= b_{i+1} - a_{i+1} c_i / u_i && \text{equation (4.11)} \\ &= b_{i+1} - a_{i+1} c_i / (q_i / q_{i-1}) && \text{by the induction hypothesis} \\ &= (b_{i+1} q_i - a_{i+1} c_i q_{i-1}) / q_i \\ &= q_{i+1} / q_i && \text{equation (4.16)} \ \blacksquare \end{aligned}$$

As with LU decomposition, we may then determine the $\mathbf{y}_i$ using forward substitution

$$\begin{aligned} \mathbf{y}_0 &= \mathbf{r}_0 \\ \mathbf{y}_i &= \mathbf{r}_i - l_i \mathbf{y}_{i-1} \qquad i = 1,2,\ldots,n-1 \end{aligned} \tag{4.18}$$

and the x_i using back substitution

$$\begin{aligned} \mathbf{x}_{n-1} &= \mathbf{y}_{n-1}/u_{n-1} \\ \mathbf{x}_i &= (\mathbf{y}_i - c_i\mathbf{x}_{i+1})/u_i \quad i=n-2, n-3,\ldots,0 \end{aligned} \tag{4.19}$$

The tree implementation of this algorithm uses LR2 as follows.

(1) Use ROTRA to send c_{i-1} (contained in L_{i-1} L cell) to L_i.

(2) Use LR2 to compute q_i according to (4.16). L_i sends up the 6-tuple $(0, b_i, -a_i c_{i-1}, \varphi, \varphi, \varphi)$ and receives q_i and q_{i-1}.

(3) L_i computes u_i according to (4.17) and l_i (4.13).

(4) Use LR1 to determine $\mathbf{y}_i$ according to (4.18). L_i sends up the pair $(\mathbf{r}_i, -l_i)$ and receives $\mathbf{y}_{i-1}$ and $\mathbf{y}_i$.

(5) Use BLR1 to determine $\mathbf{x}_i$ according to (4.19). L_i sends up the pair $(\mathbf{y}_i/u_i, -c_i/u_i)$ and receives $\mathbf{x}_{i-1}$ and $\mathbf{x}_i$.

The analysis of this variant of LU decomposition is summarized in Figure 4.4a. This algorithm requires $O(\log n)$ time to execute.

Recursive Doubling

Stone [Ston73a] describes recursive doubling, a parallel algorithm for solving the second-order linear recurrence (4.16) in $O(\log n)$ steps on a hypothetical n-processor machine similar in structure to an ILLIAC-IV. He observed that equation (4.16) can be transformed into the matrix recurrence relation

$$\begin{bmatrix} q_i \\ q_{i-1} \end{bmatrix} = \begin{bmatrix} b_i & -a_i c_{i-1} \\ 1 & 0 \end{bmatrix} \begin{bmatrix} q_{i-1} \\ q_{i-2} \end{bmatrix}, \quad i \geq 1. \tag{4.20}$$

If we let

$$Q_{-1} = \begin{bmatrix} 1 \\ 0 \end{bmatrix}, Q_i = \begin{bmatrix} q_i \\ q_{i-1} \end{bmatrix}, \text{ and } P_i = \begin{bmatrix} b_i & -a_i c_{i-1} \\ 1 & 0 \end{bmatrix}, \quad i \geq 0 \tag{4.21}$$

then

$$Q_i = P_i Q_{i-1} = \prod_{k=0}^{i} P_j \, Q_{-1} = \prod_{k=0}^{i} P_j \ , \quad i \geq 0. \tag{4.22}$$

To obtain $\prod_{k=0}^{i} P_j$, Stone suggests the algorithm depicted in Figure 4.4b, where $P_{ij} = \prod_{k=i}^{j} P_k$. Each row is considered to be a vector of elements. In the ith step, the current vector is multiplied by itself, element by element, with the entries shifted by 2^i elements. In the final step, the current vector is multiplied by itself with the multiplier delayed by $n/2$ elements. We see that in $\lceil \log n \rceil$ steps, all of the required products are obtained.

Unfortunately, recursive doubling as described by Stone cannot be implemented in a straightforward way on a tree machine. For if we distribute the matrices P_{ii}, $0 \leq i \leq n-1$, among the L cells, the final step of recursive doubling requires that the L cells in the left half of the tree send their values to L cells in the right half of the tree. As the root is the only communication link between these groups of cells, $O(n)$ amount of information must pass through the root. This for of the algorithm is therefore $O(n)$.

We can modify Stone's algorithm, however, to achieve $O(\log n)$ execution time. Note that (4.20) is a first-order linear recurrence whose variables are 2-vectors and whose coefficients are (2×2) matrices. Because matrix multiplication is associative, we may use a slightly modified version of LR1. The modification is a minor one: matrix multiplication replaces scalar multiplication. The tree algorithm proceeds as follows.

(1) Identical with step (1) of LU decomposition and LU decomposition variant.

(2) Use a modified version of LR1 to compute Q_i (4.22). L_i sends up the 4-tuple $(b_i, -a_i c_{i-1}, 1, 0)$ and receives $Q_i = (q_i, q_{i-1})$ and $Q_{i-1} = (q_{i-1}, q_{i-2})$. A T cell executes LR1 replacing scalar multiplication with matrix multiplication. Note that there is no matrix addition involved.

(3)-(5)

Identical with steps (3)-(5) of LU decomposition and LU decomposition variant.

The analysis of this variant of recursive doubling is summarized in Figure 4.4c. This algorithm requires $O(\log n)$ time to execute.

3. Methods Using Cyclic Reduction

Cyclic Reduction

Cyclic reduction is a method of solving (4.1) in $O(\log n)$ steps ([Hock65], [Ston75]). The method has two parts, elimination and back substitution. The basic step in the elimination phase reduces the number of variables in the system by half, transforming the tridiagonal system into another tridiagonal system half the size. For convenience, we consider the case of $n = 2^p - 1$, where p is a positive integer. Let

$$a_i \mathbf{x}_{i-1} + b_i \mathbf{x}_i + c_i \mathbf{x}_{i+1} = \mathbf{r}_i \tag{4.23}$$

be an odd-indexed (i=odd) equation. In the first elimination step, the even-indexed variables are eliminated from the odd-indexed equations, transforming (4.23) into

$$a' \mathbf{x}_{i-2} + b' \mathbf{x}_i + c' \mathbf{x}_{i+2} = \mathbf{r}' \tag{4.24}$$

where

$$
\begin{aligned}
a' &= b_{i+1}a_i a_{i-1} \\
b' &= b_{i+1}a_i c_{i-1} + b_{i-1}c_i a_{i+1} - b_{i-1}b_{i+1}b_i \\
c' &= b_{i-1}c_i c_{i+1} \\
r' &= b_{i+1}a_i r_{i-1} + b_{i-1}c_i r_{i+1} - b_{i-1}b_{i+1}r_i
\end{aligned}
\tag{4.25}
$$

The odd-indexed equations now form a new tridiagonal system, half the size of the original. This process is repeated on the new system until we are left with one equation in one unknown,

$$b\,\mathbf{x}_{2^p-1_{-1}} = \mathbf{r}\,. \tag{4.26}$$

After solving (4.26), back substitution begins. Back substitution traces back the steps carried out by elimination; variables are solved in the reverse order of their elimination. In the last iteration, for example, equations of the form (4.23) are solved for $\mathbf{x}_i$

$$\mathbf{x}_i = (r_i - a_i \mathbf{x}_{i-1} - c_i \mathbf{x}_{i+1}) / b_i \tag{4.27}$$

using the (now known) values for $\mathbf{x}_{i-1}$ and $\mathbf{x}_{i+1}$. Note that boundary variables are expressed as a function of only one variable. Figure 4.5 shows the movement of data when executing cyclic reduction for n=7.

To map this algorithm onto a tree machine, each L cell must contain the following registers:

a, b, c, r	the coefficients of one equation,
x	the solution of one equation, initially 0,
$seq\#$	L cell sequence number, $0 \le seq\# \le n-1$,
M	mask, explained later,

where, by definition, $a_0 = c_{n-1} = 0$.

As Figure 4.5 shows, in each step, some L cells must send data, other L cells must receive and process it, while still others do not participate at all. The main problem is coordinating all this activity. We accomplish this with the use of a mask (M), Atomic Rotate Left (ROTLA), and the analogous Atomic Rotate Right (ROTRA). The L cell's mask, computed from its sequence number, tells the L cell whether or not it is to send information to other L cells and whether or not it is to make use of information received from other L cells. The following algorithm initializes M:

$M := 1;$
while $(\text{mod}(seq\#, 2^M)=2^M-1)$ **do** $M := M+1;$

where mod(i, j) is the remainder obtained after dividing i by j. Figure 4.6 shows the sequence numbers and mask for $n=15$ L cells.

Figure 4.7 shows eight snapshots of the data communication that occurs during cyclic reduction among $n=7$ L cells. The circles represent L cells; solid circles are L cells that process data received. The arrows indicate communication. The analysis of cyclic reduction, summarized in Figure 4.8 shows that the implementation of cyclic reduction on a tree machine requires $O((\log n)^2)$ time.

Buneman Algorithm

Buzbee, Golub, and Nielson [BuGN70] and Stone [Ston75] describe the Buneman algorithm which is similar to cyclic reduction but with the desirable property of being stable in situations where cyclic reduction is not. Buneman's algorithm differs from cyclic reduction in the way **r** is computed in (4.25) during elimination, and the way **x** is computed in (4.26) and (4.27) during back substitution. During elimination, we replace the computation for **r**' (4.25) with the com-

putation of two new variables d' and e'

$$\begin{aligned} d' &= d_i + (e_i - a_i d_{i-1} - c_i d_{i+1})/b_i \\ e' &= b_{i+1}a_i e_{i-1} + b_{i-1}c_i e_{i+1} - d'(b_{i+1}a_i c_{i-1} + b_{i-1}a_{i+1}c_i) \end{aligned} \tag{4.28}$$

After elimination, we solve one equation in one unknown but, in place of (4.26), we use

$$b_i \mathbf{x}_i = \mathbf{r}_i = b_i d_i + e_i \tag{4.29}$$

or

$$\mathbf{x}_i = d_i + e_i / b_i \tag{4.30}$$

for $i=2^{p-1}-1$. During back substitution, in place of (4.27), we solve equations of the form

$$a_i \mathbf{x}_{i-1} + b_i \mathbf{x}_i + c_i \mathbf{x}_{i+1} = \mathbf{r}_i = b_i d_i + e_i \tag{4.31}$$

for $\mathbf{x}_i$, i.e.,

$$\mathbf{x}_i = d_i + (ei - a_i \mathbf{x}_{i-1} - c_i \mathbf{x}_{i+1})/b_i \tag{4.32}$$

where $\mathbf{x}_{i-1}$ and $\mathbf{x}_{i+1}$ are now known quantities. The analysis of the Buneman algorithm is summarized in Figure 4.9. This algorithm requires $O((\log n)^2)$ time to execute.

C. Iterative Methods

1. Jacobi and Jacobi Over-relaxation

The classical Jacobi algorithm transforms each equation of (4.1) into

$$b_i \mathbf{x}_i = -a_i \mathbf{x}_{i-1} - c_i \mathbf{x}_{i+1} + \mathbf{r}_i \tag{4.33}$$

$(i=0,\ldots,n-1)$ from which we obtain the iterative equation

$$x_i^{(k+1)} = (-a_i / b_i)x_{i-1}^{(k)} + (-c_i / b_i)x_{i+1}^{(k)} + (r_i / b_i) \qquad (4.34)$$

$(i=0,\ldots,n-1;\ k=0,\ldots,M)$. Initially, $x_i^{(0)}$ is assigned some value, possibly the result obtained from a direct tridiagonal linear system solver. Note that the equations for x_0 and x_{n-1} will involve two (not three) variables. For the tree algorithm, as before, we assume that all information pertinent to the ith equation is stored in L_i. For the Jacobi method, this includes the variables

a_i, b_i, c_i, r_i	coefficients of equation i
$x_i^{(k)}$, $x_{i-1}^{(k)}$, $x_{i+1}^{(k)}$	current approximation of x_i, x_{i-1}, x_{i+1}
$x_i^{(k+1)}$	new approximation of x_i
t_i	temporary.

By definition, $a_0=c_{n-1}=0$. Moreover, we assume that the quantities $(-a_i / b_i)$, $(-c_i / b_i)$, and (r_i / b_i) are evaluated only once at the start of the operation. The kth iteration of the Jacobi method requires two sweeps up and down the tree, as shown below.

(1) Use ROTRA to send $x_{i-1}^{(k)}$ (contained in L_{i-1}) to L_i which computes

$$t_i = (r_i / b_i) + (-a_i / b_i)x_{i-1}^{(k)}. \qquad (4.35)$$

(2) Use ROTLA to send $x_{i+1}^{(k)}$ (contained in L_{i+1}) to L_i which computes the new approximation for x_i

$$x_i^{(k+1)} = t_i + (-c_i / b_i)x_{i+1}^{(k)}. \qquad (4.36)$$

Note that, in step 1, L_0 receives $x_1^{(k)}$ and L_{n-1} receives $x_0^{(k)}$; these values should be ignored by the receiving L cell. This is handled by the tree machine by initializing a_0 and c_{n-1} to 0. Iteration may continue a fixed number of times or until a criterion for convergence has been fulfilled. The stopping criterion might

be

$$\max_i |\mathbf{x}_i^{(k+1)} - \mathbf{x}_i^{(k)}| < \varepsilon. \tag{4.37}$$

for some tolerance ε. After each iteration, a test may be performed in one sweep as described in step (3).

(3) L_i determines the value of $|\mathbf{x}_i^{(k+1)} - \mathbf{x}_i^{(k)}|$ and sends the result up the tree. A T cell sends to its father the larger of the two values it receives from its sons. The C cell compares the value it receives from its son to a pre-assigned value for ε and, depending upon the result, sends a "CONTINUE" or a "HALT" signal to the L cells.

The analysis of one iteration of the Jacobi method, not including setup operations, is shown in Figure 4.10. This algorithm requires O(log n) time per iteration.

The Jacobi over-relaxation (JOR) method is a generalization of the Jacobi method. JOR replaces equation (4.34) with

$$\mathbf{x}_i^{(k+1)} = (1-\omega)\mathbf{x}_i^{(k)} + \omega[(-a_i / b_i)\mathbf{x}_{i-1}^{(k)} + (-c_i / b_i)\mathbf{x}_{i+1}^{(k)} + (\mathbf{r}_i / b_i)] \tag{4.38}$$

where ω is called the relaxation factor, used to "overcorrect" or "undercorrect" iterates produced by the Jacobi method. Note that if $\omega=1$, the equations (4.34) and (4.38) are identical. Given that the quantities $(1-\omega)$, $(-\omega a_i / b_i)$, $(-\omega c_i / b_i)$, and $(\omega \mathbf{r}_i / b_i)$ are computed before iterating, the tree algorithm for one iteration of the JOR method proceeds as follows.

(1) Use ROTRA to send $x_{i-1}^{(k)}$ to L_i which computes

$$t_i = (1-\omega)\mathbf{x}_i^{(k)} + (-\omega a_i / b_i)\mathbf{x}_{i-1}^{(k)} + (-\omega \mathbf{r}_i / b_i). \qquad (4.39)$$

(2) Use ROTLA to send $\mathbf{x}_{i+1}^{(k)}$ to L_i which computes the new approximation for $\mathbf{x}_i$

$$\mathbf{x}_i^{(k+1)} = t_i + (-\omega c_i / b_i)\mathbf{x}_{i+1}^{(k)}. \qquad (4.40)$$

The analysis of the JOR method is shown in Figure 4.11. This algorithm requires $O(\log n)$ time per iteration.

2. Gauss-Seidel and Successive Over-relaxation

On a sequential computer, the Gauss-Seidel method is similar to the Jacobi method except that a new value for $\mathbf{x}_i$ is used as soon as it is available. In place of (4.34) we use

$$\mathbf{x}_i^{(k+1)} = (-a_i / b_i)\mathbf{x}_{i-1}^{(k+1)} + (-c_i / b_i)\mathbf{x}_{i+1}^{(k)} + (\mathbf{r}_i / b_i)\,. \qquad (4.41)$$

which requires the value of $\mathbf{x}_{i-1}^{(k+1)}$ in order to evaluate $\mathbf{x}_i^{(k+1)}$. The L cells prepare for iteration by computing and saving $(-a_i / b_i)$, $(-c_i / b_i)$, and $(\mathbf{r}_i / b_i)$. One iteration of the tree machine algorithm is shown below.

(1) Use ROTRA to send $\mathbf{x}_{i+1}$ to L_i which computes

$$t_i = (-c_i / b_i)\mathbf{x}_{i+1}^{(k)} + (\mathbf{r}_i / b_i) \qquad (4.42)$$

This reduces (4.41) to the first-order linear recurrence

$$\mathbf{x}_i^{(k+1)} = t_i + (-a_i / b_i)\mathbf{x}_{i-1}^{(k+1)}. \qquad (4.43)$$

(2) Use LR1 to solve (4.43). L_i sends up the pair $(t_i, -a_i / b_i)$ and receives $\mathbf{x}_i^{(k+1)}$ and $\mathbf{x}_{i-1}^{(k+1)}$.

The analysis of the Gauss-Seidel method is summarized in Figure 4.12. This algorithm requires $O(\log n)$ time per iteration.

The successive over-relaxation method (SOR) is a generalization of the Gauss-Seidel method. Like the JOR method, it uses a relaxation parameter ω to correct the current approximation by a smaller (if $\omega<1$) or a larger (if $\omega>1$) amount than would the Gauss-Seidel method. The SOR method uses the following equation.

$$\mathbf{x}_i^{(k+1)} = (1-\omega)\mathbf{x}_i^{(k)} + \omega[(-a_i / bi)\mathbf{x}_{i-1}^{(k+1)} + (-c_i / b_i)\mathbf{x}_{i+1}^{(k)} + (\mathbf{r}_i / b_i)] . \quad (4.44)$$

Initially, L_i computes and saves the quantities $(1-\omega)$, $(-\omega a_i / b_i)$, $(-\omega c_i / b_i)$, and $(\omega \mathbf{r}_i / b_i)$. One iteration would proceed as follows.

(1) Use ROTLA to send $\mathbf{x}_{i+1}^{(k)}$ to L_i which evaluates part of the right hand side of (4.44)

$$t_i = (1-\omega)\mathbf{x}_i^{(k)} + \omega[(-c_i / b_i)\mathbf{x}_{i+1}^{(k)} + (\mathbf{r}_i / b_i)] \quad (4.45)$$

reducing (4.44) to a first-order linear recurrence

$$\mathbf{x}_i^{(k+1)} = t_i + (-\omega a_i / b_i)\mathbf{x}_{i-1}^{(k+1)} . \quad (4.46)$$

(2) Use LR1 to solve this recurrence. L_i sends up the pair $(t_i, -\omega a_i / b_i)$ and receives $\mathbf{x}_i^{(k+1)}$ and $\mathbf{x}_{i+1}^{(k+1)}$, discarding the latter (it is not needed).

The analysis of SOR is shown in Figure 4.13. This algorithm requires $O(\log n)$ time per iteration.

3. Red-black Successive Over-relaxation

Lambiotte and Voigt [LaVo75] point out that it is impossible to implement the SOR method expressed in its usual form (4.44) efficiently on vector computers. This is because, on a vector computer, all components needed to execute a

vector instruction must be known before the instruction is executed. Equation (4.44), however, requires that the ith component of the $(k+1)$th approximation, $\mathbf{x}_i^{(k+1)}$, be obtained from the $(i-1)$th component of the $(k+1)$th approximation, which is not known at the start of the vector operation. They suggest that it is possible to reorder the equations, called a red-black ordering [Youn71], so that a modified version of SOR can be implemented on a vector computer.

The modification amounts to separating the odd-indexed equations from the even-indexed equations. The $(k+1)$th approximation of the even-indexed variables is obtained using the kth approximation of the odd-indexed variables. Then the $(k+1)$th approximation of the odd-indexed variables is obtained using the (now known) $(k+1)$th approximation of the even-indexed variables. I.e.,

$$\mathbf{x}_i^{(k+1)} = (1-\omega)\mathbf{x}_i^{(k)} + \omega[(-a_i/bi)\mathbf{x}_{i-1}^{(k)} + (-c_i/b_i)\mathbf{x}_{i+1}^{(k)} + (\mathbf{r}_i/b_i)] \qquad (4.47)$$

for i=0, 2, 4, ..., followed by

$$\mathbf{x}_i^{(k+1)} = (1-\omega)\mathbf{x}_i^{(k)} + \omega[(-a_i/bi)\mathbf{x}_{i-1}^{(k+1)} + (-c_i/b_i)\mathbf{x}_{i+1}^{(k+1)} + (\mathbf{r}_i/b_i)] \qquad (4.48)$$

for i=1, 3, 5, ..., with the equations for $\mathbf{x}_0$ and $\mathbf{x}_{n-1}$ involving one less term.

The tree algorithm for red-black SOR is straightforward with the help of the L cell's sequence number ($seq\#$) as shown below. It is similar to the JOR tree algorithm applied to half of the set of equations each time. As before, we set a_0=c_{n-1}=0 and the L cells compute and save the values $(1-\omega)$, $(-\omega a_i/b_i)$, $(-\omega c_i/b_i)$, and $(\omega \mathbf{r}_i/b_i)$ before the start of the iteration.

(1) Use ROTLA to send $\mathbf{x}_{i+1}^{(k)}$ to L_i. All L cells receive a value but only L cells with even sequence numbers compute

$$t_i = (1-\omega)\mathbf{x}_i^{(k)} + \omega[(-c_i / b_i)\mathbf{x}_{i+1}^{(k)} + (\mathbf{r}_i / b_i)] \quad \textbf{(4.49)}$$

(2) Use ROTRA to send $\mathbf{x}_i^{(k)}$ to L_i. Again, all L cells receive a value but only L cells with even sequence numbers replace their current x-approximation with

$$\mathbf{x}_i^{(k+1)} = t_i + (-\omega a_i / b_i)\mathbf{x}_{i-1}^{(k)} \quad (4.50)$$

(3) Use ROTLA to allow L_{i+1} to send its *current* x-approximation to L_i. Note that the odd-numbered L cells will be sending $\mathbf{x}_i^{(k)}$ while the even-numbered L cells will be sending $\mathbf{x}_i^{(k+1)}$. Only the odd-numbered L cells will compute

$$t_i = (1-\omega)\mathbf{x}_i^{(k)} + \omega[(-c_i / b_i)\mathbf{x}_{i+1}^{(k+1)} + (\mathbf{r}_i / b_i)] \quad (4.51)$$

(4) Use ROTRA to allow L_{i-1} to send its current x-approximation to L_i. Only the odd-numbered L cells replace their current x-approximation with

$$\mathbf{x}_i^{(k+1)} = t_i + (-\omega a_i / b_i)\mathbf{x}_{i-1}^{(k+1)} \quad (4.52)$$

The analysis of red-black SOR is shown in Figure 4.14. This algorithm requires $O(\log n)$ time per iteration.

4. Parallel Gauss: An Iterative Analog of LU Decomposition

Traub [Trau73] observed that the equations describing methods such as Gaussian elimination, the Thomas algorithm, and LU decomposition are not well-suited for execution on vector computers. He developed a method, called parallel Gauss, which he compared to the Jacobi, JOR, Gauss-Seidel, and SOR methods on a model problem run on a PDP-10 in APL.

Traub's method transforms the three equations for LU decomposition into iterative equations. Thus, equations (4.11-14) are replaced by (4.53-56).

$$\begin{aligned} u_i^{(0)} &= b_i \\ u_0^{(k+1)} &= b_0 \\ u_i^{(k+1)} &= b_i - a_i c_i / u_{i-1}^{(k)} \end{aligned} \tag{4.53}$$

for $i=1, \ldots, (n-1)$ and $k=0, \ldots, M-1$. Let $\bar{u}=u^{(M)}$.

$$l_i = a_i / \bar{u}_{i-1} \tag{4.54}$$

$$\begin{aligned} y_i^{(0)} &= r_i \\ y_0^{(k+1)} &= r_0 \end{aligned}$$

$$y_i^{(k+1)} = r_i - l_i y_{i-1}^{(k)} \tag{4.55}$$

for $i=1, \ldots, (n-1)$ and $k=0, \ldots, N-1$. Let $\bar{y}=y^{(N)}$.

$$\begin{aligned} x_i^{(0)} &= \bar{y}_i / \bar{u}_i \\ x_{n-1}^{(k+1)} &= \bar{y}_{n-1} / \bar{u}_{n-1} \\ x_i^{(k+1)} &= (\bar{y}_i - c_i x_{i+1}^{(k)}) / \bar{u}_i \end{aligned} \tag{4.56}$$

for $i=(n-2), \ldots, 0$ and $k=0, \ldots, P-1$. Let $\bar{x}=x^{(P)}$.

For each of the iterative equations (4.53, 4.55 and 4.56), Traub noted that after the jth iteration, $u_i^{(j)}$, $y_i^{(j)}$, and $x_i^{(j)}$, are correct for $i \leq j+1$. This means that each iteration requires at most $(n-1)$ iterations. This presents a savings for vector computers since these components need not be recomputed, i.e. the length of the vector processed decreases by one in each iteration.

The tree algorithm for Traub's method is shown below.

(1) Compute $\bar{u}_i$. L_i initializes $u_i=b_i$. In the kth iteration, use ROTRA to send $u_{i-1}^{(k)}$ to L_i which computes (4.53). Iterate M times, i.e., until convergence

occurs.

(2) Compute l_i. L_i evaluates (4.54).

(3) Compute $\bar{\mathbf{y}}_i$. L_i initializes $\mathbf{y}_i=\mathbf{r}_i$. In the kth iteration, use ROTRA to send $y_{i-1}^{(k)}$ to L_i which computes (4.55). Iterate N times.

(4) Compute $\bar{\mathbf{x}}_i$. L_i initializes $\mathbf{x}_i=\bar{y}_i / \bar{u}_i$. In the kth iteration, use ROTLA to send $x_{i+1}^{(k)}$ to L_i which computes (4.56). Iterate P times.

The analysis of one iteration of Traub's method is shown in Figure 4.15. This algorithm requires $O(\log n)$ time per iteration.

D. Summary and Conclusions

1. General Remarks

Each tridiagonal linear system solver described in the previous sections is composed in a simple manner of algorithms described in Chapter 3, namely LR1, BLR1, LR2, FRACTION, ROTLA, and ROTRA. Therefore, the variations applicable to LR1, *etc.*, are also applicable to the tridiagonal system algorithms. These variations are described below.

Empty L cells. It may happen that some of the L cells are empty or do not participate in the tridiagonal system algorithm. What the nonparticipating L cells do depends on the component algorithms of the tridiagonal system solver.

As an example, consider the Thomas algorithm (§4.B.1). Its component algorithms are FRACTION, LR1, and BLR1. During the execution of FRACTION, empty L cells should send up the quadruple (0, 1, 1, 0) and receive two (meaningless) values, as described in Chapter 3. Similarly, for LR1 and BLR1, empty L

cells should send up the pair (0, 1) and receive two values. In short, it is not necessary to make any modification to the component algorithms in order to use them in the implementation of the Thomas algorithm or of any of the other tridiagonal system solvers.

Solving several independent tridiagonal linear systems simultaneously using a direct method. It is possible to solve more than one tridiagonal linear system simultaneously, provided that there are enough L cells to accommodate all of the systems, with one equation occupying one L cell, and all systems are solved by the same method. An immediate concern are the boundaries between different systems. We find, however, that if the coefficients of the boundary equations are properly initialized, the algorithms execute correctly.

As an example, consider the method of LU decomposition (§4.B.2, equations (4.10-15)) simultaneously applied to two tridiagonal linear systems. Recall that each L cell contains the following variables:

a,b,c,r	coefficients of one equation
l,u,y	storage for intermediate values
x	solution of one equation

where, by definition, $a_0=c_{n-1}=0$. For the convenience of the reader, we repeat equations (4.10), (4.12), and (4.13) here.

$$A = LU = \begin{bmatrix} 1 & & & 0 \\ l_1 & 1 & & \\ & l_2 & 1 & \\ & & \ddots & \\ 0 & & l_{n-1} & 1 \end{bmatrix} \begin{bmatrix} u_0 & c_0 & & 0 \\ & u_1 & c_1 & \\ & & \ddots & \\ & & u_{n-2} & c_{n-2} \\ 0 & & & u_{n-1} \end{bmatrix}. \qquad (4.10)$$

where

$$u_0 = b_0$$
$$u_i = \frac{-a_i c_{i-1} + b_i u_{i-1}}{0 + 1 u_{i-1}} \qquad i=1,2,\ldots,n-1 \tag{4.12}$$

and

$$l_i = a_i / u_{i-1} \qquad i=1,2,\ldots,n-1 \tag{4.13}$$

which expresses u_i as a quotient of two linear functions of u_{i-1}, and l_i as a function of u_i. The component algorithms of LU decomposition are ROTRA, FRACTION, LR1, and BLR1. Figure 4.16 shows the L cells which contain the two linear systems. Let L_i represent the L cell with sequence number i. The first system is stored in cells L_0 through L_{n-1}; the second is stored in L_n through L_{n+m-1}.

The first step uses ROTRA to send c_{i-1} (stored in the $(i-1)$th L cell) to the ith L cell. This causes L_0 and L_n to receive c_{n+m-1} and c_{n-1} respectively. Note, however, that a_0 and a_n will be multiplied with c_{n+m-1} and c_{n-1} in the next step, and that all four terms are 0.

The second step uses FRACTION to evaluate equation (4.12). Each L cell must send up the quadruple $(-a_i c_{i-1}, b_i, 0, 1)$. L_0 and L_n have the added condition that the first component, i.e. $-a_i c_{i-1}$, must be 0. We see that this condition is satisfied as a_0, c_{n+m-1}, a_n, and c_{n-1} are all 0. FRACTION, therefore, executes correctly.

The third step is for each L cell to compute l_i using equation (4.13). The cells L_0 and L_n must have l_0 and l_n equal to 0. This is, in fact, the case because a_0 and a_n are both 0.

This continues through the rest of the algorithm. The algorithm does not need to make a special case of the boundary L cells because the proper initialization to 0 of some of their coefficients assures the correct evaluation of other values.

Solving several independent tridiagonal linear systems simultaneously using an iterative method. It is also possible to solve several systems simultaneously with an iterative method. As with a direct method, the potential problem of error occurring in the boundary L cells is avoided by proper initialization of the boundary coefficients. The number of iterations needed to solve all of the systems is the maximum number needed to solve any one of the systems, i.e., it will take as long to solve all of the systems as the least convergent of the systems.

Solving the same system for different constant values. The LU decomposition, LU decomposition variant, and recursive doubling methods allow the user to solve the same tridiagonal system for different values without having to decompose the coefficient matrix each time. The implementation is straightforward.

The coefficient matrix is decomposed once at the start of the operation. One set of constants is stored in the L cells and the algorithms LR1 and BLR1 are executed. The solution values are flushed out, a new set of constants is read in, and the tree is ready to execute LR1 and BLR1 once again. This continues as many times as there are sets of constants.

2. Comparison of the Tree Algorithms

Figure 4.17 shows a summary of the complexity of the direct tridiagonal system solvers. All methods except cyclic reduction and the Buneman algorithm require $O(\log n)$ time. Although cyclic reduction and the Buneman algorithm execute $O(\log n)$ arithmetic operations, communication requires $O((\log n)^2)$ time. The increased communication time could be acceptable if it were

offset by a significant reduction in the number of arithmetic operations. As this is not the case, neither cyclic reduction nor the Buneman algorithm can compete with the other, more traditional, algorithms.

Of the remaining, the LU decomposition variant is the least desirable. It requires at least 60% more parallel additions, 25% more parallel multiplications, and 10% more communication time than the Thomas algorithm, Gaussian elimination, LU decomposition, or recursive doubling variant. This is because of its use of LR2 which, as has been pointed out in Chapter 3, is considerably more expensive than LR1 and somewhat more expensive than FRACTION.

The three traditional tridiagonal linear system solvers and recursive doubling emerge as the best, with the Thomas algorithm holding a slight communication time edge over Gaussian elimination and LU decomposition. LU decomposition, of course, is to be preferred if one must solve many different linear systems using the same coefficient matrix.

Comparing the iterative methods (Figure 4.18) is more difficult as the rate of convergence of the methods must also be considered. Considering only arithmetic operations and communication time per iteration, the Gauss-Seidel and SOR methods compare poorly with the Jacobi and JOR methods. However, the Gauss-Seidel method has a rate of convergence approximately twice that of the Jacobi method [Arnes77], provided both converge. For the particular case of solving Laplace's equation on a square region, if h is the space between grid points (see Figure 2.1), Ames shows that computation time is reduced by the factor $2h^{-1}$ if one uses SOR rather than the Gauss-Seidel method. However, one must consider the overhead of obtaining the optimum relaxation factor ω before deciding to use SOR over Gauss-Seidel. The parallel Gauss method described by Traub has the least amount of computation per iteration. However, one must

realize that this method actually solves each of three equations iteratively and in sequence. The total computation time may be more than the other iterative methods. The decision of which iterative method to choose, therefore, must depend on the characteristics of the particular linear system one is solving and on an analytic estimate of the number of iterations each method would require for convergence.

3. Comparison with Sequential, Vector and Array Algorithms

Lambiotte and Voigt [LaVo75] studied the implementation of direct and iterative tridiagonal linear system solvers on a vector processor, the CDC STAR-100. Stone [Ston75] analyzed direct tridiagonal linear system algorithms for a hypothetical parallel processor similar in structure to an ILLIAC-IV. Their results are compared with the complexity of the same algorithms, implemented on sequential processors and on tree machines, in Figures 4.19 and 4.20. An entry in brackets means that the method was not implemented in the papers mentioned but the complexity of the algorithm may be inferred from the methods that were. A blank entry means that inference is difficult to make.

Lambiotte and Voigt point out that LU decomposition cannot be efficiently implemented on a vector computer, primarily because of the sequential nature of the equations to be solved. They *do* implement and analyze it, however, in order to compare it with other, more efficiently implementable, methods. Because the Thomas algorithm and Gaussian elimination use equations similar in structure to those of LU decomposition, we may infer from their study that both the Thomas algorithm and Gaussian elimination also require $O(n)$ time on a vector computer. Similarly, because the Buneman algorithm is similar to cyclic reduction, we expect that the Buneman algorithm also requires $O(n)$ time. With

this in mind, we see from Figure 4.19 that tree machines consistently do better (asymptotically) than either sequential or vector processors.

Stone asserts, for reasons similar to Lambiotte and Voigt's, that conventional tridiagonal linear system solvers cannot do better than $O(n)$ time on an array processor. Consequently, he does not present an implementation of either the Thomas algorithm, Gaussian elimination, or LU decomposition in his paper. If his premise is correct, we see from Figure 4.19 that, except for cyclic reduction or the Buneman algorithm, tree machines match or better the complexity of algorithms run on array processors. Recall that for cyclic reduction and the Buneman algorithm, tree machines required $O(\log n)$ arithmetic operations and $O((\log n)^2)$ communication time. Stone's analysis considered only arithmetic operations and did not take into account costs of data routing, arrangement, and rearrangement, which may make his estimates unrealistically low.

Lambiotte and Voigt also analyzed three iterative methods, the Jacobi method, red-black successive over-relaxation, and Traub's parallel Gauss algorithm. They conclude that methods such as the Gauss-Seidel method and successive over-relaxation are not efficient on a vector computer because of the sequential nature of the equations involved. Figure 4.20 compares the orders of complexity of six different iterative methods when implemented on sequential, vector, and tree processors and it shows that tree machines produce consistently better results.

Thomas Algorithm								
Step	Swps	Comm. Time	Parallel Operations					
			L cells		T cells		C cell	
			+	×	+	×	+	×
1. FRAC	1	$6\log n + 6$	0	0	$6\log n$	$11\log n$	0	1
2. LR1	1	$4\log n + 4$	1	4	$2\log n$	$3\log n$	0	0
3. BLR1	1	$4\log n + 4$	0	1	$2\log n$	$3\log n$	0	0
Total	3	$14\log n + 14$	0	5	$10\log n$	$17\log n$	0	1

Figure 4.1 Analysis of the Thomas Algorithm. The Thomas algorithm requires three sweeps through the tree. In the first sweep, we use FRAC and solve for the intermediate value e_i. The second sweep uses LR1 to evaluate $\mathbf{y}_i$. The final sweep evaluates the desired values $\mathbf{x}_i$.

Gaussian Elimination								
Step	Swps	Comm. Time	Parallel Operations					
			L cells		T cells		C cell	
			+	×	+	×	+	×
1. ROTRA	1	$2\log n + 2$	0	0	0	0	0	0
2. FRAC	1	$6\log n + 6$	0	2	$6\log n$	$11\log n$	0	1
3. LR1	1	$4\log n + 4$	0	2	$2\log n$	$3\log n$	0	0
4. BLR1	1	$4\log n + 4$	0	3	$2\log n$	$3\log n$	0	0
Total	4	$16\log n + 16$	0	7	$10\log n$	$17\log n$	0	1

Figure 4.2 Analysis of Gaussian elimination. Gaussian elimination requires one sweep through the tree more than the Thomas algorithm because of the need of L_i for c_i prior to the computation of d_i. The number of parallel operations for both methods, however, is the same.

LU Decomposition								
Step	Swps	Comm. Time	Parallel Operations					
			L cells		T cells		C cell	
			+	×	+	×	+	×
1. ROTRA	1	$2\log n + 2$	0	0	0	0	0	0
2. FRAC	1	$6\log n + 6$	0	2	$6\log n$	$11\log n$	0	1
3.	0	0	0	1	0	0	0	0
4. LR1	1	$4\log n + 4$	0	1	$2\log n$	$3\log n$	0	0
5. BLR1	1	$4\log n + 4$	0	3	$2\log n$	$3\log n$	0	0
Total	4	$16\log n + 16$	0	7	$10\log n$	$17\log n$	0	1

Figure 4.3 Analysis of LU decomposition. We need four sweeps through the tree machine. The first sweep sends c_{i-1} to L_i. The second sweep uses FRAC to obtain the u_i. Note that after the second sweep, L_i has received both u_i and u_{i-1}. It can immediately compute $l_i = a_i / u_{i-1}$. The third sweep (step 4) computes the intermediate values $\mathbf{y}_i$. The final sweep gives the desired values $\mathbf{x}_i$.

LU Decomposition Variant								
Step	Swps	Comm. Time	Parallel Operations					
			L cells		T cells		C cell	
			+	×	+	×	+	×
1. ROTRA	1	$2\log n + 2$	0	0	0	0	0	0
2. LR2	1	$10\log n + 10$	0	2	$12\log n$	$16\log n$	0	0
3.	0	0	0	2	0	0	0	0
4. LR1	1	$4\log n + 4$	0	1	$2\log n$	$3\log n$	0	0
5. BLR1	1	$4\log n + 4$	0	3	$2\log n$	$3\log n$	0	0
Total	4	$20\log n + 20$	0	8	$16\log n$	$22\log n$	0	0

Figure 4.4a Analysis of LU decomposition variant. After the two first sweeps (Steps 1-2), q_i has been determined. Step 3 computes u_i and l_i. Steps 4 and 5 are similar to Steps 4 and 5 of LU decomposition.

	P_{00}	P_{11}	P_{22}	P_{33}	P_{44}	P_{55}	P_{66}	P_{77}
×		P_{00}	P_{11}	P_{22}	P_{33}	P_{44}	P_{55}	P_{66}
=	P_{00}	P_{01}	P_{12}	P_{23}	P_{34}	P_{45}	P_{56}	P_{67}
×			P_{00}	P_{01}	P_{12}	P_{23}	P_{34}	P_{45}
=	P_{00}	P_{01}	P_{02}	P_{03}	P_{14}	P_{25}	P_{36}	P_{47}
×					P_{00}	P_{01}	P_{02}	P_{03}
=	P_{00}	P_{01}	P_{02}	P_{03}	P_{04}	P_{05}	P_{06}	P_{07}

Figure 4.4b Recursive doubling for n=8, where $P_{ij} = \prod_{k=i}^{j} P_k$. In $\log n = 3$ steps, the desired products P_{0i}, $0 \le i \le 7$ are obtained.

Recursive Doubling Variant								
Step	Swps	Comm. Time	Parallel Operations					
			L cells		T cells		C cell	
			+	×	+	×	+	×
1. ROTRA	1	$2\log n + 2$	0	0	0	0	0	0
2. LR1M	1	$8\log n + 8$	0	2	$6\log n$	$12\log n$	0	0
3.	0	0	0	2	0	0	0	0
4. LR1	1	$4\log n + 4$	0	1	$2\log n$	$3\log n$	0	0
5. BLR1	1	$4\log n + 4$	0	3	$2\log n$	$3\log n$	0	0
Total	4	$18\log n + 18$	0	8	$10\log n$	$18\log n$	0	0

Figure 4.4c Analysis of recursive doubling variant. LR1M is a modified version of LR1 in which the coefficient of the recurrence is a (2×2) matrix, the variables are 2-vectors, and the recurrence equation has no constant term. Consequently, the T cells must perform one matrix multiplication during the upward sweep and one matrix-vector multiplication during the downward sweep.

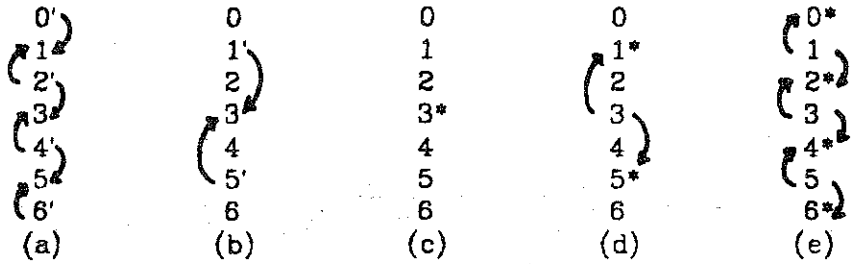


Figure 4.5 Data communication in cyclic reduction. Elimination takes place in (a) and (b) while back substitution takes place in (d) and (e). The numbers represent equations. An arrow means that the coefficients of one equation are required by (and sent to) another. A prime (') means that an equation has been eliminated. A star (*) means that the equation has been solved. In (a), equation i (i=1,3,5) receives the coefficients of equations ($i-1$) and ($i+1$); equation i is modified, eliminating the even-indexed variables. In (b) the step is repeated with only equations 1, 3, and 5 participating. In (c), equation 3 has become an equation in one variable (i.e., $\mathbf{x}_3$). We solve for $\mathbf{x}_3$. In (d), the value of $\mathbf{x}_3$ is sent to (the modified) equations 1 and 5 which are now able to solve for $\mathbf{x}_1$ and $\mathbf{x}_5$ respectively. In (e), the rest of the variables are evaluated.

Sequence#:	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
Mask:	1	2	1	3	1	2	1	4	1	2	1	3	1	2	1

Figure 4.6 L cell sequence numbers and mask.

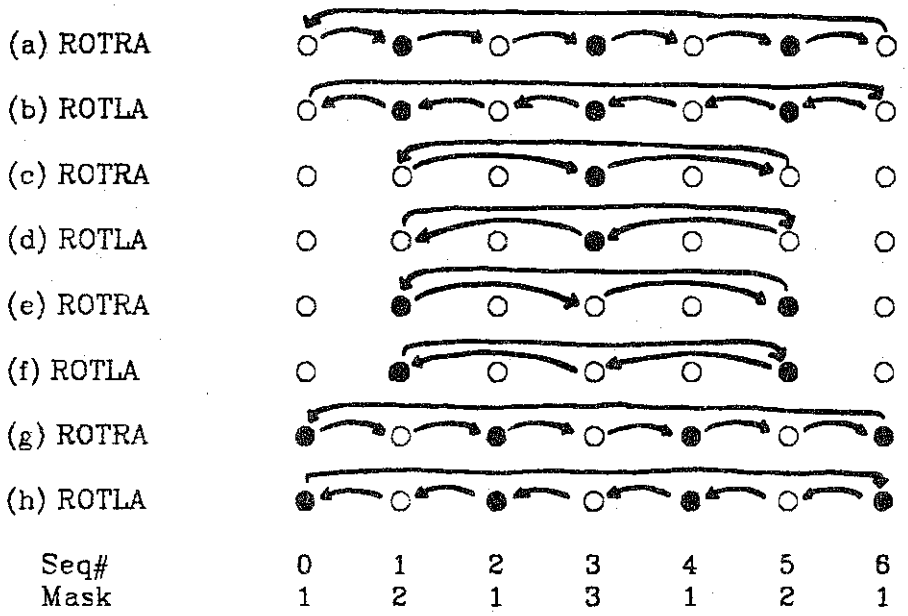


Figure 4.7 Communication among the L cells during cyclic reduction for $n=7$. In the first elimination step, the even-indexed L cells must send their coefficients to the odd-indexed L cells. To do this, we use ROTRA (a) and ROTLA (b). In both figures, L cells with $M \geq 1$ are programmed to send and receive information, but only L cells with $M>1$ (solid circles) are programmed to process the information. In general, during the ith elimination step, ($i=1,2,\ldots,\log(n+1)-1$), L cells with $M \geq i$ send and receive information while only L cells with $M>i$ process it. (c) and (d) show the movement of data during the second elimination step. In the ith back substitution step, ($i=\log(n+1)-1,\ldots,1$), L cells with $M \geq i$ send and receive information while only those with $M=i$ process it. (e) through (h) show the movement of data during back substitution.

Cyclic Reduction								
Operation	Swps	Comm. Time	Parallel Operations					
			L cells		T cells		C cell	
			+	×	+	×	+	×
Elimin	$2(h-1)$	$2(h-1)(2h+5)$	$4(h-1)$	$16(h-1)$	0	0	0	0
Solve	0	0	0	1	0	0	0	0
Backsub	$2(h-1)$	$2(h-1)(2h+5)$	$2(h-1)$	$3(h-1)$	0	0	0	0
Total	$4(h-1)$	$4(h-1)(2h+5)$	$6(h-1)$	$19(h-1)+1$	0	0	0	0

Figure 4.8 Analysis of cyclic reduction, $h=\log(n+1)$. Both elimination and back substitution require $(h-1)$ iterations and each iteration involves one ROTRA and one ROTLA; hence, each requires $2(h-1)$ up and down sweeps through the tree. In one execution of ROTRA (or ROTLA), all cells send 4 atoms to their fathers during the upward sweep and 4 atoms to their sons during the downward sweep. Pipelining allows a T cell to send data from its son (father) to its father (son) as soon as the data is received. With pipelining, one execution of ROTRA requires $2h+5$ communication steps. As elimination and back substitution each take $2(h-1)$ sweeps, a total of $4(h-1)(2h+5)$ communication steps are required. Only the L cells execute any arithmetic instructions. During elimination, selected L cells solve for a', b', c', r', as found in equation (4.22), after every 2 sweeps. After elimination, one L cell executes one division (4.23). During back substitution, selected L cells evaluate (4.24). The entire operation requires $O(h^2)=O((\log n)^2)$ time.

Buneman Algorithm								
Operation	Swps	Comm. Time	Parallel Operations					
			L cells		T cells		C cell	
			+	×	+	×	+	×
Elimin	$2(h-1)$	$2(h-1)(2h+5)$	$8(h-1)$	$20(h-1)$	0	0	0	0
Solve	0	0	1	1	0	0	0	0
Backsub	$2(h-1)$	$2(h-1)(2h+5)$	$3(h-1)$	$3(h-1)$	0	0	0	0
Total	$4(h-1)$	$4(h-1)(2h+5)$	$11(h-1)+1$	$23(h-1)+1$	0	0	0	0

Figure 4.9 Analysis of the Buneman algorithm, $h=\log(n+1)$. The stability provided by this algorithm (compared to cyclic reduction) comes at the cost of added parallel additions and multiplications.

Jacobi (per iteration)								
Step	Swps	Comm. Time	Parallel Operations					
			L cells		T cells		C cell	
			+	×	+	×	+	×
1. ROTRA	1	$2\log n + 2$	1	1	0	0	0	0
2. ROTLA	1	$2\log n + 2$	1	1	0	0	0	0
3. CTEST	1	$2\log n + 2$	1	0	0	0	0	0
Total	3	$6\log n + 6$	3	2	0	0	0	0

Figure 4.10 Analysis of the Jacobi method. It is assumed that the quantities $-a_i / b_i$, $-c_i / b_i$, and r_i / b_i have been computed and saved once, prior to the first iteration. Three sweeps are needed. L_i receives data from L_{i-1} evaluates (4.32) in the first, receives data from L_{i+1} evaluates (4.33) in the second, and conducts a convergence test (4.34) in the third.

Jacobi Over-relaxation (per iteration)								
Step	Swps	Comm. Time	Parallel Operations					
			L cells		T cells		C cell	
			+	×	+	×	+	×
1. ROTRA	1	$2\log n + 2$	2	2	0	0	0	0
2. ROTLA	1	$2\log n + 2$	1	1	0	0	0	0
3. CTEST	1	$2\log n + 2$	1	0	0	0	0	0
Total	3	$6\log n + 6$	4	3	0	0	0	0

Figure 4.11 Analysis of the JOR method. The proper selection of the relaxation parameter ω provides faster convergence for the JOR method (compared to the Jacobi method). The cost is one added multiplication and addition per iteration.

Gauss-Seidel (per iteration)								
Step	Swps	Comm. Time	Parallel Operations					
			L cells		T cells		C cell	
			+	×	+	×	+	×
1. ROTLA	1	$2\log n + 2$	1	1	0	0	0	0
2. LR1	1	$4\log n + 4$	0	0	$2\log n$	$3\log n$	0	0
3. CTEST	1	$2\log n + 2$	1	0	0	0	0	0
Total	2	$8\log n + 8$	2	1	$2\log n$	$3\log n$	0	0

Figure 4.12 Analysis of Gauss-Seidel.

SOR (per iteration)								
Step	Swps	Comm. Time	Parallel Operations					
			L cells		T cells		C cell	
			+	×	+	×	+	×
1. ROTLA	1	$2\log n + 2$	2	2	0	0	0	0
2. LR1	1	$4\log n + 4$	0	0	$2\log n$	$3\log n$	0	0
3. CTEST	1	$2\log n + 2$	1	0	0	0	0	0
Total	2	$8\log n + 8$	3	2	$2\log n$	$3\log n$	0	0

Figure 4.13 Analysis of SOR.

Red-Black SOR (per iteration)								
Step	Swps	Comm. Time	Parallel Operations					
			L cells		T cells		C cell	
			+	×	+	×	+	×
1. ROTRA	1	$2\log n + 2$	2	2	0	0	0	0
2. ROTLA	1	$2\log n + 2$	1	1	0	0	0	0
3. ROTRA	1	$2\log n + 2$	2	2	0	0	0	0
4. ROTLA	1	$2\log n + 2$	1	1	0	0	0	0
5. CTEST	1	$2\log n + 2$	1	0	0	0	0	0
Total	5	$10\log n + 10$	7	6	0	0	0	0

Figure 4.14 Analysis of red-black SOR.

Parallel Gauss (per iteration)								
Step	Swps	Comm. Time	Parallel Operations					
			L cells		T cells		C cell	
			+	×	+	×	+	×
1. ROTRA	1	$2\log n + 2$	1	1	0	0	0	0
CTEST	1	$2\log n + 2$	1	0	0	0	0	0
2.	0	0	0	1	0	0	0	0
3. ROTRA	1	$2\log n + 2$	1	1	0	0	0	0
CTEST	1	$2\log n + 2$	1	0	0	0	0	0
4. ROTLA	1	$2\log n + 2$	1	1	0	0	0	0
CTEST	1	$2\log n + 2$	1	0	0	0	0	0

Figure 4.15 Analysis of Parallel Gauss.

L cells	O	...	O	O	...	O
seq#	0		$n-1$	n		$n+m-1$

Figure 4.16 Solving two tridiagonal linear systems simultaneously. The first system occupies the L cells with sequence numbers 0 through $n-1$. The second system occupies the L cells with sequence numbers n through $n+m-1$.

Summary: Direct Methods								
Method	Swps	Comm. Time	Parallel Operations					
			L cells		T cells		C cell	
			+	×	+	×	+	×
TA	3	$14\log n + 14$	0	5	$10\log n$	$17\log n$	0	1
GE	4	$16\log n + 16$	0	7	$10\log n$	$17\log n$	0	1
LU	3	$16\log n + 16$	0	7	$10\log n$	$17\log n$	0	1
LV	4	$20\log n + 20$	0	8	$16\log n$	$22\log n$	0	0
RD	4	$18\log n + 18$	0	8	$10\log n$	$18\log n$	0	0
CR	$4(h-1)$	$4(h-1)(2h+5)$	$6(h-1)$	$19(h-1)+1$	0	0	0	0
BA	$4(h-1)$	$4(h-1)(2h+5)$	$11(h-1)+1$	$23(h-1)+1$	0	0	0	0

Figure 4.17 Summary of analyses of direct methods. TA=Thomas algorithm, GE=Gaussian elimination, LU=LU decomposition, LV=LU decomposition variant, RD=recursive doubling variant, CR= cyclic reduction, BA=Buneman algorithm, $h=\log(n+1)$.

Summary: Iterative Methods								
Method	Swps	Comm. Time	Parallel Operations					
			L cells		T cells		C cell	
			+	×	+	×	+	×
J	3	$6\log n + 6$	3	2	0	0	0	0
JOR	3	$6\log n + 6$	4	3	0	0	0	0
GS	2	$8\log n + 8$	2	1	$2\log n$	$3\log n$	0	0
SOR	2	$8\log n + 8$	3	2	$2\log n$	$3\log n$	0	0
RBS	5	$10\log n + 10$	7	6	0	0	0	0
PG	2	$4\log n + 4$	1	1	0	0	0	0

Figure 4.18 Summary of analyses of iterative methods. J=Jacobi, JOR=Jacobi over-relaxation, GS= Gauss-Seidel, SOR=successive over-relaxation, RBS=red-black successive over-relaxation, PG=Parallel Gauss.

Direct Methods				
Method	Algorithm Complexity			
	Sequential	Vector	Array	Tree
TA	n	$[n]$		$\log n$
GE	n	$[n]$		$\log n$
LU	n	n		$\log n$
LV	n	$[n]$		$\log n$
RD	$n \log n$	$n \log n$	$\log n$	$\log n$
CR	n	n	$\log n$	$(\log n)^2$
BA	n	$[n]$	$\log n$	$(\log n)^2$

Figure 4.19 Comparison of the orders of asymptotic complexity of direct tridiagonal linear system solvers for sequential, vector, array, and tree processors. TA=Thomas algorithm, GE=Gaussian elimination, LU=LU decomposition, LV=LU decomposition variant, RD=recursive doubling, CR=cyclic reduction, BA=Buneman algorithm. The results for vector processors were described by Lambiotte and Voigt [LaVo75]. Entries in brackets indicate that the methods were not implemented by Lambiotte and Voigt but the complexity of the algorithms can be inferred from the results obtained from methods that were. The results for array processors were described by Stone [Ston75]. Blank entries mean that implementations of these methods have not been described in the literature.

Iterative Methods			
Method	Algorithm Complexity (per iteration)		
	Sequential	Vector	Tree
J	n	n	$\log n$
JOR	n	$[n]$	$\log n$
GS	n	$[n]$	$\log n$
SOR	n	$[n]$	$\log n$
RBS	n	n	$\log n$
PG	n	n	$\log n$

Figure 4.20 Comparison of the orders of asymptotic complexity of iterative tridiagonal linear system solvers for sequential, vector, and tree processors. J=Jacobi, JOR=Jacobi over-relaxation, GS=Gauss-Seidel, SOR=successive over-relaxation, RBS=red-black successive over-relaxation, PG=Parallel Gauss. The results for vector processors were described by Lambiotte and Voigt [LaVo75]. Entries in brackets indicate inferred orders of complexity (see caption of Figure 4.19).

CHAPTER 5. BLOCK-TRIDIAGONAL LINEAR SYSTEM SOLVERS

A. Overview

The primary objective of this dissertation is the parallel solution on a tree machine of Laplace's equation

$$\frac{\partial^2 z}{\partial x^2} + \frac{\partial^2 z}{\partial y^2} = 0 \tag{5.1}$$

or Poisson's equation

$$\frac{\partial^2 z}{\partial x^2} + \frac{\partial^2 z}{\partial y^2} = r, \quad r \text{ a constant} \tag{5.2}$$

on a rectangular region. The values of the function **z** are known on the boundary of the region and the object is to obtain an approximation of **z** at any point in the interior. The method of finite differences lays a rectangular mesh with n rows and m columns on the region. Equation (5.1) or (5.2) is then replaced by $(n-2)\times(m-2)$ second-order *difference* equations, one for each interior mesh point. The solution of the system of difference equations provides an approximation of **z** at each of the mesh points. The system of equations is block-tridiagonal and may be solved by iterative or direct numerical methods. Figure 5.1 shows a mesh with $n=7$ and $m=9$. The mesh points, both boundary and interior, are numbered from from 0 through $(nm-1)$ in row major order. The corresponding block-tridiagonal system of equations is shown in Figure 5.2.

The iterative methods for solving block-tridiagonal linear systems considered in this chapter are point iterative methods, block iterative methods, and alternating direction implicit methods. Point iterative methods modify the value of one interior mesh point at a time. These methods are called *explicit* methods because the new value at a mesh point is explicitly expressed as a function of already computed approximations of neighboring points. Examples are the Jacobi, Jacobi over-relaxation (JOR), Gauss-Seidel, and successive over-relaxation methods (SOR) methods [Youn71, Ames77]. Another class of iterative methods are block iterative methods (also called methods by lines). At each step, the values of one or more rows of points are modified, typically involving the solution of a tridiagonal system of equations. Such methods are called *implicit* methods because the approximate value of a mesh point is expressed as a component of the solution of a linear system. These methods have been extensively studied [Ames77, DoRa56, FoWa60, Hell77, Hell78, PeRa55, Varg62, Youn71]. Examples are Jacobi by lines, Jacobi over-relaxation by lines, Gauss-Seidel by lines, and SOR by lines. A third class of methods are alternating direction implicit (ADI) methods [PeRa55, DoRa56], which first improve the values of points a row at a time and then a column at a time. As with block iterative methods, the solution of a row or column of points requires the solution of a tridiagonal linear system. This chapter presents tree implementations of these three classes of algorithms.

Direct methods for the solution of sparse *block-tridiagonal* systems have also been widely studied. The initial paper by Hockney [Hock65] described a method based on Fourier transform and cyclic reduction and required $O(n^3)$ arithmetic operations compared with $O(n^4)$ operations required by traditional band matrix methods. A stable version of cyclic reduction was described and

implemented in FORTRAN by Buneman [Bune69]. A further improvement was made using the Fast Fourier Transform [BuGN70]. Subsequent work includes studies of non-rectangular regions [BDGG71, DiFe76, Swar74, SwSw73, Swee73]. Heller [Hell76, Hell77] describes still another variation of cyclic reduction and a general analysis of direct block-elimination methods. Generalized cyclic reduction techniques are described by Sweet [Swee74] and Hageman and Varga [HaVa64]. Other direct methods include an LU decomposition of block-tridiagonal systems described by Varah [Vara72] and a decomposition based on the computation of the eigenvalues of the coefficient matrix described by Buzbee [Buzb75]. Survey papers have been written by Dorr [Dorr70] and Heller [Hell78].

I attempted to develop a tree machine algorithm to implement the Buneman version of cyclic reduction as described by Buzbee, Golub, and Nielson [BuGN70]. Unfortunately, the best algorithm I found was much more complex than the algorithms required to solve the block-tridiagonal system using traditional iterative methods. As a result, I have not included cyclic reduction in this dissertation.

B. Point Iterative Methods

Approximating an elliptic partial differential equation (pde) on a rectangular region using finite differences involves (1) laying a rectangular mesh of points over the region (Figure 5.1), (2) assigning a value, known *a priori*, to each boundary point, (3) replacing the elliptic pde with a system of linear difference equations, and (4) solving the resulting linear system. The boundary values (conditions) remain fixed throughout the process and determine the solution for

the set of interior points. The simplest linear approximation involves an interior point and its four closest neighbors. This produces a linear system of equations whose coefficient matrix has a special block-tridiagonal structure, as shown in Figure 5.2. In this section, we discuss point iterative methods for solving such a block-tridiagonal system.

1. Jacobi

Iterative methods for solving linear systems generally transform the system equation

$$A\mathbf{Z} = \mathbf{R} \tag{5.3}$$

into one more suitable for iteration. If the diagonal elements of A do not vanish, the Jacobi method transforms (5.3) into

$$\mathbf{Z}^{(j+1)} = B\mathbf{Z}^{(j)} + \mathbf{C} \tag{5.4}$$

where

$$B = I - D^{-1}A$$
$$\mathbf{C} = D^{-1}\mathbf{R}$$
$$D = diagonal\ (A)$$
$$j = iteration\ number.$$

When applied to a block-tridiagonal system where the underlying pde is Laplace's equation (5.1), then the ith equation of (5.4) is

$$\mathbf{z}_i^{(j+1)} = (\mathbf{z}_{i-m}^{(j)} + \mathbf{z}_{i-1}^{(j)} + \mathbf{z}_{i+1}^{(j)} + \mathbf{z}_{i+m}^{(j)})/4\ . \tag{5.5}$$

Equation (5.5) shows how the ith interior mesh point must be modified in the $(j+1)$th iteration. Similarly, if the underlying pde is Poisson's equation (5.2), then the ith equation is

$$z_i^{(j+1)} = (z_{i-m}^{(j)} + z_{i-1}^{(j)} + z_{i+1}^{(j)} + z_{i+m}^{(j)} + rh^2)/4 \qquad (5.6)$$

where h is the uniform distance between two adjacent mesh points.

After an initial value is assigned to each z-variable, equation (5.5) or (5.6) is used to obtain new iterates. Equations (5.5) and (5.6) are true even for interior points *adjacent* to the boundary; such points would have boundary points as one or more of their neighbors. The $(j+1)$th approximation of an interior point is determined *entirely* by the jth approximation of its four neighbors. This characteristic of the Jacobi method allows parallelism to be introduced: it is theoretically possible to modify all interior points simultaneously.

To implement this method on a tree machine, we distribute the mesh points (both boundary and interior) among the L cells, one to an L cell and in row-major order. Each L cell holds the following registers

z	the current value of one boundary or interior point (boundary points never change their values),
mask	"0" if boundary point, "1" if interior point,
t	temporary, serves as an accumulator.

Boundary points are distinguished from interior points by a mask in the L cell. Moreover, if an interior point is contained in L_i, its north and south neighbors lie in L_{i-m} and L_{i+m} and its west and east neighbors lie in L cells L_{i-1} and L_{i+1} (Figure 5.3). This means that two of the point's neighbors are in L cells immediately to its left and to its right. The other two neighbors are in L cells a distance m to its left and to its right, where m is the number of points in one row of the mesh. In one iteration, each L cell must receive the values of these four other L cells, or equivalently, L_i must send its value to each of its four neighbors. One iteration of the Jacobi method proceeds as follows.

(1) Use ROTLA to send the value of $L_{(i+1) \bmod mn}$ to L_i. Each mesh point receives the value of its east neighbor. Each L cell executes $t :=$ *value received*.

(2) Use ROTRA to send the value of $L_{(i-1) \bmod mn}$ to L_i. Each mesh point receives the value of its west neighbor. Each L cell executes $t := t +$ *value received*.

(3) Use GDCA($-m$-shift) to send the value of $L_{(i+m) \bmod mn}$ to L_i, i.e., all L cell values move a distance m to the left. Each mesh point receives the value of its south neighbor. Each L cell executes $t := t +$ *value received*.

(4) Use GDCA(m-shift) to send the value of $L_{(i-m) \bmod mn}$ to L_i, i.e., all L cell values move a distance m to the right. Each mesh point receives the value of its north neighbor. Each L cell executes $t := t +$ *value received*.

(5) An L cell containing an interior mesh point computes a new z-value: $z = t/4$ (equation (5.5)), or $z = (t + rh^2)/4$ (equation (5.6)). An L cell containing a boundary point does nothing with the information it receives.

To analyze the Jacobi method, we let $N = 2^p$, where $p = \lceil \log mn \rceil$. N is the number of L cells in the smallest tree machine that can hold mn points. ROTLA and ROTRA each require $2 \log N + 2$ communication steps (Chapter 3) for a tree with N L cells. Moreover, to perform an m-shift, GDCA initially requires $2 \log N + 3$ communication steps to construct the T cell directories, followed by

$$2m + 2 \log N - 1 \qquad (5.7)$$

communication steps to perform the shift. Because GDCA is used many times, the T cells construct the directories only once at the start of the operation and retain them during the entire process.

Figure 5.4 shows the analysis of one iteration of the Jacobi method applied to Laplace's equation. Steps (1) and (2) use ROTLA and ROTRA respectively; step

(2) includes one addition executed by each L cell involved. In both step (3) and step (4) the L cells execute GDCA to perform an m-shift and perform one addition. In step (5), the L cells execute the final division (by 4) and store the new value in their z-registers. The "sweeps" through the tree during steps (3) and (4) are more complex, and take more time, than the simple sweeps in steps (1) and (2). So, in a sense, using the east and west neighbors is cheap, and using the north and south neighbors is expensive. The total number of communication steps required per iteration is $8m + 12\log N + 8$, or $O(m)$, where m is the number of columns of the mesh.

2. Extensions

The technique used in the tree algorithm for the Jacobi method applies to other problems as well. We introduce the "computation molecule" notation used by Bickley [Bick48]. In this notation, the approximation at the point $z = \mathbf{z}_i$ of Laplace's equation is

$$\frac{\partial^2 z}{\partial x^2} + \frac{\partial^2 z}{\partial y^2} : \begin{bmatrix} & 1 & \\ 1 & -4 & 1 \\ & 1 & \end{bmatrix} / h^2 + O(h^2) . \tag{5.8}$$

Other differential equation approximations are

$$\frac{\partial z}{\partial x} = (\mathbf{z}_{i+1} - \mathbf{z}_{i-1}) / 2h + O(h^2) : (-1\ 0\ 1) / 2h + O(h^2) \tag{5.9}$$

$$\frac{\partial z}{\partial y} = (\mathbf{z}_{i+m} - \mathbf{z}_{i-m}) / 2h + O(h^2) : \begin{bmatrix} -1 \\ 0 \\ 1 \end{bmatrix} / 2h + O(h^2) . \tag{5.10}$$

Similarly,

$$\frac{\partial^2 z}{\partial x^2} = (z_{i-1} - 2z_i + z_{i+1})/h^2 + O(h^2)$$
$$: (1\ -2\ 1)/h^2 + O(h^2) \qquad (5.11)$$

and

$$\frac{\partial^2 z}{\partial x\,\partial y} : \begin{bmatrix} -1 & 0 & 1 \\ 0 & 0 & 0 \\ 1 & 0 & -1 \end{bmatrix} / 2h^2 + O(h^2)\,. \qquad (5.12)$$

More complicated molecules include

$$\frac{\partial^4 z}{\partial x^4} + \frac{\partial^4 z}{\partial y^4} : \begin{bmatrix} & & 1 & & \\ & 2 & -8 & 2 & \\ 1 & -8 & 20 & -8 & 1 \\ & 2 & -8 & 2 & \\ & & 1 & & \end{bmatrix} / h^4 + O(h^2) \qquad (5.13)$$

$$\int\int z\ dx\ dy : \begin{bmatrix} 1 & 4 & 1 \\ 4 & 16 & 4 \\ 1 & 4 & 1 \end{bmatrix} h^2/9 + O(h^6) \qquad (5.14)$$

The right-hand side of (5.14) is Simpson's Rule applied to double integration.

Designing the tree algorithm for any one of these problems is straightforward, once the mesh points have been distributed among the L cells. Each of the molecules describes precisely which neighbors each mesh point must communicate with and what weight to assign to a value received. The proper combination of ROTLA, ROTRA, and GDCA (k-shift) provides the necessary communication. Consider, for example, equation (5.13) with the weights in the molecule labeled as follows

$$\begin{bmatrix} & & a & & \\ & b & c & d & \\ e & f & g & h & i \\ & j & k & l & \\ & & m & & \end{bmatrix}.$$

Let $\mathbf{z}_a$ through $\mathbf{z}_m$ represent the mesh points corresponding to weights a

through m. The communication requirements between the center point $\mathbf{z}_g$ and each of its neighbors are as follows:

$$\begin{array}{ccccc} & & G & & \\ & G & G & G & \\ R & R & \mathbf{z}_g & R & R \\ & G & G & G & \\ & & G & & \end{array}$$

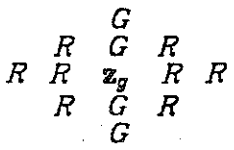
where R represents a ROTRA or ROTLA and G represents a GDCA. Each application of ROTLA and ROTRA requires $2\log N + 2$ communication steps and each application of GDCA (k-shift) (or GDCA ($-k$-shift)), $1 \le k \le \lceil mn/2 \rceil$, requires at most $4k + 4\log N + 2$ communication steps. Thus, we need at most

Value Obtained	*Operation*	*Steps*
$\mathbf{z}_h$, $\mathbf{z}_i$	$2 \times$ ROTLA	$4\log N + 4$
$\mathbf{z}_e$, $\mathbf{z}_f$	$2 \times$ ROTRA	$4\log N + 4$
$\mathbf{z}_b$, $\mathbf{z}_l$	$2 \times$ GDCA($m+1$)	$8(m+1) + 8\log N + 4$
$\mathbf{z}_c$, $\mathbf{z}_k$	$2 \times$ GDCA(m)	$8(m) + 8\log N + 4$
$\mathbf{z}_j$, $\mathbf{z}_d$	$2 \times$ GDCA($m-1$)	$8(m-1) + 8\log N + 4$
$\mathbf{z}_a$, $\mathbf{z}_m$	$2 \times$ GDCA($2m$)	$8(2m) + 8\log N + 4$
	Total	$40m + 40\log N + 24$

communication steps, where m is the number of elements of one row in the mesh (the distance between $\mathbf{z}_c$ and $\mathbf{z}_g$, for example), n is the number of elements in one column, and $N = 2^p$, $p = \lceil \log mn \rceil$.

This straightforward algorithm can be improved. For example, four GDCA's are required to send $\mathbf{z}_a$, $\mathbf{z}_b$, $\mathbf{z}_c$, and $\mathbf{z}_d$ individually to $\mathbf{z}_g$. Instead, we can use GDCA (m) to send $\mathbf{z}_a$, ROTRA to send $\mathbf{z}_b$, and ROTLA to send $\mathbf{z}_d$ to the L cell containing $\mathbf{z}_c$, have this L cell take the weighted average of all four points ($\mathbf{z}_a$, $\mathbf{z}_b$, $\mathbf{z}_c$, and $\mathbf{z}_d$, whose weights are a, b, c, and d, respectively), and use GDCA (m) to send the result to the L cell containing $\mathbf{z}_g$. Similarly, we can use ROTRA, ROTLA, and two applications of GDCA (m) to send the weighted average of points $\mathbf{z}_j$, $\mathbf{z}_k$,

$\mathbf{z}_l$, and $\mathbf{z}_m$ to the L cell containing $\mathbf{z}_g$. We would need a total of only 4 GDCA's and 8 ROTRA's or ROTLA's:



and total communication time would be

Value Obtained	*Operation*	*Steps*
$\mathbf{z}_e$, $\mathbf{z}_f$	2×ROTRA	$4\log N + 4$
$\mathbf{z}_h$, $\mathbf{z}_i$	2×ROTLA	$4\log N + 4$
$\mathbf{z}_a$, $\mathbf{z}_b$, $\mathbf{z}_c$, $\mathbf{z}_d$	ROTRA, ROTLA	$4\log N + 4$
(combined)	2×GDCA(m)	$8(m) + 8\log N + 4$
$\mathbf{z}_j$, $\mathbf{z}_k$, $\mathbf{z}_l$, $\mathbf{z}_m$	ROTRA, ROTLA	$4\log N + 4$
(combined)	2×GDCA(m)	$8(m) + 8\log N + 4$
	Total	$16m + 32\log N + 24$

or a reduction by approximately 60%. Further improvements are possible (note that every L cell receives the values of its east and west neighbors three times) but these improvements would reduce only the $\log N$ term of the equation. No further reduction on the high-order (m) term appears possible.

3. Jacobi Over-relaxation

The Jacobi over-relaxation, or JOR, method is similar to the Jacobi method except that we use a parameter ω to "correct" the new iterate. For example, for Laplace's equation, we use

$$\mathbf{z}_i^{(j+1)} = \omega(\mathbf{z}_{i-m}^{(j)} + \mathbf{z}_{i-1}^{(j)} + \mathbf{z}_{i+1}^{(j)} + \mathbf{z}_{i+m}^{(j)})/4 + (1-\omega)\mathbf{z}_i^{(j)} \qquad (5.15)$$

instead of (5.5) to improve each interior point. The method of choosing the

relaxation parameter ω is described by Young [Youn71]. Note that ω is a constant determined before the first iteration. The first four steps of the tree algorithm for JOR are identical to those of the Jacobi tree algorithm. Step (5) of JOR proceeds as follows.

(5) An L cell containing an interior mesh point replaces the old **z** value with a new one using equation (5.15).

We assume each L cell computes the constant quantities $\omega/4$ and $1-\omega$ once at the start of the process and then retains them. The T cell directories used in GDCA are also determined only once and retained throughout.

The analysis of one iteration of JOR is shown in Figure 5.5. Step 5 of JOR requires an additional two multiplications and one addition of the term $(1-\omega)\mathbf{z}_i^{(j)}$. One iteration of this method requires $O(m)$ time.

4. Gauss-Seidel and Successive Over-relaxation

The Gauss-Seidel and successive over-relaxation (SOR) methods are similar to the Jacobi and JOR methods, respectively, except that the new value $\mathbf{z}_i^{(j+1)}$ is used as soon as it is available. Thus equations (5.5) and (5.15) are replaced by

$$\mathbf{z}_i^{(j+1)} = (\mathbf{z}_{i-m}^{(j+1)} + \mathbf{z}_{i-1}^{(j+1)} + \mathbf{z}_{i+1}^{(j)} + \mathbf{z}_{i+m}^{(j)})/4 \qquad (5.16)$$

and

$$\mathbf{z}_i^{(j+1)} = \omega(\mathbf{z}_{i-m}^{(j+1)} + \mathbf{z}_{i-1}^{(j+1)} + \mathbf{z}_{i+1}^{(j)} + \mathbf{z}_{i+m}^{(j)})/4 + (1-\omega)\mathbf{z}_i^{(j)} \qquad (5.17)$$

respectively.

Mesh points are modified in row-major order with new values used as soon as available. One pass through the entire mesh constitutes one iteration.

Figure 5.6 gives a snapshot of the mesh midway through an iteration (circles represent points that have already been modified). The new value of z_t is currently being obtained. The Gauss-Seidel method is known to have twice the convergence rate of the Jacobi method [Ames77], and, on sequential computer, it is preferable. It is unfortunate, however, that the Gauss-Seidel method lacks the feature that made the Jacobi method attractive for parallel computation, i.e., the possibility of replacing all interior point values simultaneously. Gauss-Seidel seems to be inherently sequential.

After a closer examination, however, we discover that it is possible to introduce some parallelism into the Gauss-Seidel method. Referring once again to Figure 5.6 and the sequential Gauss-Seidel algorithm, we first observe that after z_{10} has been modified, it is possible to modify *both* z_{11} and z_{19}. If we consistently apply the rule that, after modifying z_i, we modify points z_{i+1} and z_{i+m}, we see that instead of improving a single point at a time, we can improve an entire diagonal of points simultaneously (Figure 5.7). We have the image of a wave front, moving from the northwest corner toward the southeast corner, and improving a diagonal values at each time step. One iteration is the time required for the wave to move over the entire mesh.

We also observe that after a diagonal row of points has been modified, it is again ready for improvement in the second time step following. In Figure 5.7, step 3, for example, as we obtain $z_{28}^{(1)}$, $z_{20}^{(1)}$, and $z_{12}^{(1)}$, i.e., the first approximations of z_{28}, z_{20}, and z_{12}, we may also obtain $z_{10}^{(2)}$. In general, after $z_i^{(1)}$ is obtained, it is possible to improve this value at every *other* time step. Figure 5.7, step 11, shows the entire mesh with different points at different stages of improvement. A circle indicates that a mesh point is currently being modified. The point z_{10} is

undergoing its sixth modification while $\mathbf{z}_{52}$, in the lower right hand corner, is going through its first. From now on, at each step, half of the mesh points are modified. One iteration, i.e., the time required to modify the entire set of points, takes two steps. The tree algorithm proceeds as follows.

(0) Startup. At each step, we obtain the first modification of a diagonal of points, starting from the northwest corner ($\mathbf{z}_{10}$) and proceeding toward the southeast corner ($\mathbf{z}_{52}$). After a diagonal of points is modified the first time, it may be modified again at every other step.

After the southeast corner has been modified the first time, every step modifies approximately half the number of mesh points. This involves the following operations.

(1) Use ROTLA to send the value of $L_{(i+1) \bmod mn}$ to L_i. Each mesh point receives the value of its east neighbor. Each L cell executes $t := \textit{value received}$.

(2) Use ROTRA to send the value of $L_{(i-1) \bmod mn}$ to L_i. Each mesh point receives the value of its west neighbor. Each L cell executes $t := t + \textit{value received}$.

(3) Use GDCA($-m$-shift) to send the value of $L_{(i+m) \bmod mn}$ to L_i. Each mesh point to be modified receives the value of its south neighbor. Each L cell executes $t := t + \textit{value received}$.

(4) Use GDCA(m-shift) to send the value of $L_{(i-m) \bmod mn}$ to L_i. Each mesh point to be modified receives the value of its north neighbor. Each L cell executes $t := t + \textit{value received}$.

(5) An L cell containing an interior mesh point computes a new z-value: $\mathbf{z} = t/4$ (equation (5.16)). L cells containing boundary points and L cells containing interior points which are not to be modified in this half-iteration do nothing with the information they receives.

After step 5, approximately half of the points have been modified. Steps 6-10 apply the same algorithm on the rest of the points. The analysis of one iteration of this algorithm, after the startup step, is shown in Figure 5.8. In each half-iteration, the time required to perform a GDCA is approximately half that usually required for an m-shift. This is because only half of the points send values each time, hence only half of the amount of data must pass through the tree branches. One iteration of this method requires $O(m)$ time.

The same technique can be used with the SOR method as easily as with the Gauss-Seidel method. We replace step (5) of the Gauss-Seidel algorithm with

(5) An L cell containing an interior mesh point computes a new z-value: $z = \omega t/4 + (1-\omega)z_i^{(j)}$ (equation (5.17)). L cells containing boundary points and L cells containing interior points that are not to be modified in this half-iteration do nothing with the information they receive.

Step (10) of Gauss-Seidel is similarly modified to obtain step (10) of SOR. The analysis of one iteration of point iterative SOR, after startup, is shown in Figure 5.9. One iteration of this method requires $O(m)$ time. It is interesting to note that in each half-iteration, the points that are modified and the points that are not form a checkerboard pattern on the mesh. This pattern, sometimes called a red-black ordering of the mesh points, is well known and is described in the next section.

5. Red-black Ordering of Mesh Points

Young [Youn71] has found that, in some cases, it is possible to change the order in which mesh points are improved without reducing the convergence rate of the iterative method used. One may, for example, arrange the points into two

groups ("red" and "black") such that if a point is red (black), all of its neighbors are black (red). Figure 5.10 shows how the points are organized, with triangles and circles representing red and black points respectively. Instead of trying to modify the values of all of the points simultaneously, we modify each group of interior points alternately, using either the Gauss-Seidel (5.16) or SOR (5.17) equations. This, in effect, breaks one iteration of the Gauss-Seidel or SOR methods into two half-iterations, each of which improves the values of approximately half the set of points. The higher convergence rate of red-black Gauss-Seidel or red-black SOR [Youn71] over the Jacobi method makes red-black ordering attractive. Lambiotte and Voigt [LaVo75] have reported that the application of red-black SOR to solve tridiagonal linear systems produces the same convergence rate as regular SOR.

Consider equation (5.17) used in straightforward SOR. When applied to a red-black ordering of points, equation (5.17) is broken up into two equations

$$z_i^{(j+1)} = \omega(z_{i-m}^{(j)} + z_{i-1}^{(j)} + z_{i+1}^{(j)} + z_{i+m}^{(j)})/4 + (1-\omega)z_i^{(j)} \qquad (5.18)$$

used for all "red" points z_i, and

$$z_i^{(j+1)} = \omega(z_{i-m}^{(j+1)} + z_{i-1}^{(j+1)} + z_{i+1}^{(j+1)} + z_{i+m}^{(j+1)})/4 + (1-\omega)z_i^{(j)} \qquad (5.19)$$

used for all "black" points z_i. Note that the variables z_{i-m}, z_{i-1}, z_{i+1}, and z_{i+m} in equation (5.18) are black points, whereas the same variables in equation (5.19) are all red points. In each half-iteration, all of the data required to compute new values are known and the only problem is communication. The communication requirements, in fact, are exactly those required by the Jacobi method, except that only half of the points are sent through the tree each time. Each L cell has a mask to distinguish red points from black points.

The analysis of one iteration of this method is identical to that of point iterative SOR (Figure 5.9). The total time to perform the GDCAs is approximately the same as required in one iteration of the Jacobi method. Each iteration of red-black SOR, however, requires twice the number of ROTLA's and ROTRA's.

C. Block iterative Methods

Section B presented methods that, when implemented on a sequential computer, modify a single point at a time. This is possible because the value at any point is expressed by an *explicit* equation involving the point's neighbors. Point iterative methods can be extended naturally to *block iterative methods* (also called *line methods*, *group methods*, or *implicit methods)* which modify a block, line, or group of points at a time. This usually means that we must solve a system of equations for each block. Ames [Ames77] reports that the redefinition of an explicit method into an implicit method often improves convergence at the expense of more computation per iterative step.

In this section, we take each of the point iterative methods described in Section B and present its block iterative counterpart. We define a block to be a row of points. The problem we examine is the solution of Laplace's equation on a rectangular region using the five-point computation molecule.

1. Line Jacobi

This method represents each of the points in a row by the following equation

$$z_{i-1}^{(j+1)} - 4z_i^{(j+1)} + z_{i+1}^{(j+1)} = -z_{i-m}^{(j)} - z_{i+m}^{(j)} \qquad (5.20)$$

where z_i represents an interior mesh point. The equation involves the $(j+1)$th approximation of the points z_i, z_{i-1}, and z_{i+1} and the jth approximation of z_{i-m} and z_{i+m} (the two latter values are assumed to be known). Each equation involves at most three consecutively indexed unknowns (points adjacent to the left or right boundaries involve only two unknowns). The equations corresponding to a row of points form a tridiagonal linear system. If each row is represented this way, we obtain n independent tridiagonal systems. Moreover, because the mesh points are initially distributed among the L cells in row major order, a row of points occupies consecutive L cells, except possibly for interspersed empty L cells. Different tridiagonal systems, therefore, occupy non-overlapping sequences of L cells and the solutions of all of these systems may be obtained simultaneously using one of the direct tridiagonal system solvers described in Chapter 3. The algorithm is, in fact, simpler as the tree machine need not distinguish between different tridiagonal systems. The n tridiagonal systems do not overlap and thus may be considered one large system. The coefficients of the first and last mesh points of a row prevent one system from interfering with another. The ability to make this simplifying assumption on a tree machine is discussed in Section 4.D.

As with the point iterative Jacobi method, we assume the following. Boundary points are distinguished from interior points by a mask in the L cell. If an interior point is contained in L_i, then its north and south neighbors lie in L_{i-m} and L_{i+m} and its west and east neighbors lie in L_{i-1} and L_{i+1}. Moreover, each L cell contains the registers a, b, and c in which we store the coefficients of one tridiagonal equation. Let L_i contain z_i. The following algorithm determines the values of a, b, and c for different mesh points.

if z_i is a boundary point then $a=b=c=0$		
else if z_{i-1} is a boundary point then $a=0,\ b=-4,\ c=1$		z_i's west neighbor is a boundary point
else if z_{i+1} is a boundary point then $a=1,\ b=-4,\ c=0$		z_i's east neighbor is a boundary point
else	$a=1,\ b=-4,\ c=1$	neither of z_i's east or west neighbors is a boundary point

The tree algorithm proceeds as follows:

(1) Use GDCA(m-shift) to send the value of $L_{(i-m) \bmod mn}$ to L_i.

(2) Use GDCA($-m$-shift) to send the value of $L_{(i+m) \bmod mn}$ to L_i.

The sum of the values obtained in steps (1) and (2) determine the constant term of each linear equation. Each L cell now has all of the information it needs to construct the linear equation (5.20) representing one interior mesh point. We are ready to solve the n tridiagonal linear systems.

(3) Let all L cells participate in a single application of one of the direct methods for solving a tridiagonal linear system (Chapter 4). The n tridiagonal systems are considered one.

An iteration ends with step (3). The analysis of the line Jacobi method is shown in Figure 5.11. One iteration of this method requires $O(m)$ time.

2. Line Jacobi Over-relaxation

Line JOR is similar to line Jacobi except that we introduce a relaxation parameter ω and use

$$z_i^{(j+1)} = \omega \bar{z}_i^{(j+1)} + (1-\omega) z_i^{(j)} \qquad (5.21)$$

where $\bar{z}_i^{(j+1)}$ is the value of z_i obtained by applying equation (5.20). After step (3) of the line Jacobi method, therefore, the line JOR tree algorithm uses equation (5.21) to compute $z_i^{(j+1)}$. The analysis of line JOR is shown in Figure 5.12. One iteration of this method requires $O(m)$ time.

3. Line Gauss-Seidel

Just as point iterative Gauss-Seidel is more difficult to implement on a parallel processor than point iterative Jacobi, so also line Gauss-Seidel is more difficult to implement than line Jacobi. On a sequential machine, line Gauss-Seidel modifies a row of points at a time by solving a tridiagonal linear system of equations. Unlike line Jacobi, the new row values are immediately used to solve the next row's values. An iteration is complete when all rows have been modified.

To obtain a new approximation of a row, each row interior point z_i is represented by the equation

$$4z_i^{(j+1)} = z_{i-1}^{(j+1)} + z_{i+1}^{(j+1)} + z_{i-m}^{(j+1)} + z_{i+m}^{(j)} \qquad (5.22)$$

The known values $z_{i-m}^{(j+1)}$ and $z_{i+m}^{(j)}$ combine to form the constant term of each equation. The unknowns are $z_i^{(j+1)}$, $z_{i-1}^{(j+1)}$ and $z_{i+1}^{(j+1)}$. Equations representing points adjacent to the left and right boundaries involve only two variables.

Figure 5.13 gives snapshots of a possible parallel implementation of line Gauss-Seidel.

(1) The first step modifies the first row of points. This involves the parallel solution of a single tridiagonal linear system. We may use one of the direct

methods for solving tridiagonal systems described in Chapter 4.

(2) The second step modifies the second row of points. Again, this involves the solution of a single tridiagonal linear system.

(3) The ith step, $i=3, 4, \ldots, n$ where n is the number of rows of interior mesh points, modifies the ith row of points. In addition, we may also modify the jth row of points where $j=i-2, i-4, \ldots, 2$ *or* 1. This means that after the ith row of points is modified the first time, it may be modified again at every other time step. As i increases, therefore, so does the amount of parallel activity.

After the last row of the mesh is modified the first time, then at every step, approximately half of the rows are modified, i.e., we alternately modify odd-indexed and even-indexed rows. Note that the parallel implementation of point iterative Gauss-Seidel or red-black SOR produced a checkerboard pattern on the mesh whereas with line Gauss-Seidel, a striped pattern is produced. The ith iteration ($i > n$ = the number of mesh rows) of the tree implementation of line Gauss-Seidel proceeds as follows.

Modify odd-indexed rows.

(1) Use GDCA(m-shift) to send the value of $L_{(i-m) \bmod mn}$ to L_i. Only points on even-indexed rows send a value.

(2) Use GDCA($-m$-shift) to send the value of $L_{(i+m) \bmod mn}$ to L_i. Only points on even-indexed rows send a value.

The sum of the values obtained in steps (1) and (2) determine the constant term of each linear equation.

(3) All L cells whose mesh points line on odd-indexed rows participate in the solution of a single tridiagonal system. As with line Jacobi, the coefficients

of the end interior points of a row prevent the different tridiagonal systems from interfering with each other.

Modify even-indexed rows.

(4) Use GDCA(m-shift) to send the value of $L_{(i-m) \bmod mn}$ to L_i. Only points on odd-indexed rows send a value.

(5) Use GDCA($-m$-shift) to send the value of $L_{(i+m) \bmod mn}$ to L_i. Only points on odd-indexed rows send a value.

The sum of the values obtained in steps (4) and (5) determine the constant term of each linear equation.

(6) All L cells whose mesh points line on even-indexed rows participate in the solution of a single tridiagonal system.

The analysis of line Gauss-Seidel is shown in Figure 5.14. One iteration of this method requires $O(m)$ time.

4. Line Successive Over-relaxation

The technique used in line Gauss-Seidel may be used to solve the block-tridiagonal system using line SOR. Instead of equation (5.22), we use

$$\mathbf{z}_i^{(j+1)} = \omega \bar{\mathbf{z}}_i^{(j+1)} + (1-\omega)\mathbf{z}_i^{(j)} \tag{5.23}$$

where $\bar{\mathbf{z}}_i^{(j+1)}$ is the value of $\mathbf{z}_i$ obtained b applying equation (5.22) and ω is the relaxation factor. The algorithm for line SOR is similar to that for line Gauss-Seidel except that equation (5.23) is evaluated after steps 3 and 6. The analysis line SOR is shown in Figure 5.15. Steps 3a and 6a (during which equation (5.23) is evaluated) each require two multiplications and one addition. One iteration of this method requires $O(m)$ time.

D. Alternating Direction Implicit Method

A method by Peaceman and Rachford [PeRa55] and a related method by Douglas and Rachford [DoRa56] improve on block iterative methods by modifying rows of points in one half-iteration and columns of points in the next half-iteration. Ames [Ames77] reports that this often produces better convergence than straightforward block iterative methods which modify points a row at a time. He states, however, that a "rational explanation ... of the effectiveness of ADI methods is still lacking." A survey report on ADI methods was presented by Birkhoff, Varga and Young [BiVY62]. The following description is based on the study by Ames [Ames77].

Basically, ADI methods perform a single row iteration on the mesh followed by a single column iteration. To perform a row iteration, we represent each row interior mesh point by

$$z_i^{(j+1/2)} = z_i^{(j)} + \rho_j [z_{i-1}^{(j+1/2)} + z_{i+1}^{(j+1/2)} - 2z_i^{(j+1/2)}] + \rho_j [z_{i-m}^{(j)} + z_{i+m}^{(j)} - 2z_i^{(j)}] \quad (5.24)$$

where ρ_j is called the iteration parameter which may vary with j. This defines a tridiagonal linear system whose solution produces a row of new values. After each row has been modified in this manner, a column iteration is performed. The interior points of a column are each represented by the equation

$$z_i^{(j+1)} = z_i^{(j+1/2)} + \rho_j [z_{i-1}^{(j+1/2)} + z_{i+1}^{(j+1/2)} - 2z_i^{(j+1/2)}] + \rho_j [z_{i-m}^{(j+1)} + z_{i+m}^{(j+1)} - 2z_i^{(j+1)}] \quad (5.25)$$

This also defines a tridiagonal system whose solution produces a column of new values. A modification of each row followed by a modification of each column constitutes one complete iteration.

As before, the mesh points are distributed among the L cells in row major order. Consider a mesh with n rows, numbered 0 through $n-1$, and m columns,

numbered 0 through $m-1$. The points belonging row i occupy L cells with indices im, $im+1$, $im+2$, ..., $im+(m-1)$. On the other hand, points belonging to column i occupy L cells with indices $i, i+m, i+2m, \ldots, i+(n-1)m$. Because of the decision to distribute the points in row major order, rows of points are contained in non-overlapping sequences of L cells. Columns of points, however, are contained in overlapping L cell sequences. This allows the tree machine to perform row iterations much faster than column iterations.

To implement a *row iteration*, we first observe that the tridiagonal linear systems produced by equation (5.24) are uncoupled, i.e., the solution of the $j+1/2$ approximation of row i depends only on the jth approximation of the rows $i-1$ and $i+1$. (Contrast this to line Gauss-Seidel and line SOR in which the $(j+1)$th approximation of row i depends on the $(j+1)$th approximation of row $i-1$ as well as the jth approximation of row $i+1$.) This means that we can solve all row tridiagonal systems simultaneously. Moreover, because the tridiagonal systems are uncoupled, we may consider all of the tridiagonal systems as forming a single tridiagonal system (discussed in detail in Section 4.D). A single application of one of the tridiagonal system solvers described in Chapter 4 on the entire tree solves all of the tridiagonal systems involved simultaneously. The tree algorithm for a single row iteration proceeds as follows.

(1) Use GDCA($-m$) to send the value of $L_{(i+m) \bmod mn}$ to L_i. L_i receives the value $\mathbf{z}_{i+m}^{(j)}$.

(2) Use GDCA(m) to send the value of $L_{(i-m) \bmod mn}$ to L_i. L_i receives the value $\mathbf{z}_{i-m}^{(j)}$.

(3) Each L cell computes the constant factor of equation (5.25):

$$z_i^{(j)}+\rho_j[z_{i-m}^{(j)}+z_{i+m}^{(j)}-2z_i^{(j)}]$$

and sets up the coefficients of the linear equation corresponding to point z_i.

(4) Apply one of the direct tridiagonal linear system solvers described in Chapter 4. At the end of step 4, the L_i contains $z_i^{(j+1/2)}$ defined by equation (5.24).

Column iterations are slower than row iterations. The reason is that columns of points, and hence the corresponding tridiagonal systems, lie in overlapping sequences of L cells. Although we can obtain new iterates for a single column of points as easily as we can for a single row of points (by masking all other columns out), we cannot solve tridiagonal systems corresponding to two or more columns simultaneously, because their computations would interfere with each other. We can, however, separate these computations in time, and solve the tridiagonal systems of the columns one after another. The tridiagonal system of one column may be solved in parallel in $O(\log n)$ time. As there are $m-2$ columns of interior points, a column iteration requires $O(m \log n)$ time. The tree algorithm for a complete column iteration proceeds as follows.

(5) Use ROTLA to send the value of $L_{(i+1) \bmod mn}$ to L_i. L_i receives the value $z_{i+1}^{(j+1/2)}$

(6) Use ROTRA to send the value of $L_{(i-1) \bmod mn}$ to L_i. L_i receives the value $z_{i-1}^{(j+1/2)}$

(7) Each L cell computes the constant factor of equation (5.25):

$$z_i^{(j+1/2)}+\rho_j[z_{i-1}^{(j+1/2)}+z_{i+1}^{(j+1/2)}-2z_i^{(j+1/2)}]$$

At this point, the L cells are ready to solve the (column) tridiagonal systems defined by equation (5.25). We must, however, solve one tridiagonal system at a time. I.e., step 8 is a loop over the columns of the mesh.

(8) For each column, use one of the direct methods described in in Chapter 4 to solve a column tridiagonal linear system. At the end of step 8, L_i holds $\mathbf{z}_i^{(j+1)}$ defined by equation (5.25).

The analysis of the tree implementation of the ADI method is shown in Figure 5.16. Row iterations and column iterations are carried out in steps 1-4 and steps 5-8, respectively. A row iteration requires $O(m)$ time because the solution of all of the tridiagonal systems (step 4) can be done simultaneously. A column iteration, however, requires $O(m \log n)$ time because each of the m tridiagonal systems must be solved in turn. One complete iteration of the ADI method, therefore, requires $O(m \log n)$ time.

E. Remarks

Figure 5.17 shows the convergence rates of several point iterative and block iterative methods, as described by Ames [Ames77]. Table entries give the value R; the number of iterations required for convergence is inversely proportional to R.

Figure 5.18 gives a summary of the analyses of the point iterative, block iterative, and ADI methods discussed. On a sequential computer, a block iterative method converges faster than the corresponding point iterative method (e.g. block Jacobi converges faster than point Jacobi) but involves more computation. On a tree machine, block iterative methods also involve more computation, but the total computation time is still significantly less than the communication time. For a large mesh (m and n large), the significance of the increased computation time diminishes, i.e., block iterative methods may not require significantly more time than point iterative methods. In summary, we can say

that on a tree machine, a block iterative method looks more attractive compared to the corresponding point iterative method than on a sequential computer.

F. Detailed Time Analysis of the Jacobi Method

This section presents a detailed analysis of the Jacobi method for solving Laplace's equation (5.6) implemented on a tree machine, and compares the performance of the tree machine with that of a sequential computer. We assume that the tree machine has the following characteristics.

(1) The (DEST#, VALUE) pairs sent through the tree during an application of the GDCA algorithm are 84 bits long, a 20-bit DEST# and 64-bit floating-point VALUE.

(2) Communication is two-way, and the channels connecting a cell to its father or sons are k bits wide, where $64/k$ is an integer.

(3) The tree has 2^{20} L cells, thus allowing the use of a mesh with 2^{20} mesh points. For the sake of simplicity, we assume a square mesh with 2^{10} rows and 2^{10} columns.

On a sequential computer, each iteration of the Jacobi method requires the following operations to be performed for each interior mesh point.

(1)-(3) Add the values of the a point's four other neighbors.

(4) Divide the sum by 4, produced by a shift operation.

For the sake of simplicity, we assume that each floating-point operation requires the same amount of time. The number of interior mesh points is

$(2^{10}-2)(2^{10}-2)$. One iteration of the Jacobi method, therefore, requires

$$4(2^{10}-2)^2 \approx 4 \text{ million} \tag{5.26}$$

floating-point operations per iteration.

The tree machine requires five operations: (1) ROTLA, (2) ROTRA, (3)-(4) two applications of GDCA, and (5) one shift operation to perform the division by 4. We assume that the T cell directories required by GDCA are already in place. If the time required for one cell to send one bit to an adjacent cell is τ, then the time required by ROTLA, ROTRA, and GDCA (Chapter 3) are

$$\begin{array}{ll} \text{ROTLA:} & [(2\log N + 2) + (64-k)/k]\tau \\ \text{ROTRA:} & [(2\log N + 2) + (64-k)/k]\tau \\ 2\times GDCA: & [2(4\log N + (2(20+64)m - k)/k)]\tau \\ \hline \text{Total:} & [(8\log N + 4) + (2/k)(168m - 2k + 64)]\tau \end{array} \tag{5.27}$$

where $\log N = 20$, $m = 2^{10}$. ROTLA and ROTRA require the time for one 64-bit floating-point number to go through the C cell and back down to an L cell. Because the channels are k bits wide, the entire number reaches the destination L cell $(64-k)/k$ time units after the first k bits arrive. Each application of GDCA requires (at most) the time for m (DEST#, VALUE) pairs to go through the root T cell and back down to the destination L cells. A (DEST#, VALUE) pair is 84 bits long. Since a sequential computer requires approximately 4 million operations, the performance of the tree machine in millions of floating-point operations per second (*MFLOPS*) is approximately

$$(number\ of\ arithmetic\ operations)\ /\ (time\ on\ a\ tree\ machine) =$$

$$\frac{(4\times 10^6)}{[(8\log N + 4) + (2/k)(168m - 2k + 64)]\tau} \approx 1.16\times 10^{-5}(k/\tau)\ MFLOPS.$$

Figure 5.19 shows the performance of a tree machine executing one iteration of

the Jacobi algorithm for $k = 1, 2, 4$ and $\tau = 40, 60, 80, 100$ *nanoseconds*.

The performance of a tree machine is even better if we use red-black SOR (5.18-19) to solve the problem. Since communication requirements on a tree machine remain approximately the same but the number of operations on a sequential computer increases from approximately 4 *million* to approximately 6 *million* performance increases by approximately 50%.

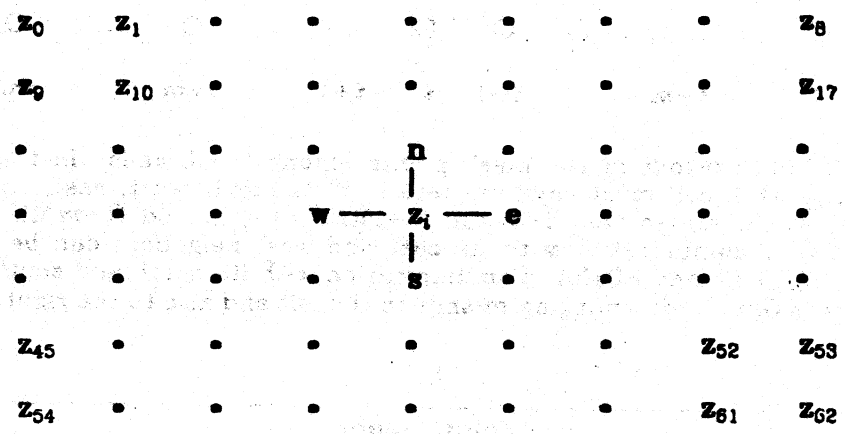


Figure 5.1 The rectangular mesh used by the method of finite differences with n=7 rows and m=9 columns. The points are numbered from z_0 through z_{62} in row major order. If an *interior* point is numbered i, then its north, west, east, and south neighbors are numbered $(i-m)$, $(i-1)$, $(i+1)$, and $(i+m)$ respectively. The block-tridiagonal linear system formed from this mesh will have $(n-2)(m-2)$=35 equations, one equation corresponding each interior point.

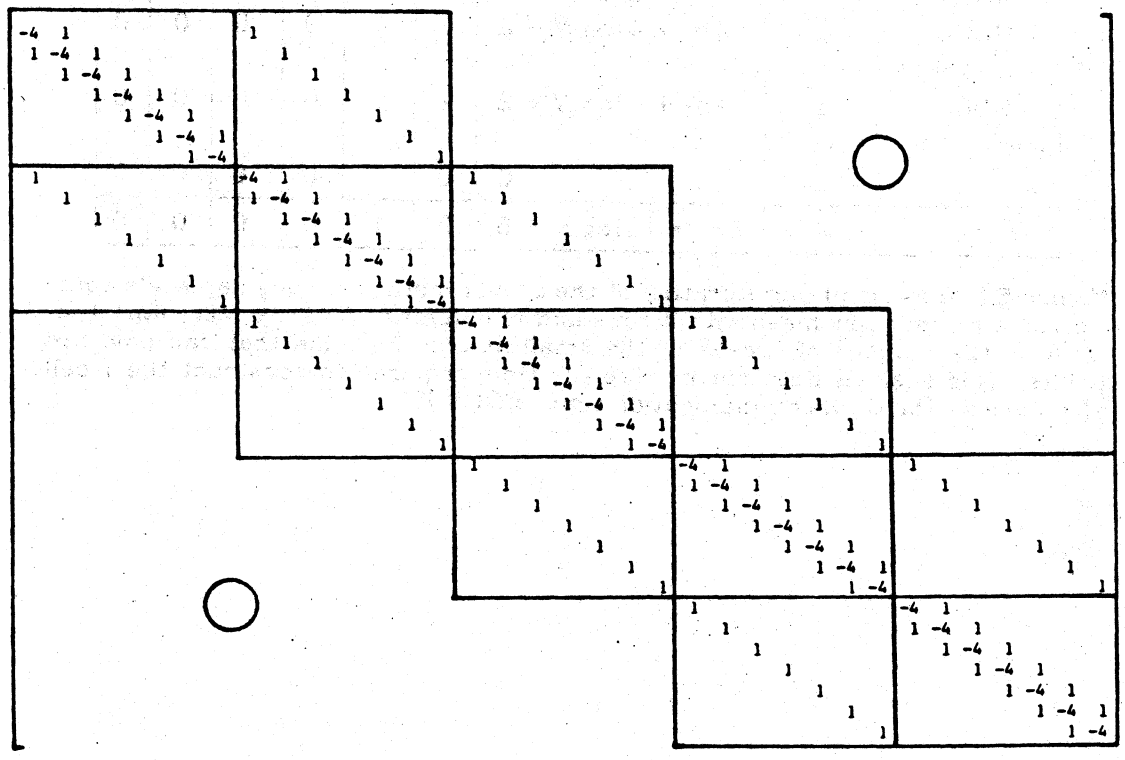


Figure 5.2 The coefficient matrix of the (35×35) block-tridiagonal linear system produced by the rectangular mesh in Figure 5.1. The solution of each interior mesh point is represented by one row of the matrix, that is, as a linear equation involving the point's four closest neighbors. Note the first five and last five equations of the system. These equations represent interior points adjacent to the boundary of the region; consequently, they involve fewer than five variables.

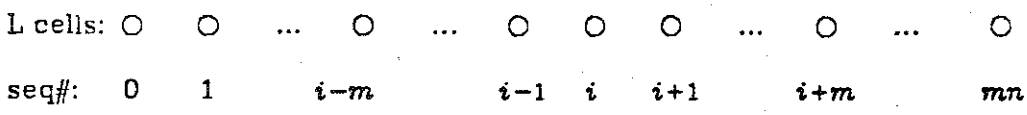


Figure 5.3 Initial layout of the mesh points among the L cells. In the Jacobi method, the ith L cell must communicate with its north, west, east, and south neighbors, which are in the $(i-m)$th, $(i-1)$th, $(i+1)$th, and $(i+m)$th L cells, respectively. Communication with its east and west neighbors can be accomplished with ROTLA and ROTRA. Communication with its north and south neighbors requires GDCA performing an m-shift to the left and also to the right.

Point Jacobi								
Step	Swps	Comm. Time	Parallel Operations					
			L cells		T cells		C cell	
			+	×	+	×	+	×
1. ROTLA	1	$2\log N + 2$	0	0	0	0	0	0
2. ROTRA	1	$2\log N + 2$	1	0	0	0	0	0
3. GDCA ($-m$-shift)	1	$4m + 4\log N + 2$	1	0	0	0	0	0
4. GDCA (m-shift)	1	$4m + 4\log N + 2$	1	0	0	0	0	0
5.	0	0	0	1	0	0	0	0
Total	4	$8m + 12\log N + 8$	3	1	0	0	0	0

Figure 5.4 Analysis of one iteration of the Jacobi method solving Laplace's equation on a rectangular mesh with n rows and m columns. $N=2^p$, $p = \lceil \log mn \rceil$, that is, N is the number of L cells of the smallest tree machine that can hold mn points. This analysis does not include the time required to construct the T cell directories. The total communication time is $O(m)$.

Point Jacobi Over-relaxation								
Step	Swps	Comm. Time	Parallel Operations					
			L cells		T cells		C cell	
			+	×	+	×	+	×
1. ROTLA	1	$2\log N + 2$	0	0	0	0	0	0
2. ROTRA	1	$2\log N + 2$	1	0	0	0	0	0
3. GDCA ($-m$-shift)	1	$4m + 4\log N + 2$	1	0	0	0	0	0
4. GDCA (m-shift)	1	$4m + 4\log N + 2$	1	0	0	0	0	0
5.	0	0	1	2	0	0	0	0
Total	4	$8m + 12\log N + 8$	4	2	0	0	0	0

Figure 5.5 Analysis of one iteration of the JOR method solving Laplace's equation on a rectangular mesh with n rows and m columns. This figure differs from Figure 5.4 only in step 5. JOR requires one more addition and multiplication per iteration than the Jacobi method.

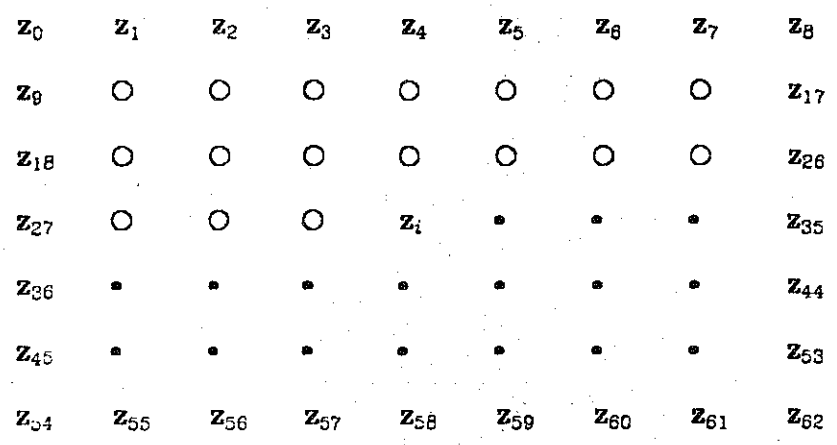
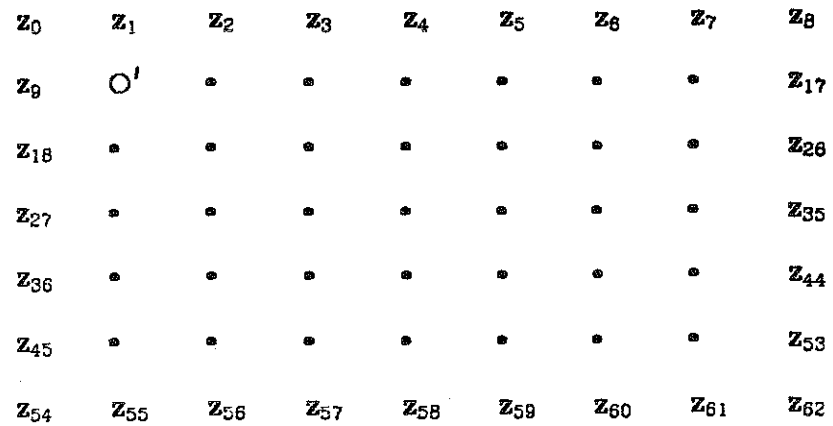
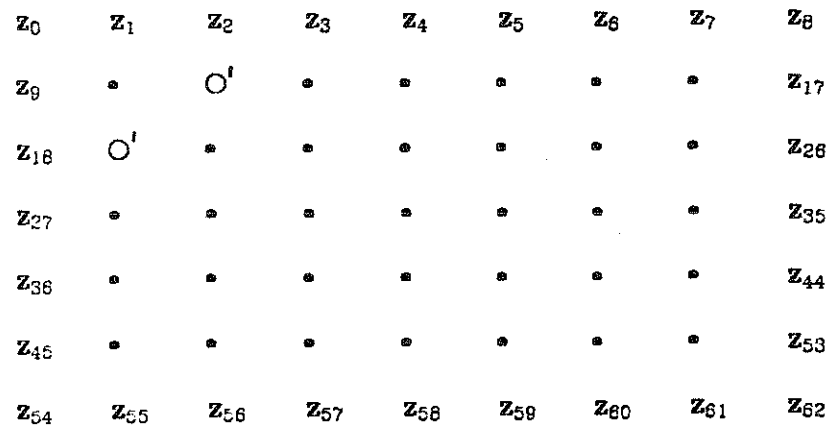


Figure 5.6 Snapshot midway through one iteration of the sequential Gauss-Seidel algorithm. Interior mesh points are modified in row-major order with the new value of a point immediately used to obtain the new value of the next point. A circle indicates that a point has been modified; the next point to be modified is z_{21}. Modifying all of the interior points constitutes one iteration.

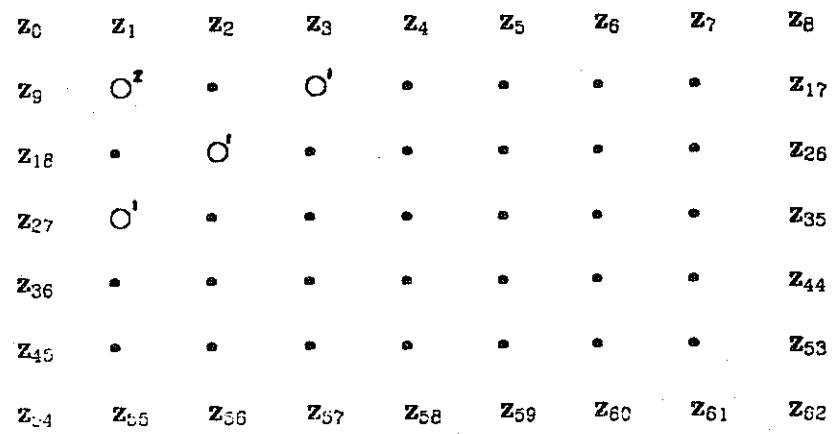
Step 1:



Step 2:



Step 3:



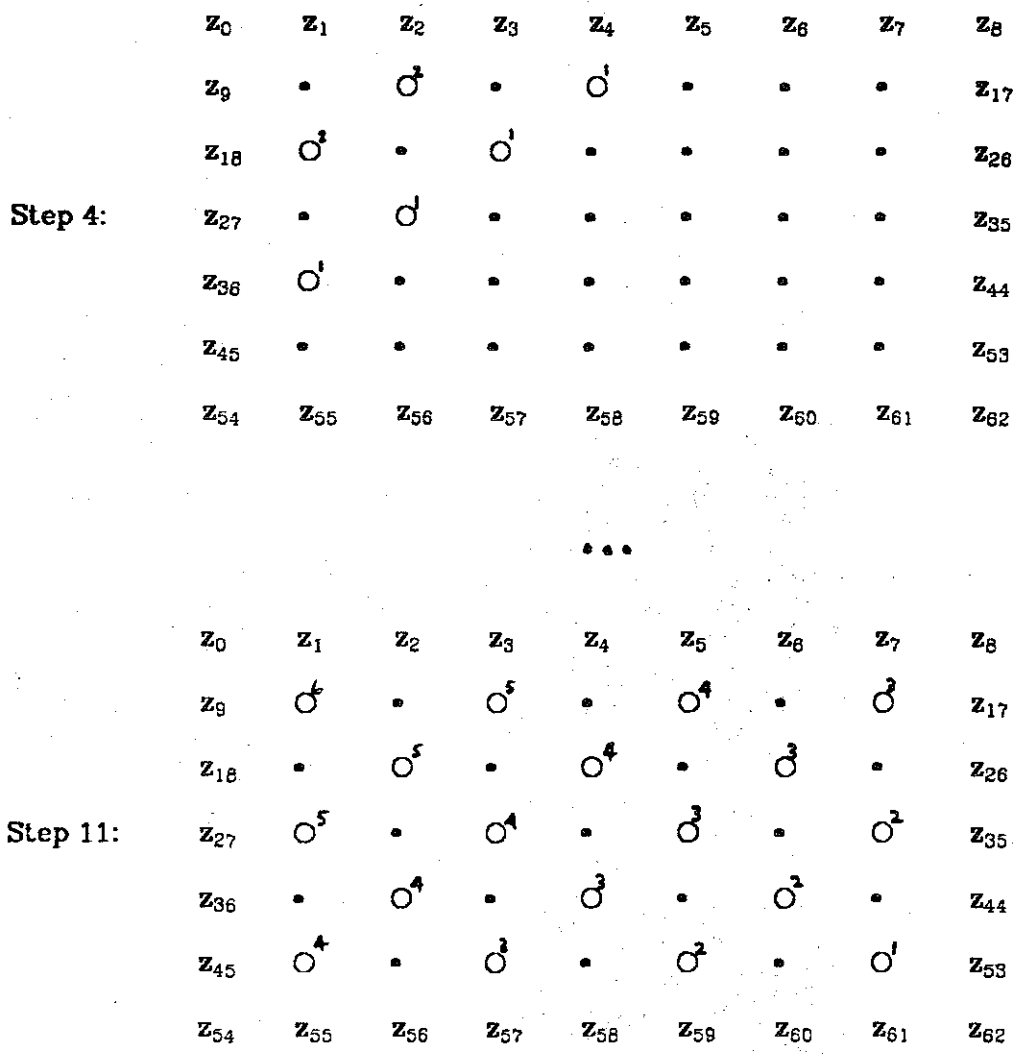


Figure 5.7 Snapshots during the startup period of the parallel Gauss-Seidel algorithm. Diagonals of points are modified each time. In step 1, z_{10} is modified the first time. In step 2, both z_{19} and z_{11} are modified the first time. In step 3, z_{28}, z_{20} and z_{12} are modified the first time and z_{10} is modified the *second* time. In general, after a point has been modified once, it may again be modified after ever other step. Step 11 shows approximately half of the points in different stages of modification: z_{52} the first time, z_{50}, z_{44} and z_{34} the second time, ..., z_{10} the sixth time. Approximately half of the points are modified in each step following step 11 and parallelism is at its peak.

Point Iterative Gauss-Seidel								
Step	Swps	Comm. Time	Parallel Operations					
			L cells		T cells		C cell	
			+	×	+	×	+	×
1. ROTLA	1	$2\log N + 2$	0	0	0	0	0	0
2. ROTRA	1	$2\log N + 2$	1	0	0	0	0	0
3. GDCA ($-m$-shift)	1	$2m + 4\log N + 2$	1	0	0	0	0	0
4. GDCA (m-shift)	1	$2m + 4\log N + 2$	1	0	0	0	0	0
5.	0	0	0	1	0	0	0	0
6. ROTLA	1	$2\log N + 2$	0	0	0	0	0	0
7. ROTRA	1	$2\log N + 2$	1	0	0	0	0	0
8. GDCA ($-m$-shift)	1	$2m + 4\log N + 2$	1	0	0	0	0	0
9. GDCA (m-shift)	1	$2m + 4\log N + 2$	1	0	0	0	0	0
10.	0	0	0	1	0	0	0	0
Total	8	$8m + 24\log N + 16$	6	2	0	0	0	0

Figure 5.8 Analysis of one iteration of point iterative Gauss-Seidel. After the startup period (beginning with step 11 of Figure 5.7, for example), each iteration is achieved in two steps, shown above as substeps 1-5 followed by substeps 6-10. In substeps 1-5, half of the points receive the values of their neighbors and are modified. Substeps 3 and 4 show that the communication time required for GDCA is approximately half that usually required for an m-shift. This is because only half of the L cells send values and the amount of information coursing through the tree branches is half its usual load.

Point Iterative Successive Over-relaxation or Red-Black Successive Over-relaxation								
Step	Swps	Comm. Time	Parallel Operations					
			L cells		T cells		C cell	
			+	×	+	×	+	×
1. ROTLA	1	$2\log N + 2$	0	0	0	0	0	0
2. ROTRA	1	$2\log N + 2$	1	0	0	0	0	0
3. GDCA ($-m$-shift)	1	$2m + 4\log N + 2$	1	0	0	0	0	0
4. GDCA (m-shift)	1	$2m + 4\log N + 2$	1	0	0	0	0	0
5.	0	0	1	2	0	0	0	0
6. ROTLA	1	$2\log N + 2$	0	0	0	0	0	0
7. ROTRA	1	$2\log N + 2$	1	0	0	0	0	0
8. GDCA ($-m$-shift)	1	$2m + 4\log N + 2$	1	0	0	0	0	0
9. GDCA (m-shift)	1	$2m + 4\log N + 2$	1	0	0	0	0	0
10.	0	0	1	2	0	0	0	0
Total	8	$8m + 24\log N + 16$	8	4	0	0	0	0

Figure 5.9 Analysis of one iteration of point iterative successive over-relaxation after the startup period, or one iteration of red-black successive over-relaxation. One iteration consists of two sub-iterations: one which modifies the "triangles" (steps 1-5) the other which modifies the "circles" (steps 6-10). Red-black SOR requires twice as many applications of GDCA per iteration as the Jacobi method but each application takes approximately half the time. Red-black SOR, however, requires twice as many ROTLA's and ROTRA's as the Jacobi method. A *mask* register in each L cell distinguishes triangles from circles. Although the complexity of the algorithm is identical, there is a difference between the two algorithms. Point iterative SOR requires a startup period whereas red-black SOR does not. Consequently, during one iteration of the former, mesh points have different iteration numbers (see Figure 5.7) whereas during one iteration of the latter, points modified all have the same iteration number.

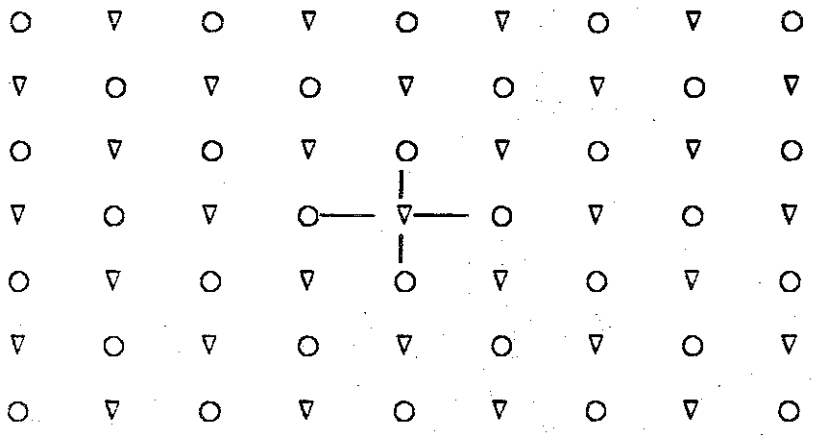


Figure 5.10 To implement red-black successive over-relaxation, mesh points are arranged in a checkerboard (red-black, triangle-circle) pattern. Points of the same type (all triangles or all circles) are modified simultaneously. One iteration, therefore, consists of two sub-iterations: one which modifies the "triangles", the other which modifies the "circles".

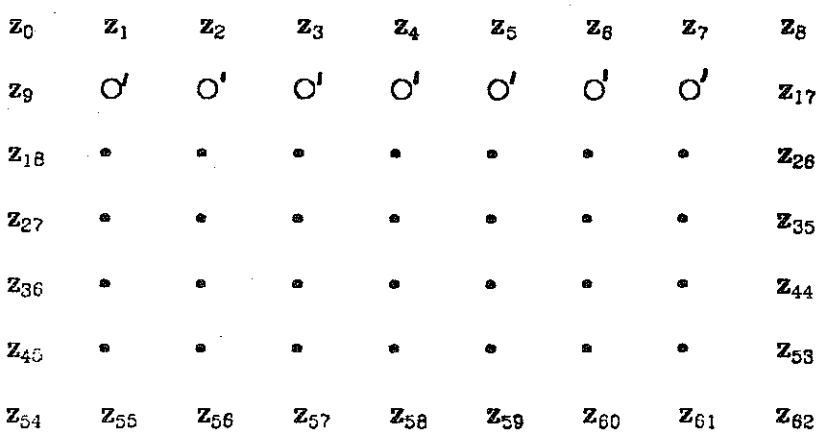
Line Jacobi								
Step	Swps	Comm. Time	Parallel Operations					
			L cells		T cells		C cell	
			+	×	+	×	+	×
1. GDCA ($-m$-shift)	1	$4m + 4\log N + 2$	0	0	0	0	0	0
2. GDCA (m-shift)	1	$4m + 4\log N + 2$	1	0	0	0	0	0
3. TA	3	$14\log N + 14$	0	5	$10\log N$	$17\log N$	0	1
Total	5	$8m + 22\log N + 18$	1	5	$10\log N$	$17\log N$	0	1

Figure 5.11 Analysis of one iteration of the line Jacobi method. In steps 1 and 2, the values of a points north and south neighbors are received by each interior mesh point. Step 2 includes one addition required to determine the constant term of each point's linear equation. In step 3, a single application of the Thomas Algorithm (TA) solves the tridiagonal linear systems simultaneously. The solution of the linear systems provides the new values of the mesh points.

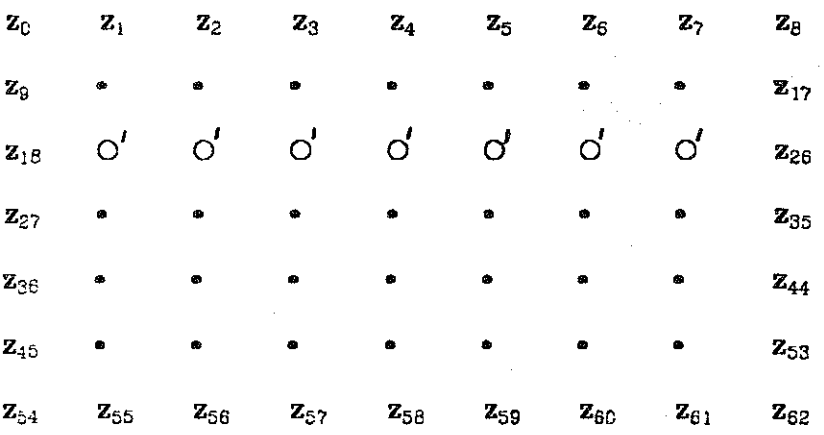
Line Jacobi Over-relaxation								
Step	Swps	Comm. Time	Parallel Operations					
			L cells		T cells		C cell	
			+	×	+	×	+	×
1. GDCA ($-m$-shift)	1	$4m + 4\log N + 2$	0	0	0	0	0	0
2. GDCA (m-shift)	1	$4m + 4\log N + 2$	1	0	0	0	0	0
3. TA	3	$14\log N + 14$	0	5	$10\log N$	$17\log N$	0	1
4.	0	0	1	2	0	0	0	0
Total	5	$8m + 22\log N + 18$	3	8	$10\log N$	$17\log N$	0	1

Figure 5.12 Analysis of one iteration of the line Jacobi over-relaxation method. This method is similar to the line Jacobi method (Figure 5.11) but requires an extra step, step 4, which computes each point's new value using equation (5.13).

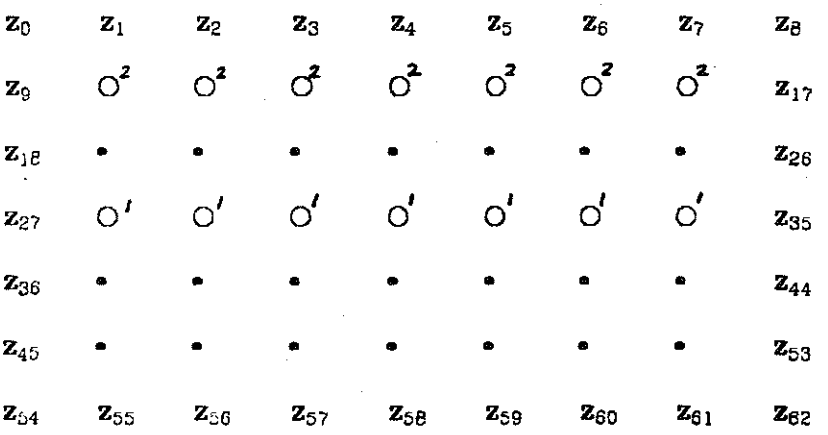
Step 1:



Step 2:



Step 3:



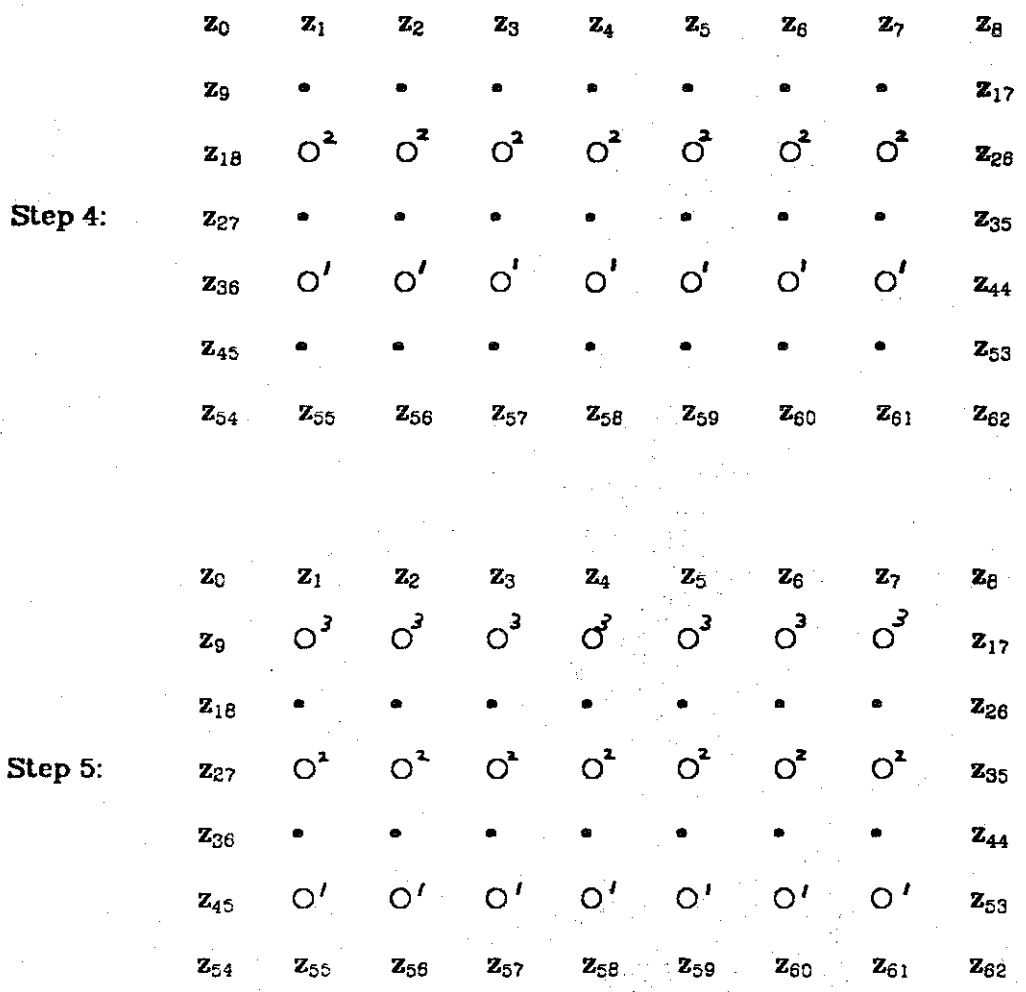


Figure 5.13 Block iterative wave snapshots. Snapshots during the startup period of the line Gauss-Seidel algorithm. Rows of interior points are modified each time. The first and second rows are modified the first time in steps 1 and 2, respectively. In step 3, the third row is modified the first time and the first row is modified the second time. In general, after a row has been modified once, it may again be modified after ever other step. Step 5 shows approximately half of the rows in different stages of modification: the fifth row is modified the first time, the third row the second time, and the first row the third time.

Line Gauss-Seidel								
Step	Swps	Comm. Time	Parallel Operations					
			L cells		T cells		C cell	
			+	×	+	×	+	×
1. GDCA ($-m$-shift)	1	$2m + 4\log N + 2$	0	0	0	0	0	0
2. GDCA (m-shift)	1	$2m + 4\log N + 2$	1	0	0	0	0	0
3. TA	3	$14\log N + 14$	0	5	$10\log N$	$17\log N$	0	1
4. GDCA ($-m$-shift)	1	$2m + 4\log N + 2$	0	0	0	0	0	0
5. GDCA (m-shift)	1	$2m + 4\log N + 2$	1	0	0	0	0	0
6. TA	3	$14\log N + 14$	0	5	$10\log N$	$17\log N$	0	1
Total	10	$8m + 44\log N + 36$	2	10	$20\log N$	$34\log N$	0	2

Figure 5.14 Analysis of one iteration of the line Gauss-Seidel tree algorithm after the startup period. Half of the rows are modified in steps 1-3, the other half in steps 4-6. In steps 1 and 2, the values of a points north and south neighbors are received by each of the interior mesh point to be modified. This requires approximately half the communication time usually required by a GDCA m−shift because only half of the L cells send values thus reducing the amount of information flowing through the tree. Step 2 includes one addition required to determine the constant term of each point's linear equation. In step 3, a single application of the Thomas Algorithm (TA) solves the tridiagonal linear systems simultaneously. The solution of the linear systems provides the new values of the mesh points. Steps 4-6 are analyzed similarly.

Line Successive Over-Relaxation								
Step	Swps	Comm. Time	Parallel Operations					
			L cells		T cells		C cell	
			+	×	+	×	+	×
1. GDCA ($-m$-shift)	1	$2m + 4\log N + 2$	0	0	0	0	0	0
2. GDCA (m-shift)	1	$2m + 4\log N + 2$	1	0	0	0	0	0
3. TA	3	$14\log N + 14$	0	5	$10\log N$	$17\log N$	0	1
3a.	0	0	1	2	0	0	0	0
4. GDCA ($-m$-shift)	1	$2m + 4\log N + 2$	0	0	0	0	0	0
5. GDCA (m-shift)	1	$2m + 4\log N + 2$	1	0	0	0	0	0
6. TA	3	$14\log N + 14$	0	5	$10\log N$	$17\log N$	0	1
6a.	0	0	1	2	0	0	0	0
Total	10	$8m + 44\log N + 36$	4	14	$20\log N$	$34\log N$	0	2

Figure 5.15 Analysis of one iteration of the line successive over-relaxation tree algorithm. Line SOR is similar to line Gauss-Seidel except for the addition of steps 3a and 6a which evaluate equation (5.23).

Alternating Direction Implicit								
Step	Swps	Comm. Time	Parallel Operations					
			L cells		T cells		C cell	
			+	×	+	×	+	×
1. GDCA ($-m$-shift)	1	$4m + 4\log N + 2$	1	0	0	0	0	0
2. GDCA (m-shift)	1	$4m + 4\log N + 2$	1	0	0	0	0	0
3.	0	0	3	2	0	0	0	0
4. TA	3	$14\log N + 14$	0	5	$10\log N$	$17\log N$	0	1
5. ROTLA	1	$2\log N + 2$	0	0	0	0	0	0
6. ROTRA	1	$2\log N + 2$	0	0	0	0	0	0
7.	0	0	3	2	0	0	0	0
8. TA (m *times*)	$3m$	$m(14\log N + 14)$	0	$5m$	$10m\log N$	$17m\log N$	0	m
Total	$3m+7$	$(14\log N+22)m + 26\log N + 22$	0	$5m+10$	$10(m+1)\log N$	$17(m+1)\log N$	0	$m+1$

Figure 5.16 Analysis of one iteration of the ADI method. One iteration is composed of a row iteration (steps 1-4) and a column iteration (steps 5-8). In steps 1 and 2, each point receives the value of its north and south neighbors. Step 3 determines the constant term of each point's linear equation. In step 4, the Thomas algorithm (TA) is applied to solve the tridiagonal linear systems formed by the rows. To perform a column iteration, each point receives the values of its east and west neighbors in steps 5 and 6 and determines the constant term of each point's linear equation. In step 8, the tridiagonal system formed by a column is solved *one at a time*. As there are m columns, step 8 requires m times the computation required for one application of TA. One iteration of the ADI method, therefore, requires $O(m \log N)$ time.

Rates of Convergence			
Molecule	Method	Point Iterative	Block Iterative
5-point	Jacobi	$h^2/2$	h^2
	Gauss-Seidel	h^2	$2h^2$
	SOR	$2h$	$2\sqrt{2}h$
9-point	Jacobi		h^2
	Gauss-Seidel		$2h^2$
	SOR		$2\sqrt{2}h$
5-point (two line)	SOR		$4h$

Figure 5.17 Rates of convergence R of some of the iterative methods discussed [Ames77], where h is the interval width (distance between a mesh point and any of its four neighbors). The number of iterations required for convergence is inversely proportional to R.

Summary: Iterative Methods								
Method	Swps	Comm. Time	Parallel Operations					
			L cells		T cells		C cell	
			+	×	+	×	+	×
PJ	4	$8m + 12\log N + 8$	3	1	0	0	0	0
PJOR	4	$8m + 12\log N + 8$	4	2	0	0	0	0
PGS	8	$8m + 24\log N + 16$	6	2	0	0	0	0
PSOR	8	$8m + 24\log N + 16$	8	4	0	0	0	0
RBSOR	8	$8m + 24\log N + 16$	8	4	0	0	0	0
LJ	5	$8m + 22\log N + 18$	1	5	$10\log N$	$17\log N$	0	1
LJOR	5	$8m + 22\log N + 18$	2	7	$10\log N$	$17\log N$	0	1
LGS	10	$8m + 44\log N + 36$	2	10	$20\log N$	$34\log N$	0	2
LSOR	10	$8m + 44\log N + 36$	4	14	$20\log N$	$34\log N$	0	2
ADI	$3m+7$	$(14\log N+22)m + 26\log N + 22$	0	$5m+10$	$10(m+1)\log N$	$17(m+1)\log N$	0	$m+1$

Figure 5.18 Summary of analyses of iterative methods. PJ=Point Jacobi, PJOR=Point Jacobi over-relaxation, PGS=Point Gauss-Seidel, PSOR=Point Successive Over-relaxation, RBSOR=Red-Black Successive Over-relaxation, LJ=Line Jacobi, LJOR=Line Jacobi over-relaxation, LGS=Line Gauss-Seidel, LSOR=Line Successive Over-relaxation, ADI=Alternating Direction Implicit, m=number of columns, n=number of rows, N=number of L cells in the tree machine.

Performance of the Tree Machine			
k / τ	1	2	4
100 *ns*	116	232	464
80 *ns*	145	290	580
60 *ns*	193	387	774
40 *ns*	290	580	1,160

Figure 5.19 Performance of a tree machine executing one iteration of the Jacobi algorithm, in millions of floating-point operations per second (MFLOPS), for different values of k (1, 2, 4) and τ. k is the width of a channel connecting two tree cells (one way) and τ is the amount of time required to send one bit of information from one cell to another, measured in *nanoseconds* (*ns*).

CHAPTER 6. CONCLUSION

We summarize this dissertation by answering the three questions asked in Chapter 1 and by making several general remarks regarding where one might go from here.

Can solutions to elliptic partial differential equations be implemented efficiently on a tree machine? Chapter 5 gave several iterative tree machine algorithms to solve two dimensional elliptic pde problems. All but one require $O(n)$ time to perform one iteration, the ADI method requires $O(n \log n)$ time. These compare favorably with the $O(n^2)$ time required on a sequential computer. The lower complexity of the tree machine algorithms is achieved by efficiently solving some of the subproblems, such as low-order linear recurrences and $(n \times n)$ tridiagonal linear systems. Linear recurrences (Chapter 3) and tridiagonal systems (Chapter 4) can both be solved in $O(\log n)$ time. In solving the elliptic pde, communication tends to be the costliest part of processing. For example, in an $(n \times n)$ mesh represented in row major order, the total time required for each mesh point to communicate with its north and south neighbors is $O(n)$.

How does the tree machine implementation compare with implementations on other high performance machines, e.g., vector and array processors? Vector processors can provide only a constant speedup over sequential computers. Hence, all of the iterative methods presented in Chapter 5 require $O(n^2)$ time on a vector processor. On the other hand, an array processor, such as the ILLIAC-

IV, can implement any of the point iterative methods in constant, i.e., O(1) time, provided there are enough processing elements to store all of the mesh point values simultaneously. This, of course, is because the interconnection among the processing elements matches the communication requirements of the mesh exactly. Consequently, a mesh point (stored in a processing element) can communicate with any of its four nearest neighbors in constant time. To implement a block iterative algorithm or the ADI method, however, requires the solution of tridiagonal systems. Stone [Ston75] asserts that an $(n \times n)$ tridiagonal system can be solved in O(log n) time on an array processor. A precise description of communication among the processors, however, was not included in Stone's analysis. It appears that communication may require $O(n)$ time for the methods he discusses. If this is the case, then block iterative and ADI methods require $O(n)$ time on an array processor.

Tree algorithms solving recurrence expressions (Chapter 3) and tridiagonal linear systems (Chapter 4) compare well with the same algorithms implemented on vector and array processors. The order of complexity of tree machine algorithms to solve recurrence expressions, O(log n), matches that of array processors. In all tridiagonal linear system methods studied, the tree algorithms are consistently better asymptotically than the same algorithms implemented on a vector processor. In fact, Lambiotte and Voigt [LaVo75] show that some methods (direct methods such as Gaussian elimination and LU decomposition, and iterative methods such as the Gauss-Seidel and successive over-relaxation) *cannot* be implemented efficiently on a vector processor. Moreover, except for two direct tridiagonal system solvers (cyclic reduction and the Buneman algorithm), the tree algorithms match the order of complexity of the best parallel algorithm known for a given problem. This is particularly encouraging because,

in many instances, what is considered the "best" algorithm is designed for an idealized parallel processor assuming, for example, that any two processors can communicate in constant (or even no) time. (Moreover, it should be mentioned that array processors are SIMD machines whereas a tree machine can be an MIMD computer [Mag679a]. It is beyond the scope of this dissertation to argue the potential advantages of tree machines over array processors.)

What conclusions regarding tree machine programming do these implementations provide? Communication among processing elements has emerged as a primary concern in parallel processing. This is emphasized by Gentleman [Gent78] who studied the role of data communication in parallel matrix computations when executed on parallel processors. He showed how communication among processors, rather than arithmetic operations, can play the dominant role in the overall performance of an algorithm. He cautions against algorithm analyses that consider only parallel arithmetic operations as they may be very misleading. (The assumptions in his analysis were such that the conclusions apply to parallel computation in general, and not just matrix operations.)

The tree algorithms ROTLA and GDCA (Chapter 3) have shown that communication among the L cells of a tree machine need not be restricted to sending all L cell values up through the root T cell and back down again, i.e., $O(N)$ time where N is the number of L cells. ROTLA and GDCA can often provide communication in less than linear time. This refutes the often repeated argument that tree machine algorithms always involve a bottleneck at the root. By masking subsets of the L cells, ROTLA and GDCA allow a variety of L cell communication patterns.

Suggestions for Further Work

All methods discussed in this dissertation have been previously developed and analyzed. No attempt was made to develop new numerical algorithms specifically suited for a tree machine. A natural extension of this dissertation would be to develop such methods. Because points are stored among the L cells in row major order, a mesh point is immediately beside its east and west neighbors but a distance m away from its north and south neighbors. Consequently, the total time for all points to communicate with their north and south neighbors is far greater than the total time for all points to communicate with their east and west neighbors. One interesting research topic is the investigation of numerical algorithms that allow a mesh point to communicate with its north and south neighbors less frequently than with its east and west neighbors. Is it possible to collect the values of a point's north and south neighbors less frequently than the values of its east and west neighbors, and still maintain comparable convergence rates? A related question: is it possible to develop methods that make good use of asymmetrical molecules? For example, a mesh point may use the values east of its east neighbor and west of its west neighbor, as well as its four original neighbors. The values of the two new neighbors can be collected inexpensively, using ROTRA and ROTLA. The large difference between the time required by GDCA and by ROTLA motivates further work on these topics.

Another interesting topic is the implementation on a tree machine of a *direct* method of solving block-tridiagonal linear systems. Hockney [Hock65, Hock70] developed a direct method called cyclic reduction, improved by Buneman [Bune69] and Buzbee, Golub and Nielson [BuGN70]. On a sequential computer, the solution of an $(n^2 \times n^2)$ block-tridiagonal linear system requires $O(n^4)$

[ChKu75] CHEN, S.C. and KUCK, D.J. Time and parallel processor bounds for linear recurrence systems. *IEEE Transactions on Computers C-24*, 7 (1975), 701-717.

[ChSa75] CHEN, S.C. and SAMEH, A.H. On parallel triangular system solvers. *Proceedings of the 1975 Sagamore Computer Conference on Parallel Processing*, (1975), 237-238.

[Chen76] CHEN, S.C. Time and parallel processor bounds for linear recurrence systems with constant coefficients. In *Proceedings of the 1976 International Conference on Parallel Processing*, P.H. Enslow Jr. (ed.), 196-205.

[ChKS78] CHEN, S.C., KUCK, D.J. and SAMEH, A.H. Practical parallel band triangular system solvers. *ACM Transactions on Mathematical Software 4*, 3 (1978), 270-277.

[CoGo73] CONCUS, P. and GOLUB, G.H. Use of fast direct methods for the efficient numerical solution of nonseparable elliptic equations. *SIAM Journal on Numerical Analysis 10*, 6 (1973), 1103-1120.

[CoMa67] COVEYOU, R.R. and MACPHERSON, R.D. Fourier analysis of uniform random number generators. *Journal of the ACM 14*, 1 (1967), 100-119.

[Diam75] DIAMOND, M.A. The stability of a parallel algorithm for the solution of tridiagonal linear systems. *Proceedings of the 1975 Sagamore Computer Conference on Parallel Processing*, (1975), 235.

[DiFe76] DIAMOND, M.A. and FERREIRA, D.L. On a cyclic reduction method for the solution of Poisson's equations. *SIAM Journal on Numerical Analysis 13*, 1 (1976), 54-70.

[Dorr70] DORR, F.W. The direct solution of the discrete Poisson equation on a rectangle. *SIAM Review 12*, 2 (1970), 248-263.

[DoRa56] DOUGLAS, J.Jr. and RACHFORD, H.H.Jr. On the numerical solution of heat conduction problems in two and three space variables. *Transactions of the American Mathematical Society 82*, (1956), 421-439.

[Doug62] DOUGLAS Jr., J. Alternating direction methods for three space variables. *Numerische Mathematik 4*, (1962), 41-63.

[FoWa60] FORSYTHE, G.E. and WASOW, W.R. *Finite-difference methods for partial differential equations*, John Wiley & Sons, Inc., New York, 1960.

[Fran79] FRANK, G.A. Virtual memory systems for closed applicative language interpreters. *Ph.D. dissertation*, Department of Computer Science, University of North Carolina at Chapel Hill, 1979.

[Gajs78] GAJSKI, D.D. Processor array for computing linear recurrence systems. In *Proceedings of the 1978 International Conference on Parallel Processing*, Lipovski, G.J. (ed.), Bellaire, MI, 1978, 246-256.

[Gajs81] GAJSKI, D.D. An algorithm for solving linear recurrence systems on parallel and pipelined machines. *IEEE Transactions on Computers C-30*, 4 (1981), 190-206.

[Gent78] GENTLEMAN, W.M. Some complexity results for matrix computations on parallel processors. *Journal of the ACM 25*, 1 (1978), 112-115.

[HaVa64] HAGEMAN, L.A. and VARGA, R.S. Block iterative methods for cyclically reduced matrix equations. *Numerische Mathematik 6*, (1964), 106-119.

[Hell74] HELLER, D.E. A determinant theorem with applications to parallel algorithms. *SIAM Journal on Numerical Analysis 11*, 3 (1974), 559-568.

[Hell76] HELLER, D.E. Some aspects of the cyclic reduction algorithm for block tridiagonal linear systems. *SIAM Journal on Numerical Analysis 13*, 4 (1976), 484-496.

[Hell77] HELLER, D.E. Direct and iterative methods for block-tridiagonal linear systems. *Ph.D. dissertation*, Department of Computer Science, Carnegie-Mellon University, April 1977.

[Hell78] HELLER, D.E. A survey of parallel algorithms in numerical linear algebra. *SIAM Review 20*, 4 (1978), 740-777.

[HeST76] HELLER, D.E., STEVENSON, D.K. and TRAUB, J.F. Accelerated iterative methods for the solution of tridiagonal systems on parallel computers. *Journal of the ACM 23*, 4 (1976), 636-654.

[Hoar78] HOARE, C.A.R. Communicating Sequential Processes. *Communications of the ACM 21*, 8 (1978), 666-677.

[Hock65] HOCKNEY, R.W. A fast direct solution of Poisson's equation using Fourier analysis. *Journal of the ACM 12*, 1 (1965), 95-113.

[Hock70] HOCKNEY, R.W. The potential calculation and some examples. *Methods in Computational Physics 9*, Adler, B., Fernback, S. and Rotenberg, M. (eds.), (1970), 135-211.

[HyKu77] HYAFIL, L. and KUNG, H.T. The complexity of parallel evaluation of linear recurrences. *Journal of the ACM 24*, 3 (1977), 513-521.

[IrHe80] IRWIN, M.J. and HELLER, D.E. On-line pipeline systems for recursive numeric computations. *SIGARCH Newsletter 8*, 3 (1980), 292-299.

[JeHo79] JESSHOPE, C.R. and HOCKNEY, R.W. (eds.) *Super-computers: Infotech State of the Art Report, Volumes 1 and 2.* Infotech International Limited, Maidenhead, Berkshire, England, 1979.

[KaMW67] KARP, R.M., MILLER, R.E. and WINOGRAD, S. The organization of computations for uniform recurrence equations. *Journal of the ACM 14*, 3 (1967), 563-590.

[KoSt73] KOGGE, P.M. and STONE, H.S. A parallel algorithm for the efficient solution of a general class of recurrence equations. *IEEE Transactions on Computers C-22*, 8 (1973), 786-793.

[Kogg73] KOGGE, P.M. Maximal rate pipelined solutions to recurrence problems. In *Proceedings of the First Annual Symposium on Computer Architecture*, Lipovski, G.J. and Szygenda, S.A. (eds.), Gainesville, FL, 1973, 71-76.

[Kogg74] KOGGE, P.M. Parallel solution of recurrence problems. *IBM Journal of Research and Development 18*, 2 (1974), 138-148.

[Kost77] KOSTER, A. Execution time and storage requirements of reduction language programs on a reduction machine. *Ph.D. dissertation*, Department of Computer Science, University of North Carolina at Chapel Hill, 1977.

[Kuck75] KUCK, D.J. Parallel processing architecture - a survey. In *Proceedings of the 1975 Sagamore Computer Conference on Parallel Processing*, (1975), 15-39.

[Kuck76] KUCK, D.J. On the speedup and cost of parallel computation. In *The Complexity of Computational Problem Solving*, R.S. Anderssen and R.P. Brent (eds.), University of Queensland Press, St. Lucia, Queensland, 1976, 63-78.

[Kung76] KUNG, H.T. New algorithms and lower bounds for the parallel evaluation of certain rational expressions and recurrences. *Journal of the ACM 23*, 2 (1976), 252-261.

[Kung79] KUNG, H.T. Let's design algorithms for VLSI systems. *Report No. CMU-CS-79-151*, Carnegie-Mellon University, Pittsburg, PA, 1979.

[Kung80] KUNG, H.T. The structure of parallel algorithms. In *Advances in Computers 19*, H.T. Kung (ed.), Academic Press, New York, 1980, 65-112.

[LaVo75] LAMBIOTTE, J.J. and VOIGT, R.G. The solution of tridiagonal linear systems on the CDC STAR-100 computer. *ACM Transactions on Mathematical Software 1*, 4 (1975), 308-329.

[Leis79] LEISERSON, C.E. Systolic priority queues. *Proceedings of the Caltech Conference on VLSI*, (January 1979), 199-214.

[MaRo77] MADSEN, N.K. and RODRIGUE, G.H. Odd-even reduction for pentadiagonal matrices. In *Parallel Computers - Parallel Mathematics*, M. Feilmeier (ed.), North-Holland Publishing Co., Amsterdam (1977), 103-106.

[Mag679a] MAGO, G.A. A network of microprocessors to execute reduction languages. Two parts. *International Journal of Computer and Information Sciences 8*, 5 (1979), 349-385, and *8*, 6 (1979), 435-471.

[Mag679b] MAGO, G.A. A cellular, language directed computer architecture. *Proceedings of the Caltech Conference on VLSI*, (January 1979), 447-452.

[Mag680] MAGO, G.A. A cellular computer architecture for functional programming. *Spring COMPCON '80. VLSI: New Architectural Horizons*, (1980), 179-185.

[MaSK81] MAGO, G.A., STANAT, D.F. and KOSTER, A. Program execution on a cellular computer: some matrix algorithms (In preparation).

[MaPa74] MALCOLM, M.A. and PALMER, J. A fast method for solving a class of tridiagonal linear systems. *Communications of the ACM 17*, 1 (1974), 14-17.

[MaRW68] MARTIN, R.S., REINSCH, C., and WILKINSON, J.H. Householder's tridiagonalization of a symmetric matrix. *Numerische Mathematik 11*, (1968), 181-195.

[Mira71] MIRANKER, W.L. A survey of parallelism in numerical analysis. *SIAM Review 13*, 4 (1971), 524-547.

[Orcu74] ORCUTT, S. Parallel solution methods for triangular linear systems of equations. *Technical Report No. 77*, Stanford Electronics Laboratories, Stanford, CA, 1974.

[OrVo77] ORTEGA, J.M. and VOIGT, R.G. Solution of partial differential equations on vector computers. *Report No. 77-7*, Institute for Computer Applications in Science and Engineering, NASA Langley Research Center, Hampton, VA, 1977.

[PeRa55] PEACEMAN, D.W. and RACHFORD, H.H.Jr. The numerical solution of parabolic and elliptic differential equations. *Journal of the Society for Industrial and Applied Mathematics 3*, 1 (1955), 28-41.

[PoVo74] POOLE Jr., W.G. and VOIGT, R.G. Numerical algorithms for parallel and vector computers: an annotated bibliography. *Computing Reviews 15*, 10 (1974), 379-388.

[Poze77] POZEFSKY, M. Programming in reduction languages. *Ph.D. dissertation*, Department of Computer Science, University of North Carolina

at Chapel Hill, 1977.

[Pres81] PRESNELL, H.A. Signal processing algorithms for a reduction machine. *Ph.D. dissertation in preparation*, Department of Computer Science, University of North Carolina at Chapel Hill.

[PrPa81] PRESNELL, H.A. and PARGAS, R.P Communication along shortest paths in a tree machine. *Proceedings of the Conference on Functional Programming Languages and Computer Architecture*, Portsmouth, 1981, 107-114.

[Rice79] RICE, J.R., GEAR, C.W., ORTEGA, J., PARLETT, B., SCHULTZ, M., SHAMPINE, L.F., WOLFE, P. and TRAUB, J.F. (eds.) Numerical computation: its nature and research directions. *ACM SIGNUM Newsletter Special Issue*, (February 1979).

[RoMK79] RODRIGUE, G.H., MADSEN, N.K. and KARUSH, J.I. Odd-even reduction for banded linear equations. *Journal of the ACM 26*, 1 (1979), 72-81.

[Same71] SAMEH, A.H. On Jacobi and Jacobi-like algorithms for a parallel computer. *Mathematics of Computation 25*, 115 (1971), 579-590.

[SaBr77] SAMEH, A.H. and BRENT, R.P. Solving triangular systems on a parallel computer. *SIAM Journal of Numerical Analysis 14*, 6 (1977), 1101-1113.

[SaKu77] SAMEH, A.H. and KUCK, D.J. Parallel direct linear system solvers - a survey. In *Parallel Computers - Parallel Mathematics*, M. Feilmeier (ed.), North-Holland Publishing Co., Amsterdam (1977), 25-30.

[SaKu78] SAMEH, A.H. and KUCK, D.J. On stable parallel linear system solvers. *Journal of the ACM 25*, 1 (1978), 81-91.

[Schu81] SCHULTZ, M. *Elliptic Problem Solvers*, Academic Press, New York, 1981.

[Ston73a] STONE, H.S. An efficient parallel algorithm for the solution of a tridiagonal linear system of equations. *Journal of the ACM 20*, 1 (1973), 27-38.

[Ston73b] STONE, H.S. Problems of parallel computation. In *Complexity of sequential and parallel numerical algorithms*, J.F. Traub (ed.), Academic Press, New York (1973), 1-16.

[Ston75] STONE, H.S. Parallel tridiagonal equation solvers. *ACM Transactions on Mathematical Software 1*, 4 (1975), 289-307.

[Ston76] STONE, H.S. Computer architecture in the 1980s. In *Computer Science and Scientific Computing*, J.M. Ortega (ed.), Academic Press, New York (1976), 127-153.

[Swar74a] SWARZTRAUBER, P.N. A direct method for the discrete solution of separable elliptic equations. *SIAM Journal on Numerical Analysis 11*, 6 (1974), 1136-1150.

[Swar74b] SWARZTRAUBER, P.N. The direct solution of the discrete Poisson equation on the surface of a sphere. *Journal of Computational Physics 15*, 1 (1974), 46-54.

[Swar79] SWARZTRAUBER, P.N. A parallel algorithm for solving general tridiagonal equations. *Mathematics of Computation 33*, 145 (1979), 185-199.

[SwSw73] SWARZTRAUBER, P.N. and SWEET, R.A. The direct solution of the discrete Poisson equation on a disk. *SIAM Journal on Numerical Analysis 10*, 5 (1973), 900-907.

[Swee73] SWEET, R.A. Direct methods for the solution of Poisson's equation on a staggered grid. *Journal of Computational Physics 12*, 3 (1973), 422-428.

[Swee74] SWEET, R.A. A generalized cyclic reduction algorithm. *SIAM Journal on Numerical Analysis 11*, 3 (1974), 506-520.

[Toll81] TOLLE, D.M. Coordination of computation in a binary tree of processors: an architectural proposal. *Ph.D. dissertation*, Department of Computer Science, University of North Carolina at Chapel Hill, 1981.

[ToSi81] TOLLE, D.M. and SIDDALL, W.E. On the complexity of vector computations in binary tree machines. *Information Processing Letters 13*, 3 (1981), 120-124.

[Trau73] TRAUB, J.F. Iterative solution of tridiagonal systems on parallel and vector computers. In *Complexity of Sequential and Parallel Numerical Algorithms*, J.F. Traub (ed.), Academic Press, New York, 1973, 49-82.

[Vara72] VARAH, J.M. On the solution of block-tridiagonal systems arising from certain finite-difference equations. *Mathematics of Computation 26*, 120 (1972), 859-868.

[Varg62] VARGA, R.S. *Matrix iterative numerical analysis*, Wiley, New York, 1962.

[Wall48] WALL, H.S. *Analytic Theory of Continued Fractions*, Van Nostrand, New York, 1948.

[Will81] WILLIAMS Jr., E.H. Analysis of FFP programs for parallel associative searching. *Ph.D. dissertation*, Department of Computer Science, University of North Carolina at Chapel Hill, 1981.

[Youn71] YOUNG, D.M. *Iterative solution of large linear systems*, Academic Press, New York, 1971.