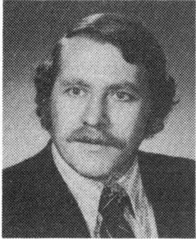
[43] J. S. Tyler, J. D. Powell, and R. K. Mehra, "The use of smoothing and other advanced techniques for VTOL aircraft parameter identification," Cornell Aeronaut. Lab., Final Rep., Naval Air Syst. Command Contract N00019-69-6-0534, June 1970.

[44] S. R. McReynolds, "Parallel filtering and smoothing algorithms," presented at the 3rd Symp. Nonlinear Estimation Theory, San Diego, Calif., Sept. 1972.



Robert E. Larson (S'58–M'66–F'73) was born in Stockton, Calif., on September 19, 1938. He received the B.S. degree from the Massachusetts Institute of Technology, Cambridge, in 1960, and the M.S. and Ph.D. degrees from Stanford University, Stanford, Calif., in 1961 and 1964, respectively, all in electrical engineering.

He has been employed by the IBM Corporation and the Hughes Aircraft Company. From 1964 to 1968 he was with the Information and Control Laboratory, Stanford Research Institute, Menlo Park, Calif. In 1968, he and two colleagues founded Systems Control, Inc., Palo Alto, Calif., where he is currently Executive Vice President. He is the author of *State Increment Dynamic Programming* (New York: Elsevier, 1968) and of over 90 technical papers. His fields of specialization are computational aspects of dynamic programming and applications of optimal control and estimation theory.

Dr. Larson received the IEEE Group on Automatic Control Best Paper Award in 1965, and the 1968 Donald P. Eckman Award for outstanding achievement in the field of automatic control from the American Automatic Control Council. He was also named the Outstanding Young Electrical Engineer for 1969 by Eta Kappa Nu. He has served the IEEE Control Systems Society in a number of capacities, and is presently the Vice President of the Society, Chairman of its Information Dissemination Committee, and an elected AdCom member. He will also be Program Chairman for the 1973 Joint Automatic Control Conference.

Edison Tse (M'70) was born in Kwangtung, China, on January 21, 1944. He received the B.S. and M.S. degrees simultaneously in 1967, and the Ph.D. degree in 1970, all in electrical engineering, from the Massachusetts Institute of Technology, Cambridge.

From 1966 to 1967 and from 1968 to 1969 he was a Teaching Assistant at M.I.T., teaching graduate courses in stochastic system and optimal control theory, and from 1967 to 1968 he was a Research Assistant at M.I.T., doing research in nonlinear filtering. From 1968 to 1969 he was also a consultant for the State Street Bank of Boston, where he applied optimal control to banking problems. Since 1969 he has been with Systems Control, Inc., Palo Alto, Calif., where he is now a Senior Research Engineer. His current research interests include optimal control theory, dynamic allocation, and stochastic estimation, identification and control.

Dr. Tse is a member of Sigma Xi, Eta Kappa Nu, and Tau Beta Pi.

A Parallel Algorithm for the Efficient Solution of a General Class of Recurrence Equations

PETER M. KOGGE AND HAROLD S. STONE

***Abstract*—An mth-order recurrence problem is defined as the computation of the series $x_1, x_2, \cdots, x_N$, where $x_i = f_i(x_{i-1}, \cdots, x_{i-m})$ for some function f_i. This paper uses a technique called recursive doubling in an algorithm for solving a large class of recurrence problems on parallel computers such as the Illiac IV.**

Recursive doubling involves the splitting of the computation of a function into two equally complex subfunctions whose evaluation can be performed simultaneously in two separate processors. Successive splitting of each of these subfunctions spreads the computation over more processors.

This algorithm can be applied to any recurrence equation of the form $x_i = f(b_i, g(a_i, x_{i-1}))$ where f and g are functions that satisfy certain distributive and associative-like properties. Although this recurrence is first order, all linear mth-order recurrence equations can be cast into this form. Suitable applications include linear recurrence equations, polynomial evaluation, several nonlinear problems, the determination of the maximum or minimum of N numbers, and the solution of tridiagonal linear equations. The resulting algorithm computes the entire series $x_1, \cdots, x_N$ in time proportional to $\lceil \log_2 N \rceil$ on a computer with N-fold parallelism. On a serial computer, computation time is proportional to N.

***Index Terms*—Parallel algorithms, parallel computation, recurrence problems, recursive doubling.**

Manuscript received October 7, 1972; revised March 21, 1973. This work was supported in part by NSF Grant GJ 1180 and in part by an IBM Corporation fellowship.

P. M. Kogge was with the Department of Electrical Engineering, Digital Systems Laboratory, Stanford University, Stanford, Calif. He is now with the Systems Architecture Department, IBM Corporation, Owego, N.Y. 13827.

H. S. Stone is with the Department of Electrical Engineering and the Department of Computer Science, Digital Systems Laboratory, Stanford University, Stanford, Calif.

INTRODUCTION

A. *Definition of Problem*

It FREQUENTLY occurs in applied mathematics that the solution to some problem is a sequence $x_1, x_2, \cdots, x_N$, where each x_i is a function of the previous m x's, namely $x_{i-1}, \cdots, x_{i-m}$. A common example of such a problem is a time-varying linear system, where the state of the system at time i is x_i, and can be computed from the equations

$$\begin{aligned}x_1 &= B_1\\ x_2 &= A_2 x_1 + B_2\\ x_3 &= A_3 x_2 + B_3\\ &\vdots\\ x_i &= A_i x_{i-1} + B_i\\ &\vdots\\ x_N &= A_N x_{N-1} + B_N\end{aligned} \tag{1}$$

where A_i and B_i represent the internal dynamics of the system. A_i and B_i can be real or complex numbers, constant or time-varying matrices, etc., depending on the problem.

The equation used to compute x_i is called a recurrence equation and, together with some initial values for some of the x_i, represents a complete problem description. Formally, a recurrence problem consists of a set of recurrence equations:

$$x_i = f_i(x_{i-1}, \cdots, x_{i-m}), \qquad i = m+1, \cdots, N \tag{2}$$

and some boundary values, which may consist of the following.

1) $x_1, \cdots, x_m$. This is an initial value problem.
2) $x_{N-m+1}, \cdots, x_N$. This is a final value problem.
3) A mixture of m initial and final values.

This paper discusses an algorithm for solving a particular class of initial value recurrence problems on parallel computing systems such as the Illiac IV. This class of problems includes the computation of the sequence $x_1, \cdots, x_N$ when the expression for x_i is a linear recurrence equation of the form of (1), the calculation of the maximum or minimum of N numbers, the evaluation of Nth-degree polynomials, and several nonlinear problems. Such problems as these can be solved in a very straightforward manner on serial processors in time proportional to N. Some have also been solved on parallel computers with special-purpose algorithms tailored to those problems, e.g., polynomial evaluation (Munro and Paterson [3]). With a computer having N-fold parallelism, the algorithm in this paper solves all these problems and others in time proportional to $\lceil \log_2 N \rceil$.[1]

B. Computer Model

The algorithm to be described in this paper is designed for a computer of the Illiac IV class. The major assumptions about the computer's architecture are as follows.

Assumption 1: There are p identical processors, each able to execute the usual arithmetic and logical operations, and each with its own memory.

Assumption 2: Each processor can communicate with every other processor. The exact method of data exchange between processors can affect the algorithm's computational complexity and will be discussed in a future report.

[1] $\lceil x \rceil$ is the ceiling function and represents the smallest integer not smaller than x.

Assumption 3: Each processor has a distinct index by which it is referenced.

Assumption 4: All processors obtain their instructions simultaneously from a single instruction stream. Thus all processors execute the same instruction, but they operate on data stored in their own memories.

Assumption 5: Any processor may be "blocked" or "masked" from performing some instruction. This mask may be set by an explicit instruction directed to that processor via its index, or by the result of some test instruction such as "set mask if accumulator = 0."

Assumption 6: Elementary arithmetic operations have two operands.

It is assumed throughout this paper that the number of processors p is greater than N, the maximum number of elements to be computed. In reality when p is less than N, this algorithm can be used $\lceil N/p \rceil$ times to calculate p elements of the series at a time.

II. General First-Order Recurrence Equation

A. Example

In this section we develop a parallel solution to a simple first-order recurrence problem. The solution is a special case of the general algorithm, but its development is not obscured by the notation needed to describe the general algorithm.

Given $x_1 = b_1$, find $x_2, \cdots, x_N$, where

$$x_i = a_i x_{i-1} + b_i. \tag{3}$$

Before solving this problem let us give the following definition.

Definition: The function $\hat{Q}(m, n)$, $n \leqslant m$, is defined to be

$$\hat{Q}(m, n) = \sum_{j=n}^{m} \left(\prod_{r=j+1}^{m} a_r \right) b_j$$

where the vacuous product $(\Pi_{r=m+1}^{m} a_r)$ is given the value 1. Stone [4] first used this notation in the derivation of this algorithm. The basic algorithm involves a concept called *recursive doubling*, which consists of breaking the calculation of one term into two equally complex subterms.

Now we can write the solution to (3) as follows:

$$\begin{aligned}x_1 &= b_1 &&= \hat{Q}(1,1)\\ x_2 &= a_2 x_1 + b_2 &&= a_2 b_1 + b_2 = \hat{Q}(2,1)\\ x_3 &= a_3 x_2 + b_3 &&= a_3 a_2 b_1 + a_3 b_2 + b_3 = \hat{Q}(3,1)\\ &\vdots\\ x_i &= a_i x_{i-1} + b_i &&= \hat{Q}(i,1)\\ &\vdots\\ x_N &= a_N x_{N-1} + b_N = \hat{Q}(N,1).\end{aligned} \tag{4}$$

We can also write this solution as

$$\hat{Q}(1,1) = x_1 = b_1$$

$$\hat{Q}(2,1)=x_2=a_2x_1+b_2=a_2\hat{Q}(1,1)+\hat{Q}(2,2)$$

$$\vdots$$

$$\begin{aligned}\hat{Q}(4,1)=x_4&=a_4x_3+b_4\\&=a_4a_3x_2+a_4b_3+b_4\\&=a_4a_3(a_2b_1+b_2)+(a_4b_3+b_4)\\&=a_4a_3\hat{Q}(2,1)+\hat{Q}(4,3)\end{aligned}$$

$$\vdots$$

In general

$$\hat{Q}(2i,1)=x_{2i}=\left(\prod_{r=i+1}^{2i}a_r\right)\hat{Q}(i,1)+\hat{Q}(2i,i+1). \quad (5)$$

Equation (5) gives us our recursive doubling. Both $\hat{Q}(i,1)$ and $\hat{Q}(2i,i+1)$ are identical in structure since they both require the same number and sequence of multiplications and additions. Also, each of these terms involves i a's and i b's, exactly one-half the number of a's and b's used in $\hat{Q}(2i,1)$. Thus if at the kth step we want to compute x_{2i}, then at the $k-1$st step we should have one processor compute $\hat{Q}(i,1)$ and another compute $\hat{Q}(2i,i+1)$. We then continue this splitting operation recursively. The resulting computation graph for the case $N=8$ is given in Fig. 1.

Note that when we compute $\hat{Q}(2i,1)$ from the two equally complex subterms $\hat{Q}(i,1)$ and $\hat{Q}(2i,i+1)$, we also need the additional product $(\Pi_{r=i+1}^{2i}a_r)$. This is not a serious hindrance since we can compute the products using the scheme shown in Fig. 2. We see that in all cases the correction products needed at one level of the tree in Fig. 1 are always available just after the previous level in Fig. 2. Figs. 1 and 2 show the computation of $\hat{Q}(8,1)$. However, it is straightforward to extend the computation to eight processors, and compute $\hat{Q}(i,1)$ for $1\leqslant i\leqslant 8$ in parallel. The algorithm solves (3) in a time proportional to $\lceil\log_2 N\rceil$.

An example of the complete solution of (3) for the case $N=8$ is given in detail in Table I.

B. A General Class of First-Order Recurrence Equations

In this section we define a general class of first-order recurrence equations for which we develop a parallel algorithm. The limitation to first-order equations is not as restrictive as it might first appear, since it is often the case that we can very easily reformulate a more general mth-order problem as a first-order problem. Section III-B describes such a reformulation.

The general parallel algorithm developed in Section II-C solves all recurrence equations that can be placed in the following form:

$$\begin{aligned}x_1&=b_1\\x_i&=f_i(x_{i-1})=f(b_i,g(a_i,x_{i-1})), \quad 2\leqslant i\leqslant N \quad (6)\end{aligned}$$

where b_i and a_i are arbitrary constants and f and g are index-independent functions that satisfy the following restrictions.

Restriction 1: f is associative. $f(x,f(y,z))=f(f(x,y),z)$.

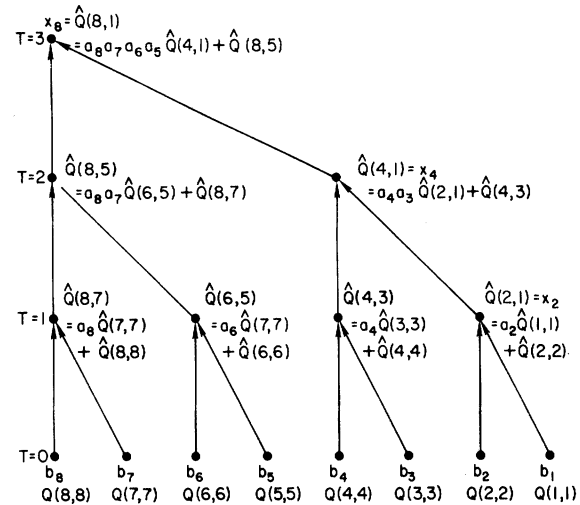


Fig. 1. Parallel computation of x_8 in the sequence $x_i=a_ix_{i-1}+b_i$.

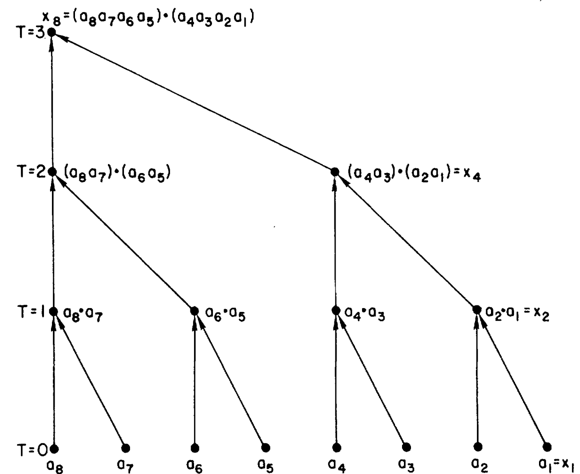


Fig. 2. Parallel computation of $x_8=\Pi_{j=1}^{8}a_j$.

Restriction 2: g distributes over f. $g(x,f(y,z))=f(g(x,y),g(x,z))$.

Restriction 3: g is semiassociative, that is, there exists some function h such that $g(x,g(y,z))=g(h(x,y),z)$.

The previous restrictions on f and g are the only ones necessary to prove the correctness of the general parallel algorithm. However, these restrictions may also limit the domains from which a_i and b_i and the variables x_i can be chosen. For most normal arithmetic operators like + or · there is no problem, but more exotic operations such as floor, ceiling, modulo division, etc., may constrain the permissible domains and should be checked carefully.

The semiassociative property of g forces h to behave as if it were associative. In particular, we have

$$\begin{aligned}g(h(h(a,b),c),d)&=g(h(a,b),g(c,d))\\&=g(a,g(b,g(c,d)))\\&=g(a,g(h(b,c),d))\\&=g(h(a,h(b,c)),d).\end{aligned}$$

Hence, iterated compositions of h when used as the first argument of the function g can be evaluated as if h were as-

TABLE I

Processor	$T=0$ $A(i)$	$T=0$ $B(i)$	$T=1$ $A(i)$	$T=1$ $B(i)$	$T=2$ $A(i)$	$T=2$ $B(i)$	$T=3$ $A(i)$	$T=3$ $B(i)$
1	**	$b_1 = X_1 = \hat{Q}(1,1)$	**	$X_1 = \hat{Q}(1,1)$	**	$X_1 = \hat{Q}(1,1)$	**	X_1
2	a_2	$b_2 = \hat{Q}(2,2)$	a_2	$a_2x_1 + b_2 = x_2 = \hat{Q}(2,1)$	a_2	$X_2 = \hat{Q}(2,1)$	a_2	X_2
3	a_3	$b_3 = \hat{Q}(3,3)$	a_3a_2	$a_3b_2 + b_3 = \hat{Q}(3,2)$	a_3a_2	$(a_3a_2)X_1 + (a_3b_2 + b_3) = X_3 = \hat{Q}(3,1)$	a_3a_2	X_3
4	a_4	$b_4 = \hat{Q}(4,4)$	a_4a_3	$a_4b_3 + b_4 = \hat{Q}(4,3)$	$a_4a_3a_2$	$(a_4a_3)X_2 + (a_4b_3 + b_4) = X_4 = \hat{Q}(4,1)$	$a_4a_3a_2$	X_4
5	a_5	$b_5 = \hat{Q}(5,5)$	a_5a_4	$a_5b_4 + b_5 = \hat{Q}(5,4)$	$a_5a_4a_3a_2$	$a_5a_4(a_3b_2 + b_3) + a_5b_4 + b_5 = \hat{Q}(5,2) = \sum_{w=2}^{5}\left(\prod_{m=w+1}^{5} a_m\right) b_w$	$\prod_{m=2}^{5} a_i^*$	X_5
6	a_6	$b_6 = \hat{Q}(6,6)$	a_6a_5	$a_6b_5 + b_6 = \hat{Q}(6,5)$	$a_6a_5a_4a_3$	$\sum_{w=3}^{6}\left(\prod_{m=w+1}^{6} a_m\right) b_w = \hat{Q}(6,3)$	$\prod_{m=2}^{6} a_i^*$	X_6
7	a_7	$b_7 = \hat{Q}(7,7)$	a_7a_6	$a_7b_6 + b_7 = \hat{Q}(7,6)$	$a_7a_6a_5a_4$	$\sum_{w=4}^{7}\left(\prod_{m=w+1}^{7} a_m\right) b_w = \hat{Q}(7,4)$	$\prod_{m=2}^{7} a_i^*$	X_7
8	a_8	$b_8 = \hat{Q}(8,8)$	a_8a_7	$a_8b_7 + b_8 = \hat{Q}(8,7)$	$a_8a_7a_6a_5$	$\sum_{w=5}^{8}\left(\prod_{m=w+1}^{8} a_m\right) b_w = \hat{Q}(8,5)$	$\prod_{m=2}^{8} a_i^*$	X_8

* Not really needed to compute $X_1, \cdots, X_8$.
** Arbitrary.

sociative without altering the output value of g. In all interesting practical problems discovered thus far, the function h is associative.

C. Parallel Algorithm

The principle of recursive doubling can be applied in a natural way to any recurrence equation that satisfies the restrictions of Section II-B. In fact, the resulting general algorithm bears a very strong resemblance to the example of Section II-A. Before giving the algorithm, however, we first give two definitions.

Definition: For any function q of two arguments define the generalized composition of q as $q_{j=n}^{(m)}(a_j)$, where

$$q_{j=n}^{(n)}(a_j) = a_n, \qquad \text{for } n \geq 1$$

$$q_{j=n}^{(m)}(a_j) = q(a_m, q_{j=n}^{(m-1)}(a_j)), \qquad \text{for } m > n \geq 1.$$

$$= q(a_m, q(a_{m-1}, \cdots, q(a_{n+2}, q(a_{n+1}, a_n)) \cdots).$$

If we let $q(a,b) = a + b$ (scalar addition), then

$$q_{j=n}^{(m)}(a_j) = (a_m + (a_{m-1} + \cdots + (a_{n+2} + (a_{n+1} + a_n)) \cdots)$$

$$= \sum_{j=n}^{m} a_j.$$

Likewise, if $q(a,b) = a \cdot b$ (scalar multiplication), then

$$q_{j=n}^{(m)}(a_j) = \prod_{j=n}^{m} a_j.$$

Definition: Define $Q(m,n)$ as

$$Q(m,n) = f_{j=n}^{(m)}(g[h_{r=j+1}^{(m)}(a_r), b_j]), \qquad m \geq n \geq 1$$

where we define

$$g(h_{r=m+1}^{(m)}(a_r), b_j) = b_j.$$

If we consider the case where f is scalar addition, and g and h are scalar multiplication, then the $Q(m,n)$ defined previously is exactly the same as the $\hat{Q}(m,n)$ defined for the example in Section II-A.

The similarities between Q and $\hat{Q}$ carry even further. The function $Q(i,1)$ is the solution of the general recurrence equation (6), that is,

$$x_i = Q(i,1), \qquad \forall\ 1 \leq i \leq N. \tag{7}$$

Also, as in the example, we can derive a formula computing $Q(2i,1)$ strictly in terms of two equally complex subterms, namely,

$$Q(2i,1) = f(Q(2i,i+1), g(h_{j=i+1}^{(2i)}(a_j), Q(i,1))). \tag{8}$$

Both (7) and a more general version of (8) are proved in the Appendix.

Equation (8) is a perfect candidate for recursive doubling. $Q(2i,i+1)$ and $Q(i,1)$ are identical in terms of the number of unique a's and b's referenced and require the same sequence of f, g, and h function calls to evaluate them. As with the second example, the only hindrance in implementing (8) directly as a recursive doubling algorithm is the correction term, the h composition. However, since h can be treated as an associative function, we can use a scheme similar to Fig. 2 to compute these correction terms exactly as they are needed.

Fig. 3 is a computation graph using (8) and the h composition algorithm to compute x_8. Despite its increased complexity, the general structure of this graph is identical to Figs. 1 and 2 and can be extended to solve for all elements of the sequence $x_1, \cdots, x_N$ in parallel.

We can now state the complete algorithm for solving our general recurrence equations. The detailed proof of the correctness of this algorithm is given in the Appendix.

Algorithm A–General Algorithm: This algorithm solves for $x_1, x_2, \cdots, x_N$ where $x_i = f(b_i, g(a_i, x_{i-1}))$ and f and g satisfy the restrictions of Section II-B.

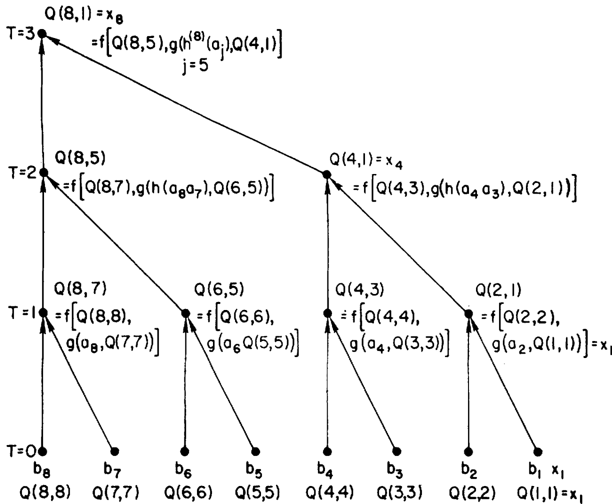


Fig. 3. Parallel computation of x_8 from the general recurrence equation.

The algorithm requires two vectors A and B of N elements. The ith component of each vector, namely $A(i)$ and $B(i)$, is stored in the memory of processor (i). The actual data structure required to represent $A(i)$ and $B(i)$ depends on the definition of the domain of the entities a_i and b_i in the basic equation (8) and may be scalars, matrices, lists, etc., depending on the problem.

Let $A^{(k)}(i)$ and $B^{(k)}(i)$ represent respectively, the contents of $A(i)$ and $B(i)$ after the kth step of the following algorithm.

Initialization Step (k = 0):

$B^{(0)}(i) = b_i$ for $1 \leqslant i \leqslant N$.

$A^{(0)}(i) = a_i$ for $1 < i \leqslant N$.

$A(1)$ is never referenced and may be initialized arbitrarily.

Recursion Steps: For $k = 1, 2, \cdots \lceil \log_2 N \rceil$ do each of the following assignment statements:

$$B^{(k)}(i) = f(B^{(k-1)}(i), g(A^{(k-1)}(i), B^{(k-1)}(i - 2^{k-1}))), \quad \text{for } 2^{k-1} < i \leqslant N. \tag{9}$$

$$A^{(k)}(i) = h(A^{(k-1)}(i), A^{(k-1)}(i - 2^{k-1})), \quad \text{for } 2^{k-1} + 1 < i \leqslant N. \tag{10}$$

Each statement is assumed to be evaluated simultaneously by all processors whose indices lie in the specified interval.

After the $\lceil \log_2 N\text{th} \rceil$ step, $B(i)$ contains x_i for $1 \leqslant i \leqslant N$.

End of Algorithm A.

Several things should be noted about any implementation of Algorithm A. First, when the ith processor executes (9) and (10) in that order, it must have the old values of $B(i - 2^{k-1})$ and $A(i - 2^{k-1})$, which can only be obtained from processor $(i - 2^{k-1})$. Thus at the beginning of the kth recursion step, all processors must shift their values of A and B to the processors with index 2^{k-1} greater than their own. Exactly how this data routing is performed depends on the processor interconnection pattern available in a given computer system.

Another problem with implementing Algorithm A lies in limiting the processors that execute (9) and (10) to just those with the proper indices. The masking feature (Section I-B) is the most direct way. This, however, requires executing explicit mask instructions during each recurrence step. If the number of available processors is greater than about $3N/2$, another method can avoid these extra instructions. The N processors with the highest indices are allocated to the solving of $x_1, \cdots, x_N$, and the next $N/2$ processors are initialized so that when one of the top N processors references their data, the values returned cause no change in the higher processor's values for $A(i)$ and $B(i)$. These bottom $N/2$ processors are completely masked off initially so that these initial values never change. These initial values are

$$A(i) = I, \quad \text{for } -N/2 \leqslant i \leqslant 1$$
$$B(i) = Z, \quad \text{for } -N/2 \leqslant i \leqslant 0$$

where for all a and b, $h(a, I) = a$ and $f(b, g(a, Z)) = b$. For the example of Section II-A, I is simply 1, and Z is 0.

III. Applications

A. Various First-Order Problems

As has been mentioned before, Algorithm A is applicable to a rather wide class of problems. Table II gives a collection of such problems that satisfy the functional constraints stated in earlier sections.

An interesting case occurs when we constrain all the a_i of Example 1 in Table II to be the same number z as indicated in Example 5 in Table II. We then get the recursion

$$x_i = zx_{i-1} + b_i$$

which, if we solve for x_N, yields

$$x_N = b_1 z^{N-1} + b_2 z^{N-2} + \cdots + b_{N-1} z + b_N.$$

But this is simply the evaluation of the polynomial $b_1 x^{N-1} + \cdots + b_N$ at $x = z$. In fact, Algorithm A in this case is simply the parallel evaluation of polynomials (Munro and Paterson [3]).

B. Extension to mth-Order Equations

The algorithm given in the previous sections is applicable to a class of first-order recurrence equations. However, a little manipulation of the description of a problem can often convert an mth-order recurrence equation into a first-order equation with a slightly more complicated data structure. The clue to how this is done can be found in the third example in Table II, a matrix or "state variable" problem.

As an example, consider the problem

$$x_i = a_{i,1} x_{i-1} + \cdots + a_{i,m} x_{i-m} + b_i. \tag{11}$$

We wish to reformulate it in a form amenable to Algorithm A.

The first step is to see that we can collapse the m x's that are needed in (11) into a single new "variable" by using state variable notation as follows.

Let

$$Z_i = \begin{bmatrix} x_i \\ \cdot \\ \cdot \\ \cdot \\ x_{i-m+1} \end{bmatrix}. \tag{12}$$

Now we can rewrite (11) as

TABLE II
APPLICATIONS OF ALGORITHM A

Example	Domain of b	Domain of a	Domain of x	$f(a, b)$	$g(a, b)$	$h(a, b)$	Comments
1)		real numbers		$a + b$	$a \cdot b$	$a \cdot b$	$x_{i+1} = b_{i+1} + a_{i+1}x_i$
2)		real numbers		$a \cdot b$	$b \uparrow a$	$a \cdot b$	$x_{i+1} = b_{i+1} \cdot (x_i \uparrow a_{i+1})$, "↑" is exponentiation
3)	$m \times 1$ matrix	$m \times m$ matrix	$m \times 1$ matrix	vector addition	mult. of matrix by vector	matrix mult.	$x_{i+1} = B_{i+1} + A_{i+1}x_i$ where A is $m \times m$ and x, B are $m \times 1$
4)		real numbers		b	min (a, b)	min (a, b)	x_i is the smallest of $a_1, \cdots, a_i$
5)		real numbers		b	max (a, b)	max (a, b)	x_i is the largest of $a_1, \cdots, a_i$
6)	real number	any real number z	real number	$a + b$	$a \cdot b$	$a \cdot b$	$x_i = x_{i-1} \cdot z + b_i$, polynomial evaluation $x_N = P(z) = b_1 z^{N-1} + b_2 z^{N-2} + \cdots + b_{N-1} z + b_N$

$$Z_i = \begin{bmatrix} a_{i,1} \cdots a_{i,m} \\ 1\,0 \cdots 0 \\ 0\,1\,0 \cdots 0 \\ 0 \cdots 0\,1\,0 \end{bmatrix} \begin{bmatrix} x_{i-1} \\ \cdot \\ \cdot \\ \cdot \\ x_{i-m} \end{bmatrix} + \begin{bmatrix} b_i \\ 0 \\ \cdot \\ \cdot \\ \cdot \\ 0 \end{bmatrix} \tag{13}$$

$$= \hat{A}_i Z_{i-1} + \hat{B}_i \tag{14}$$

where $\hat{A}_i$ and $\hat{B}_i$ are the $m \times m$ matrix and $m \times 1$ vector respectively. The first row of $\hat{A}_i$ represents the original (11) and the remaining rows simply select the proper x_j to make Z_i be consistent.

Equation (14), however, is in exactly the right format for Example 3 of Table II to be applied. The variables in the recursion are m-element vectors, the $\hat{A}_i$ are $m \times m$ matrices, and the B_i are m-element vectors. The function f is vector addition, g is multiplication of a matrix by a vector, and h is matrix multiplication. Thus if we rewrite (11) into (14) we can apply Algorithm A to get a parallel solution to the original problem (11).

This particular formulation, however, is not very efficient in its use of the parallel processors. At the end of the calculation we have N m-element vectors $Z_1, \cdots, Z_N$. Only one mth of each Z_i, namely its first component x_i, represents new calculations not available from previous Z's. Most of the matrix calculations done in the recurrence steps are redundant.

We can increase the amount of parallelism in the problem by propagating (14) forward m steps before using Algorithm A. This results in a new formulation of the problem, which yields $Z_{(k+1)m} = (x_{(k+1)m}, \cdots, x_{km+1})'$ directly from $Z_{km} = (x_{km}, \cdots, x_{(k-1)m+1})'$.

It is easy to show by induction that Z_{km+m} can be computed as follows:

$$Z_{km+m} = \left(\prod_{j=km+1}^{km+m} \hat{A}_j \right) Z_{km} + \sum_{r=km+1}^{km+m} \left(\prod_{j=r+1}^{km+m} \hat{A}_j \right) \hat{B}_r, \quad k = 1, \cdots, N/m - 1. \tag{15}$$

This equation can be restated in a form directly usable by Algorithm A as follows.

Let

$$X_{k+1} = Z_{(k+1)m} = \begin{bmatrix} x_{(k+1)m} \\ \cdot \\ \cdot \\ \cdot \\ x_{km+1} \end{bmatrix}$$

$$A_{k+1} = \left(\prod_{j=km+1}^{(k+1)m} \hat{A}_j \right) \quad B_{k+1} = \sum_{r=km+1}^{(k+1)m} \left(\prod_{j=r+1}^{(k+1)m} \hat{A}_j \right) \hat{B}_r. \tag{16}$$

Now (15) becomes

$$X_{k+1} = A_{k+1} X_k + B_{k+1}, \quad k = 1, \cdots, N/m \tag{17}$$

which again is our familiar first-order linear matrix recurrence equation.

Now to compute all N elements of (11), we need only compute N/m elements of the series $X_1, \cdots, X_{N/m}$ using (17). Using Algorithm A we can compute these N/m elements with $\lceil \log_2 N/m \rceil$ applications of the recurrence step, plus some initial time to compute the initial A's and B's given by (16). Further, since there are only N/m elements to compute, Algorithm A also calls for only N/m processors.

The important aspect of this reformulation is not that the number of steps has been reduced, but that the number of processors has dropped. Equation (17) takes $\lceil \log_2 m \rceil$ fewer recurrence iterations to evaluate than does (14), but about $\lceil \log_2 m \rceil$ additional iterations are required to set up (17) from (14) with N processors. Thus we have not reduced the time to solve the problem, but we have reduced redundant computations to the point where we need only N/m processors after the initial setup.

IV. Summary and Conclusion

Various researchers have developed parallel algorithms for specific problems, such as polynomial evaluation (Munro and Paterson [3]), and the solution of tridiagonal systems of equations (Buneman [1], Buzbee *et al.* [2], and Stone [4]). As with Algorithm A, these algorithms typically require execution times proportional to $\lceil \log_2 N \rceil$. None of them, however, is applicable to any wider class of problems than the particular ones they were designed to solve. Algorithm A, on the other hand, solves any problem for which the solution can be stated in terms of a recurrence equation satisfying a few simple restrictions. It is worthwhile mentioning that the running time for Algorithm A can vary widely from problem to problem

even though the time is always proportional to $\lceil \log_2 N \rceil$. The constant of proportionality depends on the time it takes to evaluate f, g, and h. These functions can be as simple as a magnitude comparison or floating-point addition and can be as complex as a matrix multiplication, with very large differences in their respective constants of proportionality.

The power of Algorithm A comes from the generalization of the technique of recursive doubling. This technique seems to hold an important key to understanding exactly how parallelism can be extracted from what appear to be highly serial problems. The major results of this paper indicate that the class of serially stated problems that are amenable to parallel solutions is a large one, and includes some problems that have been thought to be poorly suited to parallel processors.

Appendix
Validity of Algorithm A

This Appendix contains some basic theorems that establish the validity of Algorithm A. We assume we are solving equations of the form of (6), where the functions f, g, and h all satisfy the restrictions of Section II-B. We also assume that the concept of generalized composition and the definition of the function $Q(m, n)$ carry over from Section II-C.

Theorem 1: For any i, k such that $1 \leqslant k < i \leqslant N$ then for any j such that $1 \leqslant j \leqslant k$

$$Q(i, i-k) = f(Q(i, i-j+1), g(h_{r=i-j+1}^{(i)}(a_r), Q(i-j, i-k))).$$

Proof: Assume $1 \leqslant j \leqslant k$. Then

$$f(Q(i, i-j+1), g(h_{r=i-j+1}^{(i)}(a_r), Q(i-j, i-k)))$$
$$= f(Q(i, i-j+1), g(h_{r=i-j+1}^{(i)}(a_r), f_{m=i-k}^{(i-j)}(g(h_{r=m+1}^{(i-j)}(a_r), b_m)))) \quad \text{[by definition of } Q(i-j, i-k)]$$
$$= f(Q(i, i-j+1), f_{m=i-k}^{(i-j)}(g(h_{r=i-j+1}^{(i)}(a_r), g(h_{r=m+1}^{(i-j)}(a_r), b_m)))) \quad [g \text{ distributes over } f]$$
$$= f(Q(i, i-j+1), f_{m=i-k}^{(i-j)}(g(h_{r=m+1}^{(i)}(a_r), b_m))) \quad [g \text{ is semiassociative}]$$
$$= f(f_{m=i-j+1}^{(i)}(g(h_{r=m+1}^{(i)}(a_r), b_m)), f_{m=i-k}^{(i-j)}(g(h_{r=m+1}^{(i)}(a_r), b_m))) \quad \text{[definition of } Q(i, i-j+1)]$$
$$= f_{m=i-k}^{(i)}(g(h_{r=m+1}^{(i)}(a_r), b_m)) \quad \text{[associativity of } f]$$
$$= Q(i, i-k).$$

End of proof.

Theorem 2: For $1 \leqslant i \leqslant N$, $x_i = Q(i, 1)$.

Proof: By induction on i.

Basis Step: $i = 1$.

$$Q(1, 1) = b_1 = x_1 \quad \text{[by definition]}.$$

Induction Step: Assume $Q(j, 1) = x_j$ for $j < i$. Then

$$x_i = f(b_i, g(a_i, x_{i-1})) \quad \text{[recurrence equation (6)]}$$
$$= f(Q(i, i), g(h_{r=i}^{(i)}(a_r), Q(i-1, 1))) \quad \text{[inductive hypothesis, definition of } Q \text{ and } h \text{ composition]}$$
$$= Q(i, 1) \quad \text{[by Theorem 1]}.$$

End of proof.

We can now state a theorem that demonstrates the validity of Algorithm A.

Theorem 3: For all $1 \leqslant i \leqslant N$, $0 \leqslant k \leqslant \lceil \log_2 N \rceil$,

$$\text{a) } A^{(k)}(i) = \begin{cases} h_{r=2}^{(i)}(a_r), & 1 < i \leqslant 2^k + 1 \\ h_{r=i-2^k+1}^{(i)}(a_r), & 2^k + 1 < i \leqslant N \end{cases}$$

$$\text{b) } B^{(k)}(i) = \begin{cases} Q(i, 1), & 1 \leqslant i \leqslant 2^k \\ Q(i, i-2^k+1), & 2^k < i \leqslant N. \end{cases}$$

Proof: Directly by induction and Theorems 1 and 2. The proof of Theorem 3 is direct but tedious; we omit it here.

Using part b) of Theorem 3 we get the immediate result.

Corollary: After the $\lceil \log_2 N \rceil$th iteration of Algorithm A, $B(i)$ contains x_i for $1 \leqslant i \leqslant N$.

Thus we have shown that not only does Algorithm A compute the solution $x_1, \cdots, x_N$ to (6), but also that it terminates in exactly $\lceil \log_2 N \rceil$ iterations.

Acknowledgment

Recursive doubling solutions to the first-order problem of Section II-A were discovered independently by H. R. Downs of Systems Control, Inc., and H. Lomax of NASA Ames Research Center. Recursive doubling solutions to second-order linear recurrences have been known to J. J. Sylvester as early as 1853. The authors wish to thank D. Knuth for pointing out Sylvester's work and for several stimulating suggestions while this research was in progress and the referees for pointing out that the h function need not be associative.

After this paper had been reviewed and accepted for publication, the authors encountered the report by Trout [5], which has several similar results. His work was done independently of the work reported here and carries the research beyond the limits of this paper.

References

[1] O. Buneman, "A compact non-iterative Poisson solver," Stanford Univ. Inst. Plasma Res., Stanford, Calif., Rep. 294, 1969.
[2] B. L. Buzbee, G. H. Golub, and C. W. Nelson, "On direct methods for solving Poisson's equations," *SIAM J. Numer. Anal.*, vol. 7, pp. 627-656, Dec. 1970.
[3] I. Munro and M. Paterson, "Optimal algorithms for parallel polynomial evaluation," in *Conf. Rec., 1971 12th Annu. Symp. Switching and Automata Theory*, IEEE Publ. 71 C 45-C, pp. 132-139.
[4] H. S. Stone, "An efficient parallel algorithm for the solution of a tridiagonal linear system of equations," *J. Ass. Comput. Mach.*, vol. 20, pp. 27-38, Jan. 1973.
[5] H. R. G. Trout, "Parallel techniques," Dep. Comput. Sci., Univ. Illinois, Urbana, Rep. UIUCDCS-R-72-549, Oct. 1972.

Peter M. Kogge (S'65-M'68) was born in Washington, D.C., on December 3, 1946. He received the B.S.E.E. degree from the University of Notre Dame, Notre Dame, Ind., in 1968, the M.S. degree in systems and information sciences from Syracuse University, Syracuse, N.Y., in 1970, and the Ph.D. degree in electrical engineering from Stanford University, Stanford, Calif., in 1973.

Since 1968 he has been with the Federal Systems Division, IBM Corporation, and is pres-

ently in the Systems Architecture Department, Owego, N.Y., where he is involved with the definition of advanced computer architecture and organizations. From 1970 to 1972 he was at Stanford University under an IBM Resident Fellowship. Prior to that time he was involved with the design and organization of a large multiprocessor system for aerospace applications. His present interests include the definition and design of advanced computer systems, with particular emphasis on the introduction of parallelism into computer design and problem solutions.

Harold S. Stone (S'61–M'63), for a photograph and biography, see this issue, p. 710.

A Fast Poisson Solver Amenable to Parallel Computation

BILLY L. BUZBEE

Abstract—**The matrix decomposition Poisson solver is developed for the five-point difference approximation to Poisson's equation on a rectangle. This algorithm's suitability for parallel computation, its simplicity, its performance relative to successive overrelaxation, and its generality are then discussed.**

Index Terms—**Linear algebra, numerical solution of PDE's, Poisson equation.**

FAST Poisson solvers have evolved during the last ten years [1]-[3], and they consist of noniterative techniques for solving finite difference approximations to Poisson's equation on a rectangle. These techniques are significant because of their efficiency. For example, the Buneman-Poisson solver will usually solve the discrete equation in $\frac{1}{20}$th to $\frac{1}{40}$th of the time required by successive overrelaxation (SOR). In this paper we will show that one of these techniques, the matrix decomposition Poisson solver (MD), offers tremendous opportunity for parallel computation. Although the MD algorithm is quite general, we will only develop it for the five-point difference approximation to Poisson's equation on a rectangle with a uniform mesh. This approach will exhibit the programming details of the algorithm and emphasize its simplicity. However, a summary of the various generalizations of MD and a machine comparison of MD with SOR are included.

Consider a finite difference approximation to

$$-\nabla^2 u = f \quad (1)$$

in a rectangle R with $u = g(x,y)$ on the boundary ∂R. We will assume N discretizations in the horizontal direction and M discretizations in the vertical direction. See Fig. 1.

Manuscript received October 7, 1972; revised March 21, 1973. This work was done under the auspices of the U.S. Atomic Energy Commission.

The author is with the Los Alamos Scientific Laboratory, University of California, Los Alamos, N. Mex. 87544.

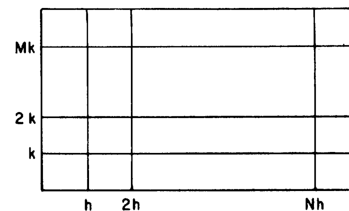


Fig. 1. $N \times M$ uniform rectangular mesh.

Let $u_{ij} = u(ih, jk)$ and approximate (1) by the five-point difference equation, that is,

$$(-\nabla^2 u)_{ij} \cong \frac{1}{h^2}\,[2u_{ij} - (u_{i+1,j} + u_{i-1,j})] + \frac{1}{k^2}\,[2u_{ij} - (u_{i,j+1} + u_{i,j-1})]. \quad (2)$$

The difference equations for the points along the line $x = h$ may be written

$$\begin{bmatrix} \delta u_{11} - \rho^2 u_{12} \\ -\rho^2 u_{11} + \delta u_{12} \quad -\rho^2 u_{13} \\ \cdots \\ -\rho^2 u_{1,M-1} + \delta u_{1,M} \end{bmatrix} + \begin{bmatrix} -u_{21} \\ -u_{22} \\ \vdots \\ -u_{2M} \end{bmatrix} = \begin{bmatrix} h^2 f_{11} + \rho^2 g_{10} + g_{01} \\ h^2 f_{12} + g_{02} \\ \vdots \\ h^2 f_{1M} + g_{0M} + \rho^2 g_{1,M+1} \end{bmatrix} \quad (3)$$

where $\rho = h/k$ and $\delta = 2(1 + \rho^2)$. If we let u_i be the M-dimensional vector