International Journal of Computer and Information Sciences, Vol. 13, No. 4, 1984

Parallel Solution of Recurrences on a Tree Machine

Roy P. Pargas[1]

Received June 1983; revised July 1984

The recurrence

$$x_0 = a_0$$
$$x_i = a_i + b_i x_{i-1}, \qquad i = 1, 2, \ldots, n-1$$

requires $O(n)$ operations on a sequential computer. Elegant parallel solutions exist, however, that reduce the complexity to $O(\log N)$ using $N \geqslant n$ processors. This paper discusses one such solution, designed for a tree-structured network of processors.

A tree structure is ideal for solving recurrences. It takes exactly one sweep up and down the tree to solve any of several classes of recurrences, thus guaranteeing a solution in $O(\log N)$ time for a tree with $N \geqslant n$ leaf nodes. If n exceeds N, the algorithm efficiently pipelines the operation and solves the recurrence in $O(n/N + \log N)$ time.

KEY WORDS: Tree machine; parallel computation; recurrences.

1. INTRODUCTION

Consider the first-order linear recurrence

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

where $n \geqslant 1$ represents the number of terms of the recurrence and a_i and b_i are real scalars. The solution of Eq. 1, i.e., the set of values $(x_0, x_1, x_2, \ldots, x_{n-1})$, is obtained in a straightforward manner by a sequen-

[1] Department of Computer Science, Clemson University, Clemson, South Carolina 29631.

0091-7036/84/0800-0251$03.50/0

tial algorithm requiring a total of $n-1$ multiplications and $n-1$ additions, i.e., $O(n)$ operations, and little can be done on a sequential computer to improve the algorithm complexity. Elegant parallel solutions exist, however, that reduce the complexity to $O(\log N)$ using $N \geqslant n$ processors. This paper discusses one such solution, designed for a tree-structured network of processors.

Solving recurrences quickly is important because recurrences are so often components of larger problems. A tridiagonal linear system of equations, for example, can be transformed into several recurrence problems. Solving the recurrences provides a solution to the linear system.

A tree structure is ideal for solving recurrences. It takes exactly one sweep up and down the tree to solve any of several classes of recurrences, thus guaranteeing a solution in $O(\log N)$ time for a tree with $N \geqslant n$ leaf nodes. If n exceeds N, the algorithm efficiently pipelines the operation and solves the recurrence in $O(n/N + \log N)$ time.

This paper has four major parts. A description of the tree machine model used is presented in Section 2. The general tree algorithm, RECUR, is described and proven correct in Section 3. Recurrences to which RECUR is applicable are described in Section 4. These include first-, second- and higher-order linear recurrences, and recurrences of the form:

$$\begin{aligned} x_0 &= a_0 \\ x_i &= (a_i + b_i x_{i-1})/(c_i + d_i x_{i-1}) \quad i = 1, 2, \ldots, n-1 \end{aligned} \tag{2}$$

Extensions and variations of RECUR are presented in Section 5. Finally, conclusions and general remarks are given in Section 6.

Parallel solutions of linear recurrences have been studied before. In a paper on the parallel solution of tridiagonal linear systems, Stone[1] introduced a method called recursive doubling, which allows one to solve linear recurrences of all orders in $O(\log N)$ steps on a parallel processor of the ILLIAC-IV type. The method was generalized by Kogge and Stone[2] and by Kogge.[3] They described a broad class of functions that enjoy special composition properties and to which the method is applicable. Kogge[4] also described how to pipeline the method to obtain the maximal computational rate.

Studies on the relationship between computation time and number of processors when solving recurrences[5–10] have resulted in 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,[9] the algorithms were designed for an idealized N-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. Our approach is different in that we start with a well-defined processor network, i.e., a tree network, and this defines the manner in which processors may communicate with each other.

Therefore, two general approaches to the problem have emerged. The first one uses function composition systematically to reduce the dependencies among the variables of the linear recurrence.[1–4,9] Furthermore, algorithms are described with a specific parallel processor structure in mind. The second approach reorders the arithmetic operations required to solve the linear recurrence and distributes them among the available processors in order to minimize computation time.[5–8] The algorithms are not designed for a specific parallel processor. This paper adopts the former approach.

Interest in tree-structured parallel processors has grown in the past five years. Magó[11,12] has proposed a cellular computer organized as a binary tree of processors, which 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,[13] in which the expression of parallelism is natural. Tolle[14] 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[15] studied parallel associative searching algorithms and presented several techniques to predict and analyze the amount of time and storage required by the algorithms on a tree machine. Frank[16] designed a virtual memory for a tree machine such as Magó's. Koster[17] and Magó, Stanat, and Koster[18] developed methods for obtaining upper and lower bounds of the execution time of programs run on the machine proposed by Magó. Their analysis carefully accounts for communication and storage management costs. Parallel algorithms for tree machines have also been developed by Browning[19] for a variety of applications, including sorting, matrix multiplication, and the color cost problem; and by Bentley and Kung[20] for searching problems. Leiserson[21] studied systolic trees and how to maintain a priority queue on one.

2. THE TREE MACHINE (TM)

The model tree machine, TM (shown in Fig. 1), on which we describe algorithms is 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 or Tolle. Branches of the tree are two-way

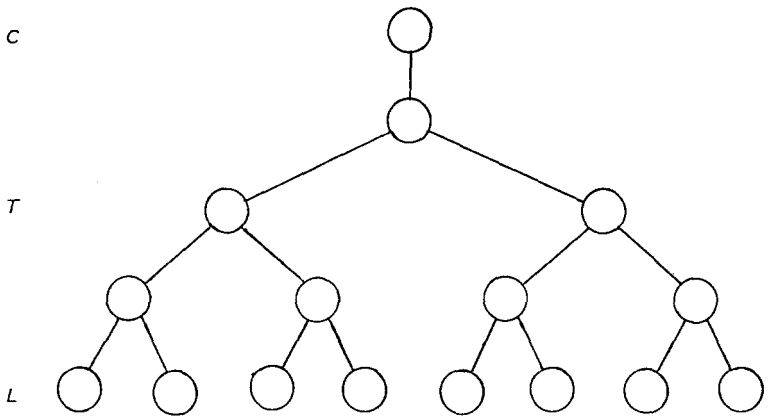

Fig. 1. Model of a tree machine. The top node is called the C-cell, interior nodes are called T-cells, and the leaf nodes are called L-cells.

communication links. Leaf and nonleaf processing elements are called L-cells and T-cells respectively. Attached to the root cell, functioning as the root cell's parent, 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 row of T-cells is 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 parent and two children and an L-cell may communicate with its parent; a C-cell communicates with the root T-cell and with external storage. 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 child the root cell, which sends the information to its two children, which send the information to their children, 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.

There is one important condition we impose on the programming of the tree cells: all cells of the same type must execute identical programs.

The main reason for this is practicality. The programming task is simplified, and feasible, because the programmer must write no more than three programs, one each for the C-cell, T-cells, and L-cells.

3. THE BASIC TREE ALGORITHM: RECUR

3.1. Description of RECUR

The purpose of this section is to present, and prove correct, the tree algorithm, RECUR. RECUR is intended to be a general algorithm, applicable to a variety of recurrences. Examples of these recurrences are presented in Section 4.

We start with the following definitions.

Definition 1. A *recurrence expression*, RE, is the pair $(C, \cdot)$ where

$$C = \{C_{i,j} \mid i \geqslant j \geqslant -1\}$$

and "$\cdot$" is a binary operator on the elements of C. The operator "$\cdot$", which we call *composition*, must satisfy the following property:

$$C_{i,j} \cdot C_{j,k} = C_{i,k}, \qquad i \geqslant j \geqslant k \geqslant -1$$

We call the subset

$$C_I = \{C_{i,i-1} \mid i \geqslant 0\}$$

the *initial values* of RE, and the subset

$$C_S = \{C_{i,-1} \mid i \geqslant 0\}$$

the *solution set* of RE.

Almost always, we are interested only in a finite subset of C, defined as follows.

Definition 2. A *recurrence expression of size n*, RE_n, is the pair $(C_n, \cdot)$ where

$$C_n = \{C_{i,j} \mid n-1 \geqslant i \geqslant j \geqslant -1\}$$

The corresponding initial and solution sets are

$$C_{In} = \{C_{i,i-1} \mid n-1 \geqslant i \geqslant 0\}$$

and

$$C_{Sn} = \{C_{i,-1} \mid n-1 \geqslant i \geqslant 0\}$$

With these definitions, the following lemmas are easily shown true.

Lemma 1. For all i and j such that $i \geqslant j \geqslant -1$,

$$C_{i,j} = C_{i,i} \cdot C_{i,j} = C_{i,j} \cdot C_{j,j}$$

Proof. Follows immediately from Definition 1. ▮

Lemma 2. Composition is associative, i.e.,

$$(C_{i,j} \cdot C_{j,k}) \cdot C_{k,l} = C_{i,j} \cdot (C_{j,k} \cdot C_{k,l})$$

for all $i \geqslant j \geqslant k \geqslant l \geqslant -1$.

Proof. From Definition 1, we know that

$$(C_{i,j} \cdot C_{j,k}) \cdot C_{k,l} = C_{i,k} \cdot C_{k,l} = C_{i,l}$$

and that

$$C_{i,j} \cdot (C_{j,k} \cdot C_{k,l}) = C_{i,j} \cdot C_{j,l} = C_{i,l}$$

Hence, the lemma is true. ▮

Recurrence problems typically require computing for the solution set C_{Sn} given only the initial set C_{In}. A straightforward sequential solution to this problem is provided by the algorithm in Fig. 2. For example, if $n = 4$ and we want to solve for $C_{3,1}$, the sequential algorithm determines the elements of C_{Sn} in a manner suggested by

$$C_{3,2} \cdot (C_{2,1} \cdot (C_{1,0} \cdot C_{0,-1}))$$

```
C_Sn = { C_0,-1 }
for i=1 to n-1 do
    C_i,-1 = C_i,i-1 • C_i-1,-1
    Add C_i,-1 to the set C_Sn
    end
```

Fig. 2. Sequential algorithm to solve a recurrence problem.

i.e., first obtain $C_{1,-1} = C_{1,0} \cdot C_{0,-1}$, then $C_{2,-1} = C_{2,1} \cdot C_{1,-1}$, and finally $C_{3,-1} = C_{3,2} \cdot C_{2,-1}$. (Note that one element of C_{Sn}, i.e., $C_{0,-1}$, was initially available.) The associativity of "$\cdot$", however, allows us to modify the order in which the partial results are obtained. We may opt to solve for $C_{3,-1}$ in the following manner instead:

$$(C_{3,2} \cdot C_{2,1}) \cdot (C_{1,0} \cdot C_{0,-1})$$
$$C_{3,1} \cdot C_{1,-1}$$
$$C_{3,-1}$$

This suggests that we may independently (and simultaneously) solve for $C_{3,1}$ and $C_{1,-1}$, and then solve for $C_{3,-1}$. This is the basis for the parallel algorithm, RECUR.

Consider a tree machine with N L-cells and let L_i be the ith L-cell counting from the right. Let $n = N$ be the size (i.e., number of terms) of a recurrence expression $(C_n, \cdot)$ as described in Definition 2. The initial values $C_{In} = \{C_{i,i-1},\ 0 \leqslant i \leqslant n-1\}$, are stored one per L-cell with $C_{i,i-1}$ stored in L_i. Figure 3 shows the initial state of an 8-L-cell tree machine.

The object is to solve for the set C_{Sn}. Figure 4 shows the instructions executed by the L-, T-, and C-cells. We study the execution of RECUR by stepping through the instructions and observing how cells interact with each other.

RECUR consists of an upward and a downward sweep through the tree. The L-cells start the upward sweep by sending the values they contain to their parents (line l_1). L_i sends up $C_{i,i-1}$ and waits to receive two solution values, $C_{i,-1}$ and $C_{i-1,-1}$, during the downward sweep (line l_2). A T-cell first waits to receive a value each from its left and right children (line t_1). When the values arrive, a T-cell applies the composition operator "$\cdot$", and sends the result to its parent (line t_2). The sweep continues upward with each T-cell sending a value (the result of composition) to its father. The upward sweep ends with the C-cell receiving the value $C_{n-1,-1}$ from the root T-cell (line c_1).

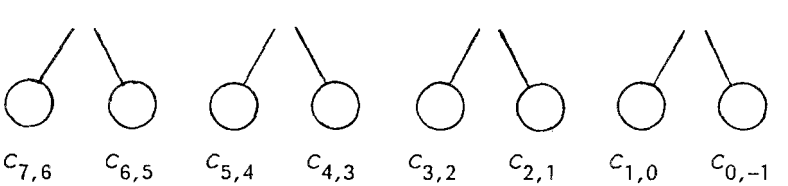

Fig. 3. Distribution of the initial values $C_{i,i-1}$ among the L-cells for $n = 8$. $C_{0,-1}$ is stored in the rightmost L-cell, $C_{1,0}$ in the next L-cell to the left, and so on .

L cell:	l_1	send $(C_{i,i-1})$
	l_2	receive $(C_{i,-1}, C_{i-1,-1})$
T cell:	t_1	L.receive $(C_{i,j})$, R.receive $(C_{j,k})$
	t_2	P.send $(C_{i,j} \bullet C_{j,k})$
	t_3	P.receive $(C_{i,-1}, C_{k,-1})$
	t_4	$C_{j,-1} = C_{j,k} \bullet C_{k,-1}$
	t_5	L.send $(C_{i,-1}, C_{j,-1})$, R.send $(C_{j,-1}, C_{k,-1})$
C cell:	c_1	receive $(C_{n-1,-1})$
	c_2	send $(C_{n-1,-1}, C_{-1,-1})$

Fig. 4. L-, T-, and C-cell programs for RECUR.

The subscripts used in the algorithm are purely for ease of presentation. A cell is not aware of the identity of the value that it contains. From our global point of view, however, we are able to make a few conclusions regarding the upward sweep.

Lemma 3. Let the initial values, C_{in}, of a recurrence expression of size n be distributed among the L-cells of a tree machine, so that L_i contains $C_{i,i-1}$, $0 \leqslant i \leqslant n-1$. Let T be an arbitrary T-cell with children T_L and T_R. Let L_i and L_{j+1} be the leftmost and rightmost L-cells in T's left subtree, and let L_j and L_{k+1} be the leftmost and rightmost L-cells in its right subtree (see Fig. 5). Then, during the upward sweep of the algorithm RECUR,

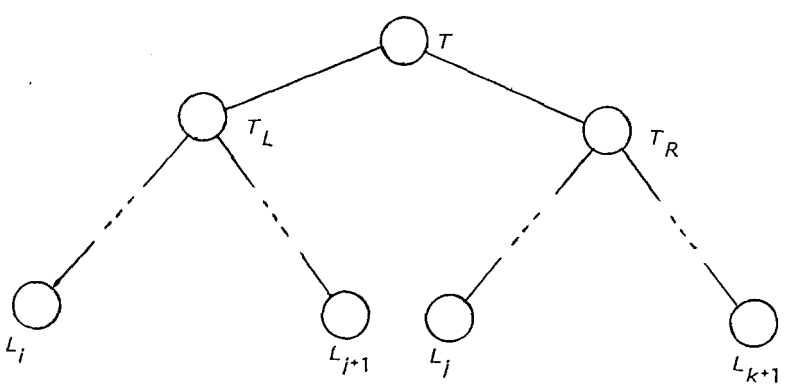

Fig. 5. Let T be an arbitrary T-cell with left and right children T_L and T_R, respectively. Let the leftmost and rightmost leaves of T_L be L_i and L_{j+1} and let the leftmost and rightmost leaves of T_R be L_j and L_{k+1}.

(a) T receives $C_{i,j}$ from its left child, T_L,
(b) T receives $C_{j,k}$ from its right child, T_R, and
(c) T sends $C_{i,k} = C_{i,j} \cdot C_{j,k}$ to its parent.

Proof. Proof by induction on the level number of the T-cells.

Basis. Let T be a level 1 T-cell, i.e., T's children are L-cells (Fig. 6). Because there is only one L-cell in T's left subtree, $L_i = L_{j+1}$. Hence, $i = j+1 => j = i-1$. The value that T receives from its left child is $C_{i,i-1} = C_{i,j}$, proving part (a). Similarly, there is only one L-cell in T's right subtree, $L_j = L_{k+1}$. Hence, $j = k+1 => k = j-1$. The value that T receives from its right child is $C_{j,j-1} = C_{j,k}$, proving part (b). Finally, line t_2 of Fig. 4 shows that T computes $C_{i,k} = C_{i,j} \cdot C_{j,k}$ and sends $C_{i,k}$ to its parent, proving part (c).

Hypothesis. Assume the lemma true for all T-cells on level h, $h \geqslant 1$.

Conclusion. Consider a T-cell on level $h+1$. L_i and L_{j+1} are the leftmost and rightmost L-cells in the subtree of which T_L is a root. By hypothesis, T_L sends $C_{i,j}$ to T, proving part (a). Similarly, L_j and L_{k+1} are the leftmost and rightmost L-cells in the subtree of which T_R is a root. By hypothesis, T_R sends $C_{j,k}$ to T, proving part (b). Finally, line t_2 of Fig. 4 guarantees that T then sends $C_{i,k}$ to its parent, proving part (c). This lemma is summarized in Fig. 7a. ▮

Lemma 4. Let C be the C cell of the tree machine described in Lemma 3. At the end of the upward sweep of RECUR, C receives the element $C_{n-1,-1}$.

Proof. Let T be the root T-cell. Then $i = n-1$ and $k = -1$. Lemma 3 tells us that the root T-cell sends the element $C_{i,k} = C_{n-1,-1}$ to its parent, C. Line c_1 of Fig. 4 tells us that C receives the pair. ▮

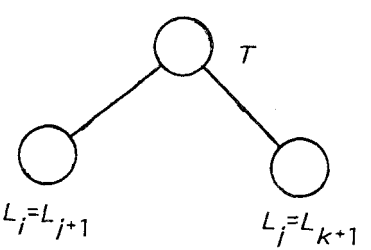

Fig. 6. If T's children are L-cells, then T's left child must be $L_i = L_{j+1}$ and T's right child must be $L_j = L_{k+1}$. Therefore, $i = j+1$ and $j = k+1$.

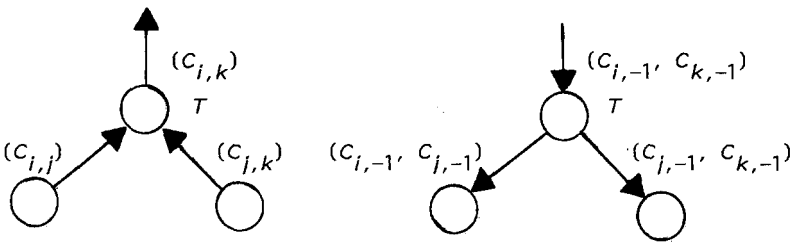

Fig. 7. (a) During the upward sweep, T receives $C_{i,j}$ from its left child, receives $C_{j,k}$ from its right child, and sends $C_{i,k} = C_{i,j} \cdot C_{j,k}$ to its parent. (b) During the downward sweep, the same T-cell, T, receives $(C_{i,-1}, C_{k,-1})$ from its parent, computes $C_{j,-1} = C_{j,k} \cdot C_{k,-1}$, and sends $(C_{i,-1}, C_{j,-1})$ to its left child and $(C_{j,-1}, C_{k,-1})$ to its right child.

During the upward sweep, several elements of C_{Sn} were solved. One of them, $C_{n-1,-1}$, was received by the C-cell. During the downward sweep, we solve for the remaining elements of C_{Sn}, with each T-cell providing one solution value.

The downward sweep begins when the C-cell sends the pair $(C_{n-1,-1}, C_{-1,-1})$ to the root T-cell (line c_2). The first component is the value the C-cell received during the upward sweep; the second is a constant known *a priori* by the C-cell. In general, a T-cell that received the value $C_{i,j}$ and $C_{j,k}$ from its children during the upward sweep receives the values $C_{i,-1}$ and $C_{k,-1}$ from its father during the downward sweep (line t_3). The T-cell uses $C_{k,-1}$ to solve for $C_{j,-1} = C_{j,k} \cdot C_{k,-1}$ (line t_4) and sends the values $C_{i,-1}$ and $C_{j,-1}$ to its left child, and the values $C_{j,-1}$ and $C_{k,-1}$ to its right child (line t_5). The downward sweep ends when the ith L-cell receives $C_{i,-1}$ and $C_{i-1,-1}$ (line l_2).

As with the upward sweep, T-cells and L-cells are unaware of the identities (subscripts) of the values they receive. However, the following lemma identifies them for us.

Lemma 5. Let T be the T-cell described in Lemma 4, i.e., during the upward sweep of RECUR, T received $C_{i,j}$ and $C_{j,k}$ from its left and right children, respectively. Then, during the downward sweep,

(a) T receives $(C_{i,-1}, C_{k,-1})$ from its parent,
(b) T sends $(C_{i,-1}, C_{j,-1})$ to its left child, and
(c) T sends $(C_{j,-1}, C_{k,-1})$ to its right child,

Proof. Proof by induction on the level number of the T-cells.

Basis. Let T be the root T-cell, i.e., T is on level log N. T received $C_{i,j} = C_{n-1,j}$ and $C_{j,k} = C_{j,-1}$ from its left and right children during the

upward sweep. Hence, $i = n-1$ and $k = -1$. During the downward sweep, T receives $C_{n-1,-1} = C_{i,-1}$ and $C_{-1,-1} = C_{k,-1}$ from the C-cell, proving part (a). Line t_4 of Fig. 4 shows that T computes $C_{j,-1}$ using $C_{j,k}$ and $C_{k,-1}$. In line t_5, T sends $(C_{i,-1}, C_{j,-1})$ to its left child and $(C_{j,-1}, C_{k,-1})$ to its right child, proving parts (b) and (c).

Hypothesis. Assume the lemma true for all T-cells on level $h \leqslant \log N$.

Conclusion. Let T be a T-cell on level $h-1$. By hypothesis, T's parent sends $(C_{i,-1}, C_{k,-1})$ to T, proving part (a). Lines t_4 and t_5 show that T sends $(C_{i,-1}, C_{j,-1})$ to its left child and $(C_{j,-1}, C_{k,-1})$ to its right child, proving parts (b) and (c). This lemma is summarized in Fig. 7b.

Lemma 6 shows that the solution set C_{Sn} is received by the L-cells at the end of the downward sweep.

Lemma 6. During the downward sweep, L_i receives $C_{i,-1}$ and $C_{i-1,-1}$.

Proof. In general, during the upward sweep, T receives $C_{i,j}$ and $C_{j,k}$ from its left and right children. Lemma 5 states that the same T returns $C_{i,-1}$ and $C_{j,-1}$ to its left child and $C_{j,-1}$ and $C_{k,-1}$ to its right child. We need to show that, regardless of whether L_i is a left or a right child, L_i receives $C_{i,-1}$ and $C_{i-1,-1}$.

If L_i is the left child of T, then $C_{i,i-1} = C_{i,j} = >i-1 = j$. Lemma 5 states that T returns $C_{i,-1}$ and $C_{j,-1} = C_{i-1,-1}$ to L_i. If L_i is the right child of T, then $C_{i,i-1} = C_{j,k} = >i = j$ and $i-1 = k$. Lemma 5 states that T returns $C_{j,-1} = C_{i,-1}$ and $C_{k,-1} = C_{i-1,-1}$ to L_i. Hence, the lemma is true. ∎

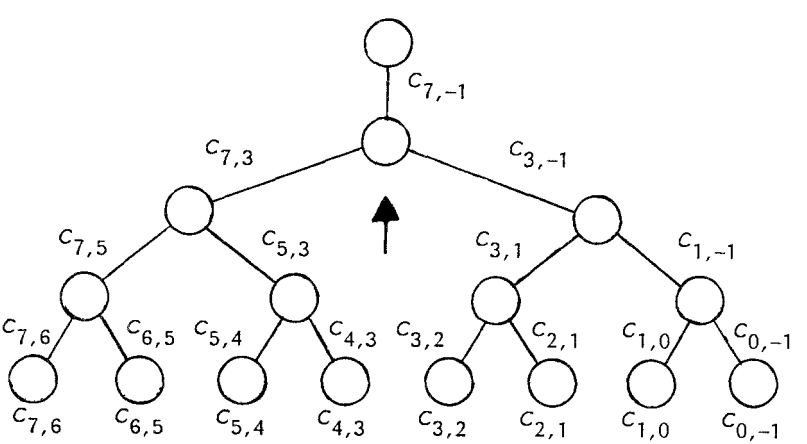

Fig. 8. Full upward sweep of RECUR for $N = n = 8$.

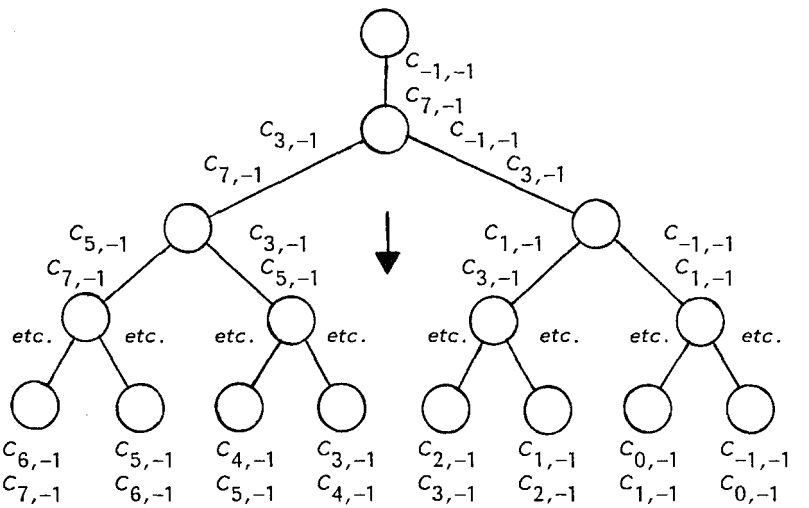

Fig. 9. Full downward sweep of RECUR for $N = n = 8$.

L_i now holds the element $C_{i,-1}$ (as well as $C_{i-1,-1}$). The L-cells, therefore, collectively hold C_{Sn} and the recurrence problem is solved. Figures 8 and 9 show the full upward and downward sweeps for $n = 8$. Lemmas 3, 4, 5, and 6 together prove the following theorem.

Theorem 1. RECUR correctly solves a recurrence of size n on a tree machine with $N = n$ leaf cells in a single sweep up and down the tree.

3.2. Analysis of RECUR

Analyzing RECUR is straightforward and simple. The time complexity of RECUR on a tree machine can be measured by (a) the amount of time involved in moving data from cell to cell, i.e., communication time, and (b) the number of arithmetic operations performed by the tree cells. We make the following assumptions refarding communication.

1. A cell can send one unit of data (e.g., one floating point number) to another cell in one time step.
2. A cell can send data to, and receive data from, each of its parent and children in the same time step.
3. Ceels operate synchronously. This allows us, for example, to imagine all of the L-cells sending information to their parents simultaneously.

A tree machine with N L-cells has $\log N$ levels of T-cells. During the upward sweep, each L-cell and each T-cell must send $|C_{i,j}|$ units of data to its parent. Similarly, during the downward sweep, the C-cell and each T-cell must send $2\,|C_{i,j}|$ units of data to its children. Because we assume that cells on the same level operate simultaneously,

$$C(n) = (|C_{i,j}| + 2\,|C_{i,j}|)(\log N + 1) \tag{3}$$

describes total communication time.

Let a_{Lu} and m_{Lu} be the number of additions and multiplications, respectively, performed by a single L-cell during the upward sweep. Let a_{Ld} and m_{Ld} be similarly defined for the downward sweep. Let the corresponding values for the T- and C-cells be a_{Tu}, m_{Tu}, a_{Td}, m_{Td}, a_{Cu}, m_{Cu}, a_{Cd}, and m_{Cd}. Then the total number of parallel additions performed during both sweeps of RECUR is

$$A(n) = (a_{Lu} + a_{Ld} + a_{Cu} + a_{Cd}) + (a_{Tu} + a_{Td}) \log N \tag{4}$$

where a parallel addition is one row of cells executing one addition. Similarly, the total number of parallel multiplications performed is

$$M(n) = (m_{Lu} + m_{Ld} + m_{Cu} + m_{Cd}) + (m_{Tu} + m_{Td}) \log N. \tag{5}$$

RECUR, therefore, is $O(\log N)$.

Note that only the T-cells performed any computation. Each T-cell performed composition $(\cdot)$ once during the upward sweep and once during the downward sweep. Thus

$$\begin{aligned} &a_{Tu} = a_{Td} = \text{number of additions in ``}\cdot\text{''} \\ &m_{Tu} = m_{Td} = \text{number of multiplications in ``}\cdot\text{''} \\ &a_{Lu} = a_{Ld} = a_{Cu} = a_{Cd} = m_{Lu} = m_{Ld} = m_{Cu} = m_{Cd} = 0 \end{aligned} \tag{6}$$

Equations 4 and 5 are deliberately made more general, however, to accommodate variations to RECUR described in Section 5. For specific recurrence expressions, we need only determine the $|C_{i,j}|$, and the various a's and m's to obtain the communication and operation counts. All of this is summarized in Fig. 10.

SWEEP	COMMMUNICATION	L	T	C
up	$\|C_{i,j}\|$ (log N + 1)	a_{Lu}	a_{Tu}(log N)	a_{Cu}
		m_{Lu}	m_{Tu}(log N)	m_{Cu}
down	$2\|C_{i,j}\|$(log N + 1)	a_{Ld}	a_{Td}(log N)	a_{Cd}
		m_{Ld}	m_{Td}(log N)	m_{Cd}

Fig. 10. This figure summarizes the number of communication steps and arithmetic operations required by RECUR for a specific application. $|C_{i,j}|$ is a measure of the number of components (e.g. real numbers) represented by the element $C_{i,j}$. The variables a and m stand for additions and multiplications. The subscripts L, T, and C indicate the cell in which the operation is performed. Finally, the subscripts u and d stand for upward and downward sweeps.

4. APPLICATIONS OF RECUR

Section 3 presented the algorithm RECUR in general. In this section, we apply RECUR to specific recurrences. For a given recurrence, we must precisely describe the meanings of:

(a) the set of objects, C_n

(b) the set of initial values, C_{In}

(c) the set of solution values, C_{Sn}, and

(d) the composition operator, $\cdot$

Once this is done, RECUR provides the general manner in which data are moved up and down the tree.

4.1. First-Order Linear Recurrences

Consider the first-order linear recurrence shown in Eq. 1. Our objective is to obtain the values x_i, $0 \leqslant i \leqslant n-1$. For the sake of uniformity, we modify Eq. 1 by defining

$$x_0 = a_0 + b_0 x_{-1} \tag{7}$$

where $b_0 = 0$ and x_{-1} is a dummy variable. Equation 1 can now be restated

$$x_i = a_i + b_i x_{i-1}, \qquad i = 0, \qquad 1,..., n-1 \tag{8}$$

which expresses each x_i as a function of x_{i-1}. It is also possible to express x_i as a function of x_j for all $j \leqslant i$. Trivially,

$$x_i = 0 + 1x_i \tag{9}$$

which expresses x_i as a function of itself. In addition,

$$\begin{aligned} x_i &= a_i + b_i x_{i-1} \\ &= (a_i + b_i a_{i-1}) + (b_i b_{i-1})\, x_{i-2} \\ &\quad (a_i + b_i a_{i-1} + b_i b_{i-1} a_{i-2}) + (b_i b_{i-1} b_{i-2})\, x_{i-3} \\ &= \ldots \\ &= \left(a_i + \sum_{r=j+1}^{i-1} a_r \prod_{s=r+1}^{i} b_s\right) + \left(\prod_{s=j+1}^{i} b_s\right) x_j \end{aligned} \tag{10}$$

for all j, $-1 \leqslant j < i$. This expansion is obtained by repeatedly applying the following rule:

$$\begin{aligned} &\text{if} \quad x_i = a + b x_j \\ &\text{and} \quad x_j = a' + b' x_k \\ &\text{then} \quad x_i = (a + ba') + (bb')\, x_k \end{aligned} \tag{11}$$

With this rule as a foundation, we may now define the sets C_n, C_{ln}, and C_{Sn}, as the composition operator in the context of first-order linear recurrences.

Let $C_{i,j}$ be the coefficients of the equation expressing x_i as a function of x_j. For example, for all i and j, $(-1 \leqslant j \leqslant i \leqslant n-1)$, C_n is the set of ordered pairs:

$$C_n = \{C_{i,j} = (a, b) \mid x_i = a + b x_j\} \tag{12}$$

From Eq. 10, we obtain an explicit formula for any element $C_{i,j}$:

$$\begin{aligned} &\text{if} \quad i = j \quad \text{then} \quad C_{i,j} = (0, 1) \\ &\qquad \text{else} \quad C_{i,j} = \left(a_i + \sum_{r=j+1}^{i-1} a_r \prod_{s=r+1}^{i} b_s\right) + \left(\prod_{s=j+1}^{i} b_s\right) x_j \end{aligned} \tag{13}$$

The set of initial values, C_{ln}, is the set

$$C_{ln} = \{C_{i,i-1} = (a_i, b_i) \mid 0 \leqslant i \leqslant n-1\} \tag{14}$$

which are, indeed, the initial values provided by Eq. 8. The solution set, C_{Sn}, is defined

$$C_{Sn} = \{C_{i,-1} \mid 0 \leqslant i \leqslant n-1\} \tag{15}$$

It is not immediately obvious that this set is equivalent to the set of solution values x_i, $0 \leqslant i \leqslant n-1$, satisfying Eq. 8. We observe from Eq. 13, however, that

$$\begin{aligned} C_{i,-1} &= \left(a_i + \sum_{r=0}^{i-1} a_r \prod_{s=r+1}^{i} b_s, \prod_{s=0}^{i} b_s \right) \\ &= \left(a_i + \sum_{r=0}^{i-1} a_r \prod_{s=r+1}^{i} b_s, 0 \right) \end{aligned} \tag{16}$$

since $b_0 = 0$. Therefore, $C_{i,-1}$ is the ordered pair $(a, 0)$ such that

$$x_i = a + 0x_{-1} = a \tag{17}$$

The first component of $C_{i,-1}$ is, indeed, the solution of x_i.

The final requirement is to define the composition operator "$\cdot$". We use Eq. 11 to obtain

$$\begin{aligned} &\text{if} \quad C_{i,j} = (a, b) \\ &\text{and} \quad C_{j,k} = (a', b') \\ &\text{then} \quad C_{i,j} \cdot C_{j,k} = (a + ba', bb'). \end{aligned} \tag{18}$$

We must now prove that composition, as defined by Eq. 18, satisfies the composition property.

Lemma 7. The composition operator, defined in Eq. 18, satisfies the composition property described in Definition 1.

$$C_{i,k} = C_{i,j} \cdot C_{j,k}, \qquad -1 \leqslant k \leqslant j \leqslant i \leqslant n-1$$

Proof. From Eq. 13, we know that

$$C_{i,j} = \left(a_i + \sum_{r=j+1}^{i-1} a_r \prod_{s=r+1}^{i} b_s, \prod_{s=j+1}^{i} b_s \right)$$

and

$$C_{j,k} = \left(a_j + \sum_{r=k+1}^{j-1} a_r \prod_{s=r+1}^{j} b_s, \prod_{s=k+1}^{j} b_s \right)$$

From Eq. 18, we have

$$
\begin{aligned}
&C_{i,j} \cdot C_{j,k} \\
&= \left(a_i + \sum_{r=j+1}^{i-1} a_r \prod_{s=r+1}^{i} b_s + \prod_{s=j+1}^{i} b_s \left(a_j + \sum_{r=k+1}^{j-1} a_r \prod_{s=r+1}^{j} b_s \right), \right. \\
&\qquad \left. \prod_{s=j+1}^{i} b_s \prod_{s=k+1}^{j} b_s \right) \\
&= \left(a_i + \sum_{r=j+1}^{i-1} a_r \prod_{s=r+1}^{i} b_s + a_j \prod_{s=j+1}^{i} b_s + \prod_{s=j+1}^{i} b_s \sum_{r=k+1}^{j-1} a_r \prod_{s=r+1}^{j} b_s, \right. \\
&\qquad \left. \prod_{s=k+1}^{i} b_s \right) \\
&= \left(a_i + \sum_{r=j+1}^{i-1} a_r \prod_{s=r+1}^{i} b_s + a_j \prod_{s=j+1}^{i} b_s + \sum_{r=k+1}^{j-1} a_r \prod_{s=r+1}^{i} b_s, \quad \prod_{s=k+1}^{i} b_s \right) \\
&= \left(a_i + \sum_{r=k+1}^{i-1} a_r \prod_{s=r+1}^{i} b_s, \prod_{s=k+1}^{i} b_s \right) \\
&= C_{i,k}
\end{aligned}
$$

Now that the sets C_n, C_{ln}, and C_{Sn}, and the operator $\cdot$ have been defined, the problem of solving the first n terms of a first-order linear recurrence can be couched in terms of an RE_n (Definition 2) and the algorithm RECUR may be used to solve such a problem. The algorithm distributes the initial values C_{ln} one to an L-cell, performs an upward sweep which sends $C_{n-1,-1}$ to the C-cell, and performs a downward sweep which sends $C_{i,-1}$ and $C_{i-1,-1}$ to L_i, $0 \leqslant i \leqslant n-1$.

4.2. Other applications

Table I shows seven types of recurrences that can be solved using RECUR. For each problem, the crucial question is whether a composition operator "$\cdot$" can be defined which satisfies the composition property (Definition 1). The ith initial value, $C_{i,i-1}$, and the definition of the composition operator for each of the seven types is given in Table II. For example, recurrence type 1 of Table I produces the sum or the product of a vector of scalars or matrices in $O(\log N)$ time. Note that, at no extra cost, RECUR also gives all partial sums or products. The required composition operation is given in line 1 of Table II. A T-cell merely sends to its parent the sum or product of the values it receives from its children. In recurrence

Table I. Seven Classes of Recurrence Types and the Domains of the Variables Involved[a,b]

Recurrence Type	Domains of Variables
1. $x_i = a_i\ op1\ x_{i-1}$	$op1 = \{\times, +\}$, $D_a = D_x = D^5$
2. $x_i = a_i\ op2\ x_{i-1}$	$op2 = \{\text{AND, OR, XOR}\}$, $D_a = D_x = D^6$
3. $x_i = op3\ (a_i, x_{i-1})$	$op3 = \{\min, \max\}$, $D_a = D_x = D^1$
4. $x_i = a_i + b_i x_{i-1}$	$D_a = D_x = D^3, D^5$, $D_b = D^5$
5. $x_i = a_i$ OR (b_i AND x_{i-1})	$D_a = D_x = D^4, D^6$, $D_b = D^6$
6. $x_i = (a_i + b_i x_{i-1})/(c_i + d_i x_{i-1})$	$D_a = D_b = D_c = D_d = D_x = D^1$
7. $x_i = (a_i$ OR b_i AND $x_{i-1})$ AND NOT (c_i OR d_i AND x_{i-1})	$D_a = D_b = D_c = D_d = D_x = D^2$

[a] D^1 = real numbers; D^2 = Boolean values; D^3 = real m-vectors; D^4 = Boolean m-vectors; $D^5 = (m \times m)$ real matrices; $D^6 = (m \times m)$ Boolean matrices; for D^3 through D^6, $m \geqslant 1$.

[b] The domain of the variables a and x (D_a and D_x, respectively) are m-vectors or $(m \times m)$ real matrices. The domain of the variable b, D_b is the set of $(m \times m)$ real matrices.

classes 1, 2, and 3, the recurrence operator may be one of several. Recurrence type 4 includes first-order linear recurrences. In the next section, however, we see that recurrence type 4 also includes second- and higher-order linear recurrences. Recurrence type 6 includes recurrences known as partial fractions and continued fractions. Finally, RECUR may also be used to solve recurrences involving Boolean variables. The rest of this section briefly describes two specific applications of RECUR.

Table II. Definition of Composition Operator for Seven Recurrence Classes[a]

$C_{i,i-1}$	$C_{i,j} \cdot C_{j,k} = C_{i,k}$
1. a_i	$a \cdot a' = a\ op1\ a'$
2. a_i	$a \cdot a' = a\ op2\ a'$
3. a_i	$a \cdot a' = op3\ (a, a')$
4. (a_i, b_i)	$(a, b) \cdot (a', b') = (a + ba', bb')$
5. (a_i, b_i)	$(a, b) \cdot (a', b') = (a$ OR b AND a', b AND $b')$
6. (a_i, b_i, c_i, d_i)	$(a, b, c, d) \cdot (a', b', c', d') = (ac' + ba', ad' + bb', cc' + da', cd' + db')$
7. (a_i, b_i, c_i, d_i)	$(a, b, c, d) \cdot (a', b', c' d') = (a$ AND c' OR ba', a AND d' OR b AND $b',$ c AND c' OR d AND a', c AND d' OR d AND $b')$

[a] For each recurrence class shown in Table I, the corresponding set of initial values, C_{In}, and the composition operator, $\cdot$, is given. For example, for first-order linear recurrences (Table I, recurrence class 4), the set C_{In} is the set of pairs (a_i, b_i) and composition is defined as follows: $(a, b) \cdot (a', b') = (a + ba', bb')$.

4.2.1. Second-Order Linear Recurrences

We want to solve a recurrence of the form

$$\begin{aligned} x_0 &= a_0 \\ x_1 &= a_1 + b_1 x_0 \\ x_i &= a_i + b_i x_{i-1} + c_i x_{i-2}, \qquad 2 \leqslant i \leqslant n-1 \end{aligned} \tag{19}$$

For the sake of uniformity, we redefine Eq. 19 thus:

$$\begin{aligned} x_0 &= a_0 + b_0 x_{-1} + c_0 x_{-2} \\ x_1 &= a_1 + b_1 x_0 + c_1 x_{-1} \\ x_i &= a_i + b_i x_{i-1} + c_i x_{i-2}, \qquad 2 \leqslant i \leqslant n-1 \end{aligned} \tag{20}$$

where $b_0 = c_0 = c_1 = 0$, and x_{-1} and x_{-2} are dummy variables. Equation 20 can now be rewritten

$$x_i = a_i + b_i x_{i-1} + c_i x_{1-2}, \qquad 0 \leqslant i \leqslant n-1 \tag{21}$$

In order to use RECUR, we employ a change of variables:

$$\begin{aligned} Y_i &= (x_i, x_{i-1})^{-1} \\ A_i &= (a_i, a_{i-1})^{-1} \\ B_i &= \begin{pmatrix} b_i & c_i \\ 0 & 1 \end{pmatrix} \end{aligned} \tag{22}$$

for $-1 \leqslant i \leqslant n-1$. This allows Eq. 21 to be rewritten as

$$Y_i = A_i + B_i Y_{i-1}, \qquad -1 \leqslant i \leqslant n-1 \tag{23}$$

Equation 23 is a first-order linear recurrence with matrix coefficients and RECUR can be applied directly. Third- and higher-order linear recurrences can be transformed in a similar manner.

4.2.2. Fractions

RECUR may also be used to solve recurrences of the form

$$x_i = (a_i + b_i x_{i-1})/(c_i + d_i x_{i-1}) \tag{24}$$

Note that Eq. 24 is called a partial fraction if $b_i = 0$, and a continued fraction if $c_i = 0$. Lines 6 of Tables I and II summarize the steps we need to

take in order to apply RECUR. The ith initial value, $C_{i,i-1}$, is the 4-tuple (a_i, b_i, c_i, d_i). A T-cell receives (a, b, c, d) and (a', b', c', d') from its left and right children and sends

$$(ac' + ba', ad' + bb', cc' + da', cd' + db') \tag{25}$$

to its parent. At the end of the downward sweep, L_i receives $C_{i,-1}$ and $C_{i-1,-1}$. $C_{i,-1}$ is the 4-tuple $(a, 0, c, 0)$ where $x_i = a/c$. L_i must perform an extra division to obtain x_i.

5. EXTENSIONS

RECUR has been described for the case where n, the size of the recurrence, equals N the number of leaf cells in the tree machine. In this section, we describe three interesting variations of RECUR: (a) how the extra (empty) L-cells are to be programmed if $n < N$, (b) how to solve two or more recurrences of the same type simultaneously, and (c) how to pipeline operations to achieve maximum productivity if $n > N$.

5.1. Empty L-Cells

The need to solve a recurrence may merely be one of many steps of a complex process. If such a process is being executed on a general purpose tree machine, such as that proposed by Magó, it may not be possible to guarantee that, when the time for solving the recurrence arrives, the data which comprise the initial values are stored in contiguous L-cells. Some of the L-cells may be idle, or active but not meant to participate in the recurrence solution. RECUR easily handles such situations. The sole requirement is that the $C_{i,j}$ are distributed from right to left, i.e., $C_{0,-1}$ is stored in the rightmost participating L-cell, $C_{1,0}$ be stored in the next participating L-cell to the left, and so on. We call all nonparticipating L-cells "empty". Surprisingly, the T-cell algorithm described in Fig. 4 does not need to be modified. The empty L-cells, however, must be initialized and must perform a minor role.

Figure 11 shows two nonempty L-cells containing $C_{i,i-1}$ and $C_{i-1,i-2}$,

	L_i	*empty*	*empty*	*empty*	L_{i-1}
send:	$C_{i,i-1}$	$C_{i-1,i-1}$	$C_{i-1,i-1}$	$C_{i-1,i-1}$	$C_{i-1,i-2}$
receive:	$C_{i,-1}$	$C_{i-1,-1}$	$C_{i-1,-1}$	$C_{i-1,-1}$	$C_{i-1,-1}$
	$C_{i-1,-1}$	$C_{i-1,-1}$	$C_{i-1,-1}$	$C_{i-1,-1}$	$C_{i-2,-1}$

Fig. 11. All empty cells between L_i and L_{i-1} send up $C_{i-1,i-1}$ and receive $C_{i-1,-1}$.

	UPWARD SWEEP			DOWNWARD SWEEP		
	receive from LCHILD	receive from RCHILD	send to PARENT	receive from PARENT	send to LCHILD	send to RCHILD
1.	$C_{i,i}$	$C_{i,i}$	$C_{i,i}$	$C_{i,-1}$, $C_{i,-1}$	$C_{i,-1}$, $C_{i,-1}$	$C_{i,-1}$, $C_{i,-1}$
2.	$C_{i,j}$	$C_{j,j}$	$C_{i,j}$	$C_{i,-1}$, $C_{j,-1}$	$C_{i,-1}$, $C_{j,-1}$	$C_{j,-1}$, $C_{j,-1}$
3.	$C_{j,j}$	$C_{j,k}$	$C_{j,k}$	$C_{j,-1}$, $C_{k,-1}$	$C_{j,-1}$, $C_{j,-1}$	$C_{j,-1}$, $C_{j,-1}$
4.	$C_{i,j}$	$C_{j,k}$	$C_{i,k}$	$C_{i,-1}$, $C_{k,-1}$	$C_{i,-1}$, $C_{j,-1}$	$C_{j,-1}$, $C_{k,-1}$

Fig. 12. Four possible situations a T-cell may find itself in. (1) All L-cells below a T-cell are empty. (2) At least one L-cell in the T-cell's left subtree is occupied. (3) At least one L-cell in the T-cell's right subtree is occupied. (4) At least one L-cell in each of the T-cell's subtrees is occupied.

$1 \leqslant i \leqslant n-1$, respectively. All empty L-cells in between should be initialized with $C_{i-1,i-1}$. If so, Lemma 1 guarantees that

$$C_{i,i-1} \cdot C_{i-1,i-1} = C_{i,i-1} \qquad \text{and} \qquad C_{i-1,i-1} \cdot C_{i-1,i-2} = C_{i-1,i-2}$$

During the upward sweep, a T-cell that receives $C_{i-1,i-1}$ from one of its children, in effect, sends to its parent the pair received from the other child. At the end of the downward sweep, the "solution" received by an empty L-cell is the solution of the first nonempty L-cell to its right. Hence, the nonempty L-cells in Fig. 11 all receive $(C_{i-1,-1}, C_{i-1,-1})$. Figure 12 shows 4 possible courses of action a T-cell may take during the upward and downward sweeps, depending upon the status of the L-cells beneath it. It should be emphasized that the status of the L-cells beneath a T-cell is transparent to the T-cell. Figures 13 and 14 show the complete execution of RECUR for $n=5$ on a tree with 8-L-cells.

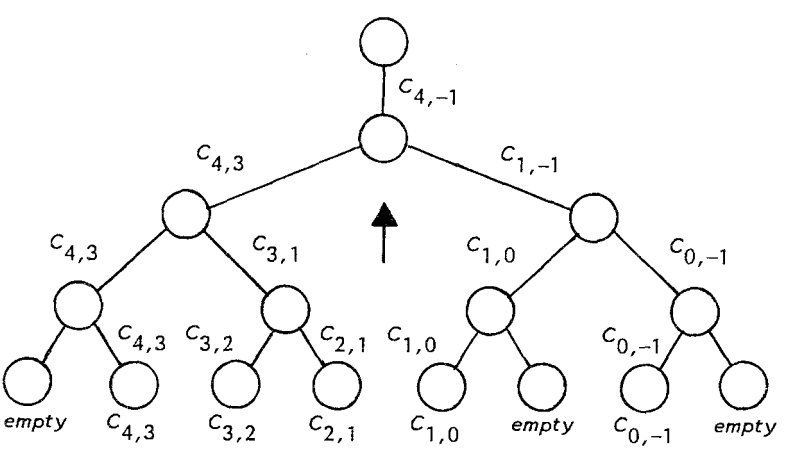

Fig. 13. Upward sweep of RECUR for $n=5$ on an 8-L-cell tree machine.

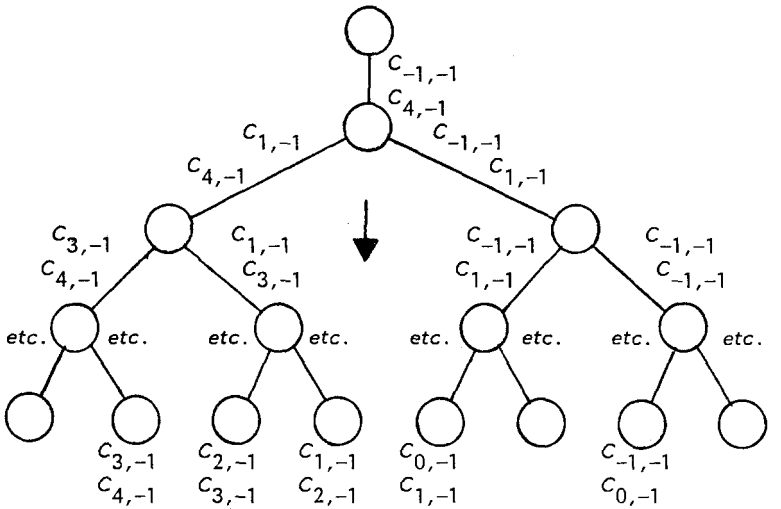

Fig. 14. Downward sweep of RECUR for $n = 5$ on an 8-L-cell tree machine.

5.2. Solving Several Independent Recurrences Simultaneously

RECUR is, in fact, more powerful than so far described. If there are enough L-cells to accommodate the initial values of two or more recurrence expressions of the same type (with one L-cell containing at one initial value of one recurrence) we may solve all recurrences simultaneously.

Let the initial values of two or more recurrences be distributed among the L-cells from right to left. Figure 15 shows two recurrences, one occupying the rightmost 3-L-cells, the other occupying the leftmost 5-L-cells. We may consider the entire set of values to be the initial values of one large recurrence and apply RECUR to all of the L-cells. This is because the first initial value of each recurrence, $C_{0,-1}$, which expresses x_0 as a function of

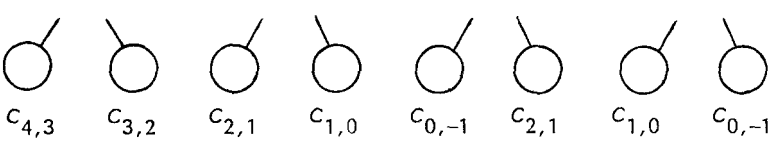

Fig. 15. Distribution of the initial values of two recurrence expressions on a single tree machine. Once recurrence occupies the leftmost five L-cells, the other occupies the rightmost three L-cells. The value $C_{0,-1}$ (in the fifth L-cell from the left) prevents the terms of the recurrence on the left to be affected by the terms of the recurrence on the right.

the dummy variable x_{-1}, disengages each recurrence from every other and we are sure that the terms of one recurrence will not be affected by the terms of another. We may therefore load as many recurrences as the L-cells can hold, apply RECUR, and in a single sweep, solve all recurrences simultaneously.

5.3. Not Enough L Cells

A slight modification of RECUR allows the number of solved terms of the linear recurrence to exceed the number of L-cells. Let n and N $(n > N)$ be the number of terms of the linear recurrence and the number of L-cells, respectively. We distribute the initial values among the L-cells from right to left (as before) and repeat when the leftmost participating L-cell is filled. This continues until we run out of values. Some of the L-cells may be empty, if so required; empty L-cells should be initialized as described in Section 5.1. Figure 16 shows one distribution of $n = 10$ initial values among $N = 8$ L-cells. In general, if all L-cells participate, each L-cell holds $\lceil n/N \rceil$ initial values. The modified cell programs are shown in Fig. 17. It makes use of Hoare's guarded command,[22] a form of a conditional useful when programming communicating processes.

Briefly, each L-cell holds $\text{mas} = \lceil n/N \rceil$ initial values. At the end of execution, each L-cell will have received max pairs of solutions (one pair for each initial value). An L-cell's task is to send the initial values, one at a time, to its parent, and to receive the solution values when they arrive. If all operations are synchronized, the first solution pair should arrive at an L-cell $2(\log N + 1)$ time units after the first initial value is sent. This corresponds to the time required for the first "wave" of initial values to reach the C-cell and back. Let the initial values contained in an L-cell be stored in an array INIT(1: max). The solution values received by an L-cell will be stored in array SOLN(1: max, 1: 2). Line l_5 instructs an L-cell to attempt to send the ith initial value to its parent and if successful, to increment i. Line l_6 instructs an L-cell to attempt to receive two solution values from its parent and if successful, to increment j, Execution ter-

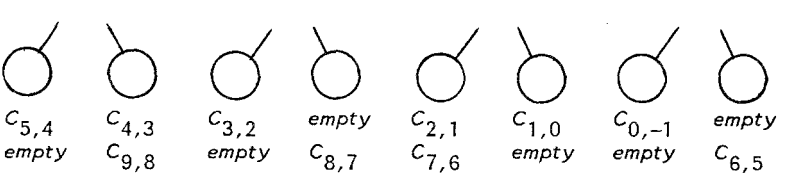

Fig. 16. One possible distribution of the initial values of a recurrence expression when n, the number of terms of the recurrence, exceeds N, the number of L-cells of the tree machine. Empty cells may be scattered throughout.

```
L cell
l1   max=⌈n/N⌉
l2   i=1, j=1
l3   do forever
l4      j>max —> exit
l5      i≤max send (INIT(i)) —> i=i+1
l6      j≤max receive (SOLN(j,1) SOLN(j,2)) —> j=j+1
l7   end
T cell
t1   max=⌈n/N⌉
t2   i=1, j=1, upbuf='empty', downbuf='empty'
t3   do forever
t4      j>max —> exit
t5      i≤max & upbuf='empty' & L.receive(Cleft) & R.receive(Cright)
           —> SAVE(j)=Cright, T1=Cleft•Cright, upbuf='full'
t6      i≤max & upbuf='full' & P.send(T1)
           —> i=i+1, upbuf='empty'
t7      j≤max & downbuf='empty' & P.receive(T1, T3)
           —> downbuf='empty', T2=SAVE(j) • T3
t8      j≤max & downbuf='full' & L.send(T1, T3) & R.send(T2, T3)
           —> j=j+1, upbuf='empty'
t9   end
C cell
c1   Cprev=C-1,-1
c2   do forever
c3      receive (Ccurr)
c4      Ccurr=Ccurr•Cprev
c5      send (Ccurr,Cprev)
c6      Cprev=Ccurr
c7   end
```

Fig. 17. *L*-, *T*-, and *C*-cell programs for RECUR when $n > N$.

minates when j exceeds max, indicating that all solutions have been received.

The *C*-cell's task is to receive a value and to return a pair of values that will trigger the solution of other values. The *C*-cell first initializes the variable *Cprev* to $C_{-1,-1}$. If we assume no empty *L*-cells, the first value the *C*-cell receives (as it executes line c_3 of Fig. 17) is $C_{N-1,-1}$ (Lemma 4). The *C*-cell returns (as it executes line c_5) $C_{N-1,-1}$ and $C_{-1,-1}$. The second value the *C*-cell receives (which is the end product of the second "wave" of values initiated by the *L*-cells) is $C_{2N-1,N-1}$ and returns $C_{2N-1,-1}$ and

$C_{N-1,-1}$. In general, the ith value the C-cell receives is $C_{iN-1,(i-1)N-1}$ and returns $C_{iN-1,-1}$ and $C_{(i-1)N-1,-1}$. Because the entire operation is pipelined, a C-cell is receiving values as fast as it is sending them.

The T-cell program is more complex than either the L- or the C-cell programs. The reason is the need to prevent deadlock. One of the T-cell's two tasks is to receive values from it children, operate on the values, and send the result to its parent. The other task is to receive two solutions from its parent, create a third solution, and send two solutions to each of its children. Lines t_5 through t_8 precisely describe the T-cell's function. Deadlock is prevented through the use of auxiliary buffers. Instructions t_5 through t_8 have the following meanings.

t_5: If the output buffer to the parent (upbuf) is empty and the T-cell successfuly receives values from each of its children, save the value from the right child (for the downward sweep), operate on the values received, and mark *upbuf* full. If the attempt to receive values is unsuccessful, abandon the statement and try one of the other instructions.

t_6: If the output buffer to the parent is full and the T-cell successfully sends a values to its parent, increment i and mark *upbuf* empty. We are now ready to receive another pair of values from the hildren. If the attempt to send is unsuccessful, abandon the statement and try one of the others.

t_7: t_8: Similar to t_5 and t_6 but controlling the downward sweep. Deadlock is prevented through the use of a flag *downbuf*.

It is important that a T-cell be able to abandon a SEND or RECEIVE instruction if it is not successful.

Analysis of this variation of RECUR is straightforward. We may think of waves of values starting from the L-cells, reaching the C-ceel, and returning. Assuming synchrony, the first wave returns to the L-cells $k(\log N + 1)$ time units after it left, for some constant k. From that time on, a new wave arrives at the L-cells after every k time units. Hence, the total time is

$$T(N, n) = k(\log N + (\lceil n/N \rceil - 1)) = O(\log N + \lceil n/N \rceil) \qquad (26)$$

to solve for the first n terms of a recurrence on a tree machine with N L-cells.

6. CONCLUSIONS

The tree structure is a natural tool for solving recurrences. On a tree with N leaf processors (L-cells), the general tree algorithm RECUR solves

the first n terms of a variety of recurrence problems in a single sweep up and down the tree, i.e., in $O(\log N)$ time, provided $n \leqslant N$. If $n > N$, RECUR solves the recurrence in groups of N. Because the operations in RECUR are pipelined, RECUR requires $O(\log N + \lceil n/N \rceil)$ time.

Tree branches have long held the reputation of being bottlenecks when data are moved among the tree processors. Almost counter intuitively, when solving recurrences, the tree branches provide an excellent (and, we believe, the ideal) means for combining and distributing partial results. The efficiency of RECUR is a result of the ease with which data are moved among the tree processors. If $N = n$, each interior tree processor determines exactly one solution value and the total work is evenly divided among, and quickly solved by, the processors.

7. ACKNOWLEDGMENTS

Many thanks to Gyula Magó for his many helpful comments on earlier versions of this paper and for designing the machine that makes algorithms such as RECUR possible.

REFERENCES

1. H. S. Stone, An efficient parallel algorithm for the solution of a tridiagonal linear system of equations. *J. of the ACM* **20**(2):27–38 (1973).
2. P. M. Kogge and H. S. Stone, A parallel algorithm for the efficient solution of a general class of recurrence equations. *IEEE Trans. on Computers* **C-22**(8):786–793 (1973).
3. P. M. Kogge, Parallel solution of recurrence problems. *IBM J. of Res. and Develop.* **18**(2):138–148 (1974).
4. P. M. Kogge, Maximal rate pipelined solutions to recurrence problems. In *Proc. of the First Annual Symp. on Computer Arch.*, G. J. Lipovski and S. A. Szygenda, (eds.), Gainesville, Florida, 71–76 (1973).
5. S. C. Chen, Time and parallel processor bounds for linear recurrence systems with constant coefficients. In *Proc. of the Inter. Conf.*, P. H. Enslow Jr. (ed.), 196–205.
6. S. C. Chen and D. J. Kuck, Time and parallel processor bounds for linear recurrence systems. *IEEE Trans. on Computers* **C-24**(7):701–717 (1975).
7. S. C. Chen and A. H. Sameh, On parallel triangular system solvers. *Proc. of the Sagamore Computer Conf. on Parallel Proc.*, 237–238 (1975).
8. S. C. Chen, D. J. Kuck, and A. H. Sameh, Practical parallel band triangular system solvers. *ACM Trans. on Mathematical Software* **4**(3):270–277 (1978).
9. D. D. Gajski, An algorithm for solving linear recurrence systems on parallel and pipelined machines. *IEEE Trans. on Computers* **C-30**(4):190–206 (1981).
10. L. Hyafil and H. T. Kung, The complexity of parallel evaluation of linear recurrences. *J. of the ACM* **24**(3):513–521 (1977).
11. G. A. Magó, A network of microprocessors to execute reduction languages. Two parts. *Inter. J. of Computer and Infor. Sci.* **8**(5):349–385 (1979); **8**(6):435–471 (1979).

12. G. A. Magó, A cellular computer architecture for functional programming. *Spring COMPCON. VLSI: New Architectural Horizons*, 179–185 (1980).
13. J. Backus, Can Programming be liberated from the von Neumann style? A functional style and its algebra of programs. *Comm. of the ACM* **21**(8):613–641 (1978).
14. D. M. Tolle, 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).
15. E. H. Williams Jr., Analysis of FFP programs for parallel associative searching. Ph.D. dissertation, Department of Computer Science, University of North Carolina at Chapel Hill, (1981).
16. G. A. Frank, Virtual memory systems for closed applicative language interpreters. Ph.D. dissertation, Department of Computer Science, University of North Carolina at Chapel Hill, (1979).
17. A. Koster, 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).
18. G. A. Magó, D. F. Stanat, and A. Koster, Program execution on a cellular computer: some matrix algorithms (In Preparation).
19. S. A. Browning, Computations on a tree of processors. *Proc. of the Caltech Conf. on VLSI*, 453–478 (January 1979); also in Chap. 8 of *Intro. to VLSI Systems*, by C. Mead and L. Conway, Addison-Wesley, Reading, Massachusetts, (1980).
20. J. L. Bentley and H. T. Kung, A tree machine for searching problems. Proc. of the *Inter. Conf. on Parallel Processing*, 257–266 (1979).
21. C. E. Leiserson, Systolic priority queues. *Proc. of the Caltech Conf. on VLSI*, 199–214 (January 1979).
22. C. A. R. Hoare, Communicating Sequential Processes. *Comm. of the ACM* **21**(8):666–677 (1978).