P. M. Kogge

Parallel Solution of Recurrence Problems

Abstract: An mth-order recurrence problem is defined as the computation of the sequence $x_1, \cdots, x_N$, where $x_i = f(\mathbf{a}_i, x_{i-1}, \cdots, x_{i-m})$ and $\mathbf{a}_i$ is some vector of parameters. This paper investigates general algorithms for solving such problems on highly parallel computers. We show that if the recurrence function f has associated with it two other functions that satisfy certain composition properties, then we can construct elegant and efficient parallel algorithms that can compute all N elements of the series in time proportional to $\lceil \log_2 N \rceil$. The class of problems having this property includes linear recurrences of all orders—both homogeneous and inhomogeneous, recurrences involving matrix or binary quantities, and various nonlinear problems involving operations such as computation with matrix inverses, exponentiation, and modulo division.

Introduction

A common problem in applied mathematics is the computation of a sequence of N elements denoted $x_1, \cdots, x_N$, given only a set of initial conditions $(x_0, \cdots, x_{-m+1})$ and a set of equations relating each x_i to m other elements of the sequence. Such a problem is called an mth-order recurrence problem. A common example is the description of a discrete-time linear system where the state of the system at time i is a linear function of the state at time $i-1$, namely:

$$\begin{aligned} &x_0 \text{ is given} \\ &x_1 = A_1 x_0 + B_1 \\ &\vdots \\ &x_i = A_i x_{i-1} + B_i \\ &\vdots \\ &x_N = A_N x_{N-1} + B_N. \end{aligned} \qquad (1)$$

Such problems appear on the surface to be highly sequential; we first use the initial conditions to compute one new x_i, then using the new x_i we compute x_{i+1}, and so on until the desired sequence is computed. This process is obviously well suited to standard single-instruction-stream, single-data-stream (SISD) computers. It is not, however, an efficient process for use on the new single-instruction-stream, multiple-data-stream (SIMD) computers that are capable of performing many simultaneous operations. The purpose of this paper is to describe certain functional properties that, when possessed by a given recurrence problem, allow the construction of new parallel algorithms that take advantage of the properties of a SIMD machine. These new algorithms run in time proportional to $\lceil \log_2 N \rceil$ as compared with the time proportional to N required by standard solutions. (The notation $\lceil x \rceil$ means the largest integer less than or equal to x.)

Most previous work in this field has centered either on very global aspects of parallelism, such as dependency ordering [1], or on highly parallel solutions to very specific problems. Typical specific solutions include polynomial evaluation [2], Poisson equation solution [3, 4], mathematical programming problems [5], tridiagonal equations [6], and minimax searches [7]. Although not formally presented as such, other methods such as the carry bypass adder [8] in reality represent direct implementations of parallel algorithms for specific recurrences.

A few general techniques for parallel solution of recurrence problems were suggested by Stone [6] and later developed into a general algorithm by Kogge and Stone [9]. Those results, however, are largely special cases of the results given in this paper.

Related topics include investigations of the numerical stability of parallel algorithms [10] and the minimal parallelism needed to solve recurrences [11,12].

Some of the concepts described in this paper were discovered concurrently and independently by Trout [13].

Definitions and notation

In this section we define the type of parallel computer assumed available, the general type of recurrence problem we consider for solution, and the notation used to describe the algorithms we develop.

- *Parallel computer*

The kind of parallel computer assumed available is a SIMD computer similar to that described in [14]. The major characteristics of such a computer's organization are as follows:

1. There are p identical processors, each able to execute the usual arithmetic and logical operations, and each with its own memory.
2. The operations performed by each processor involve at most two operands.
3. Each processor has a distinct index by which it may be referenced by an instruction.
4. All processors obtain their instructions simultaneously from a single instruction stream. Thus all processors execute the same instruction, but operate on data stored in their own memories.
5. Any processor may be "blocked" or "masked" from performing an instruction. This mask may be set by an explicit instruction directed to a processor by its index or by the result of some global test instruction, such as "set mask if accumulator is zero."
6. Under program control all unmasked processors can exchange data with each other over predefined data paths.

Figure 1 shows a diagram of a computer with the above characteristics.

- *Recurrence problem*

For this paper we define a general recurrence problem as follows:

Definition 1 The solution to an mth-order recurrence problem is a sequence $x_1, \cdots, x_N$, where initially we are given

1. a set of m initial values $\{x_0, \cdots, x_{-m+1}\}$.
2. a recurrence function (r function) f such that for all $i, 1 \le i \le N$,

   $$x_i = f(\mathbf{a}_i, x_{i-1}, \cdots, x_{i-m}), \tag{2}$$

   where $\mathbf{a}_i$ is termed a parameter vector and is a set of parameters independent of any x_j and is referenced by f during the computation of x_i.

This definition allows efficient use to be made of a SIMD computer's capabilities. Each processor can use the same instructions needed to evaluate f but on different parameters and data.

The simple problem (1) fulfills the above definition. Here

$$\begin{aligned} x_i &= A_i x_{i-1} + B_i \\ &= \mathbf{a}_i(1)x_{i-1} + \mathbf{a}_i(2) \\ &= f(\mathbf{a}_i, x_{i-1}), \end{aligned} \tag{3}$$

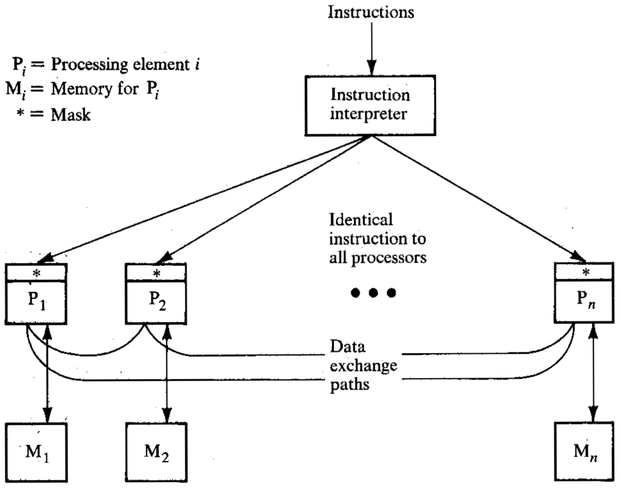


Figure 1 Computer model.

where the parameter vector $\mathbf{a}_i$ is the pair (A_i, B_i), and the function f is an add and a multiply.

We note that the lengths of the parameter vectors may vary from problem to problem. In the above example the length of each $\mathbf{a}_i$ is 2; for the problem

$$x_i = f(\mathbf{a}_i, x_{i-1}) = \frac{\mathbf{a}_i(1) + \mathbf{a}_i(2)x_{i-1}}{\mathbf{a}_i(3) + \mathbf{a}_i(4)x_{i-1}}, \tag{4}$$

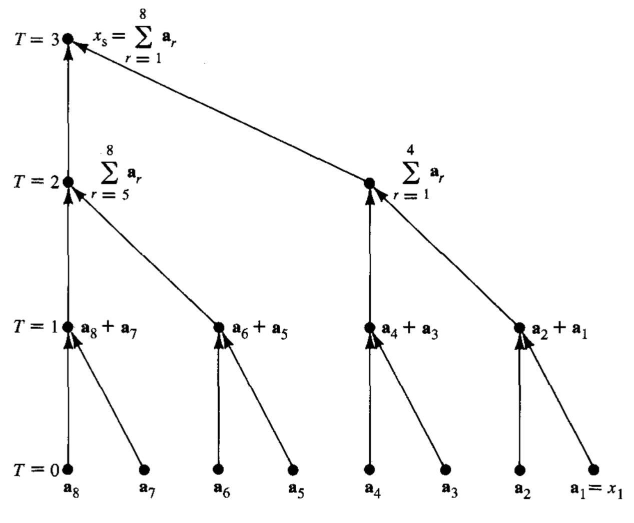
the length is 4. However, for any particular problem the length of each $\mathbf{a}_i$ must be constant for all i.

- *Algorithm notation*

The parallel algorithms developed in this paper are all described in an ALGOL-like notation. The major variation from standard ALGOL is found in descriptions of those aspects of a program that would be directly affected by execution on a SIMD computer. The primary differences are:

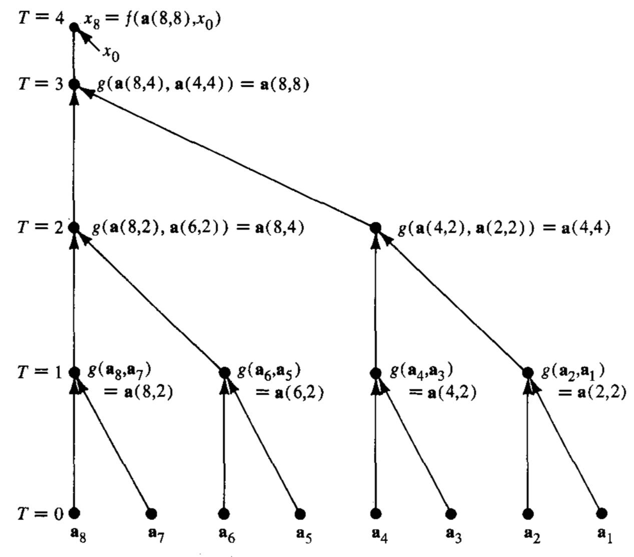
1. Boldface variables denote lists of elements, with the length of the list determined by the problem being solved. Thus $\mathbf{A} = g(\mathbf{B}, \mathbf{C})$ denotes a function g applied to two arguments $\mathbf{B}$ and $\mathbf{C}$ and returning as an output the list $\mathbf{A}$.
2. Arrays defined as parallel have one dimension (the one defined by the $*$) stored across the processing element memories. Thus if we have *parallel array* $\mathbf{A}(*, 1::5, 1::6)$, the ith processor has a 5×6 array $\mathbf{A}(i, 1, 1)$ through $\mathbf{A}(i, 5, 6)$ in its memory.
3. An inequality of the form $(M < i < N)$ following an assignment statement indicates that the statement is to be executed simultaneously for each value of index in the specified range.

This particular notation is not meant to reflect what exists, or should exist, in any real programming language



$T=3$
$x_8=\sum_{r=1}^{8}\mathbf{a}_r$
$T=2$
$\sum_{r=5}^{8}\mathbf{a}_r$
$\sum_{r=1}^{4}\mathbf{a}_r$
$T=1$
$\mathbf{a}_8+\mathbf{a}_7$
$\mathbf{a}_6+\mathbf{a}_5$
$\mathbf{a}_4+\mathbf{a}_3$
$\mathbf{a}_2+\mathbf{a}_1$
$T=0$
$\mathbf{a}_8$ $\mathbf{a}_7$ $\mathbf{a}_6$ $\mathbf{a}_5$ $\mathbf{a}_4$ $\mathbf{a}_3$ $\mathbf{a}_2$ $\mathbf{a}_1=x_1$

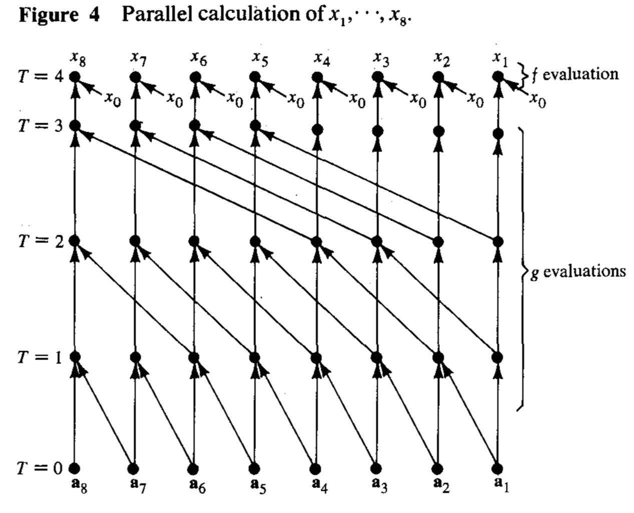
Figure 2 The log-sum algorithm.



$T=4$ $x_8=f(\mathbf{a}(8,8),x_0)$
x_0
$T=3$ $g(\mathbf{a}(8,4),\mathbf{a}(4,4))=\mathbf{a}(8,8)$
$T=2$ $g(\mathbf{a}(8,2),\mathbf{a}(6,2))=\mathbf{a}(8,4)$
$g(\mathbf{a}(4,2),\mathbf{a}(2,2))=\mathbf{a}(4,4)$
$T=1$ $g(\mathbf{a}_8,\mathbf{a}_7)=\mathbf{a}(8,2)$
$g(\mathbf{a}_6,\mathbf{a}_5)=\mathbf{a}(6,2)$
$g(\mathbf{a}_4,\mathbf{a}_3)=\mathbf{a}(4,2)$
$g(\mathbf{a}_2,\mathbf{a}_1)=\mathbf{a}(2,2)$
$T=0$
$\mathbf{a}_8$ $\mathbf{a}_7$ $\mathbf{a}_6$ $\mathbf{a}_5$ $\mathbf{a}_4$ $\mathbf{a}_3$ $\mathbf{a}_2$ $\mathbf{a}_1$

Figure 3 Parallel calculation of x_8.

Figure 4 Parallel calculation of $x_1,\cdots,x_8$.



x_8 x_7 x_6 x_5 x_4 x_3 x_2 x_1
f evaluation
$T=4$
x_0
$T=3$
$T=2$
g evaluations
$T=1$
$T=0$
$\mathbf{a}_8$ $\mathbf{a}_7$ $\mathbf{a}_6$ $\mathbf{a}_5$ $\mathbf{a}_4$ $\mathbf{a}_3$ $\mathbf{a}_2$ $\mathbf{a}_1$

designed for SIMD computers. It is simply a convenient notation for expressing the general ideas behind the various algorithms.

General first-order algorithm

The simplest class of recurrence problems is the first-order case in which x_i depends only on x_{i-1}. Many of the recurrence problems with known parallel solutions, such as Horner's rule for evaluating polynomials or the solution of (1), fall into this category. The classic parallel solution to this type of problem is the "log-sum" algorithm [15] for solving $x_i=\mathbf{a}_i+x_{i-1}$. The introduction of parallelism into the solution of this recurrence stems from the associativity of addition which allows us to rewrite the standard serial evaluation of, for example,

$$x_4=\mathbf{a}_4+(\mathbf{a}_3+(\mathbf{a}_2+\mathbf{a}_1)), \tag{5}$$

as

$$x_4=(\mathbf{a}_4+\mathbf{a}_3)+(\mathbf{a}_2+\mathbf{a}_1). \tag{6}$$

This second formulation allows two processors to work during the first step, one computing $(\mathbf{a}_4+\mathbf{a}_3)$ and the other computing $(\mathbf{a}_2+\mathbf{a}_1)$. For arbitrary N, the generalization of (6) allows $N/2$ parallel additions at the first step, $N/4$ at the second, $\cdots$, and $N/2^k$ at the kth until, at the $\lceil\log_2 N\rceil$th step, x_N is computed with one final addition. Figure 2 gives a diagram of this process for the computation of x_8.

Most recurrences, such as Eq. (1), are not associative and consequently cannot be solved in such a direct fashion. However, many of the most common r functions do have related to them another function with an associative-like property. In turn, this auxiliary function permits a "log-sum"-like solution to a large part of the solution of the original problem. This auxiliary function has as arguments two parameter vectors and produces a new parameter vector as output. It is defined as follows:

Definition 2 An r function f is said to have a companion function g (c function) if, for all x of the problem's domain and all parameter vectors $\mathbf{a}$, $\mathbf{b}$,

$$f(\mathbf{a},f(\mathbf{b},x))=f(g(\mathbf{a},\mathbf{b}),x). \tag{7}$$

For example, the c function for (1) is

$$g(\mathbf{a},\mathbf{b})=(\mathbf{a}(1)\mathbf{b}(1),\mathbf{a}(1)\mathbf{b}(2)+\mathbf{a}(2)). \tag{8}$$

All c functions have the following two easily proved properties:

Theorem 1 All c functions are associative with respect to their r functions; i.e., for all $\mathbf{a}$, $\mathbf{b}$, $\mathbf{c}$, x,

$$f(g(\mathbf{a},g(\mathbf{b},\mathbf{c})),x)=f(g(g(\mathbf{a},\mathbf{b}),\mathbf{c}),x). \tag{9}$$

Theorem 2 If f has a c function g, then any x_i can be expressed in terms of any x_j, $0\le j<i\le N$, as

Table 1 Applications of the FORA algorithm.

Problem class	*Companion function* $g(\mathbf{a}, \mathbf{b})$	*Examples* ($\mathbf{D}_r$ = *domain of variable r*)
1. $x_i = f(\mathbf{a}_i, x_{i-1})$ i. f is associative	$f(\mathbf{a}, \mathbf{b})$	1. $x_i = A_i x_{i-1}$ 2. $x_i = A_i + x_{i-1}$ $\Big\}$, $\mathbf{D}_A = \mathbf{D}_x = \{m \times m \text{ matrices}\}, m \geq 1$ 3. $x_i = \min(A_i, x_{i-1})$ 4. $x_i = \max(A_i, x_{i-1})$ $\Big\}$, $\mathbf{D}_A = \mathbf{D}_x = \text{real}$ 5. $x_i = A_i$ AND x_{i-1} 6. $x_i = A_i$ OR x_{i-1} 7. $x_i = A_i$ XOR x_{i-1} $\Big\}$, $\mathbf{D}_x = \mathbf{D}_A = \{0, 1\}$ 8. $x_i = A_i x_{i-1}$, $\mathbf{D}_x = \mathbf{D}_A = \{m \times m \text{ Boolean matrices}\}$
2. $x_i = f(\mathbf{a}_i(2), g(\mathbf{a}_i(1), x_{i-1}))$ i. f is associative ii. g has a companion function h iii. g right-distributes over f	$(h[\mathbf{a}(1), \mathbf{b}(1)],$ $f[\mathbf{a}(2), g(\mathbf{a}(1), \mathbf{b}(2))]$	1. $x_i = A_i x_{i-1} + B_i$ 2. $x_i = B_i x_{i-1}^{A_i}$ $\Big\}$, $\mathbf{D}_A = \mathbf{D}_B = \mathbf{D}_x = \text{real}$ 3. $x_i = (A_i$ AND $x_{i-1})$ OR B_i, $\mathbf{D}_A = \mathbf{D}_B = \mathbf{D}_x = \{0, 1\}$ 4. $x_i = A_i x_{i-1} + B_i$, $\mathbf{D}_A = \{m \times m \text{ matrices}\}$ $\mathbf{D}_B = \mathbf{D}_x = \{m \times 1 \text{ vectors}\}$
3. $x_i = [\mathbf{a}_i(1) + \mathbf{a}_i(2)x_{i-1}]$ $[\mathbf{a}_i(3) + \mathbf{a}_i(4)x_{i-1}]^{-1}$ i. $+$ is any associative and commutative function ii. $\cdot$ is any associative function that distributes over $+$ iii. for all x, y $(x \cdot y)^{-1} = y^{-1}x^{-1}$ iv. $x(yy^{-1}) = x$.	$(\mathbf{a}(1)\mathbf{b}(3) + \mathbf{a}(2)\mathbf{b}(1),$ $\mathbf{a}(1)\mathbf{b}(4) + \mathbf{a}(2)\mathbf{b}(2),$ $\mathbf{a}(3)\mathbf{b}(3) + \mathbf{a}(4)\mathbf{b}(1),$ $\mathbf{a}(3)\mathbf{b}(4) + \mathbf{a}(4)\mathbf{b}(2))$	1. $x_i = (A_i + B_i x_{i-1})(C_i + D_i x_{i-1})^{-1}$, $\mathbf{D}_x = \mathbf{D}_A = \mathbf{D}_B = \mathbf{D}_C = \mathbf{D}_D = \{m \times m \text{ matrices}\},$ $m \geq 1$ 2. $x_i = A_i / (C_i + D_i x_{i-1})$ partial fraction expansion 3. $x_i = B_i + A_i / x_{i-1}$ continued fraction expansion 4. $x_i = (A_i$ OR $(B_i$ AND $x_{i-1}))$ AND NOT $(C_i$ OR $(D_i$ AND $x_{i-1}))$, $\mathbf{D}_x = \mathbf{D}_{A,B,C,D} = \{0, 1\}$
	Special recurrence problems	
4. i. $x_i = (\mathbf{a}_i(1)x_{i-1} + \mathbf{a}_i(2))\|m$	$(\mathbf{a}(1)\mathbf{b}(1),$ $\mathbf{a}(1)\mathbf{b}(2) + \mathbf{a}(2))$	Random number generator (see Knuth [20], p. 10)
ii. $x_i = (\mathbf{a}_i(1)x_{i-1}{}^2 + \mathbf{a}_i(2))^{\frac{1}{2}}$	$(\mathbf{a}(1)\mathbf{b}(1),$ $\mathbf{a}(1)\mathbf{b}(2) + \mathbf{a}(2))$	

$$x_i = f(\mathbf{a}(i, i-j), x_j), \tag{10}$$

where

$$\mathbf{a}(i, r) = \begin{cases} \mathbf{a}_i \text{ for } r = 1, \\ g(\mathbf{a}_i, g(\mathbf{a}_{i-1}, \cdots, g(\mathbf{a}_{i-r+2}, \mathbf{a}_{i-r+1}) \cdots) \end{cases}$$
$$\text{for } r > 1. \tag{11}$$

The existence of a c function permits rapid construction of a parallel algorithm for the original problem. The second theorem allows, for example, any x_4,

$$x_4 = f(\mathbf{a}_4, f(\mathbf{a}_3, f(\mathbf{a}_2, f(\mathbf{a}_1, x_0)))), \tag{12}$$

to be rewritten as

$$x_4 = f(g(\mathbf{a}_4, g(\mathbf{a}_3, g(\mathbf{a}_2, \mathbf{a}_1))), x_0). \tag{13}$$

The associativity of g further allows this to be rewritten as

$$x_4 = f(g(g(\mathbf{a}_4, \mathbf{a}_3), g(\mathbf{a}_2, \mathbf{a}_1)), x_0). \tag{14}$$

The two terms $g(\mathbf{a}_4, \mathbf{a}_3)$ and $g(\mathbf{a}_2, \mathbf{a}_1)$ can be computed in parallel.

As with the log-sum algorithm, this procedure can be generalized to the solution of any x_N. Exactly $\lceil \log_2 N \rceil$ parallel g evaluations compute the parameter vector $\mathbf{a}(N, N)$. A single f evaluation combines this vector with x_0 to compute x_N. The computation of x_8 in this fashion is diagrammed in Fig. 3.

This procedure also extends to the simultaneous solution of the entire sequence $x_1, \cdots, x_N$, as depicted in Fig. 4. A set of N processors performs $\lceil \log_2 N \rceil$ g evaluations to compute the set of parameter vectors $\{\mathbf{a}(i, i) | 1 \leq i \leq N\}$, and a single f evaluation computes the desired sequence.

The following ALGOL-like program summarizes the algorithm of Fig. 4. It is called FORA (First Order Recurrence Algorithm).

```
procedure FORA;
begin
  comment allocate one parameter vector per processor,
    and initialize to $\mathbf{a}_i$;
  parallel array $\mathbf{A}(*)$;
  $\mathbf{A}(i) = \mathbf{a}_i, \ (1 \leq i \leq N)$;
  comment the following loop computes $\mathbf{a}(i, i)$;
  for $k = 1$ step 1 until $\lceil \log_2 N \rceil$ do
    $\mathbf{A}(i) = g(\mathbf{A}(i), \mathbf{A}(i - 2^{k-1})), \quad (2^{k-1} < i \leq N)$;
  comment now apply the initial conditions;
  $x_i = f(\mathbf{A}(i), x_0), \quad (1 \leq i \leq N)$;
end FORA.
```

The validity of this algorithm follows directly from Theorems 1 and 2.

• *Some examples*

Table 1 lists some general classes of recurrence problems that are suitable for solution by FORA. For each class, the general companion function and particular examples are given.

Class 1 in Table 1 covers associative functions such as $+$, $\times$, max, and min. It is clear that all such functions satisfy Definition 2 directly. In these cases the FORA solutions are identical to direct modifications of the log-sum algorithm.

The second class of recurrence problems listed in Table 1 has the form $f(\mathbf{a}, x) = f(\mathbf{a}(2), g(\mathbf{a}(1), x))$, where f and g have certain functional properties. Suitable problems include the introductory linear problem (1) and several highly nonlinear ones. This particular class of problems has been solved previously by Kogge and Stone [9] who used the concept of recursive doubling.

The third class of problems in Table 1 represents certain nonlinear problems with no previously known general parallel solution. Stone [6] was able to solve example 3 of this class with a parallel algorithm that is faster than FORA. His algorithm, however, is based on the recursive doubling solution of a second-order recurrence that can itself be solved still faster by the generalization of FORA described in a later section.

• *Minimization of execution time*

As evidenced by Stone's partial fraction algorithm, FORA is not necessarily the fastest method to solve all the problems it is capable of solving. For specific problems, or under certain conditions, modifications of FORA or other algorithms can run in considerably less time than a direct FORA implementation. This section describes several such situations.

There are two ways of measuring the execution time of an algorithm such as FORA: 1) counting only the number of (parallel) function evaluations and 2) actually computing running time in terms of the relative time required to compute each type of function. The first approach is often used in theoretical arguments about the complexity of an algorithm and is usually the approach that lends most insight into the question of lower bounds. The second approach is more pragmatic and is of most interest when an algorithm is selected for implementation on a real computer.

In terms only of the number of parallel evaluations, simple tree arguments based on combining the N parameter vectors $\mathbf{a}_1, \cdots, \mathbf{a}_N$ and x_0 with only two-argument functions indicate an absolute lower bound of $\alpha = \lceil \log_2 N + 1 \rceil$ parallel function evaluations. FORA requires about $\alpha + 1$ function evaluations (one f evaluation and $\lceil \log_2 N \rceil g$ evaluations) and so is quite close to the lower bound. However, if more detail is known about the nature of the f and g functions, a more accurate accounting of function evaluations can yield somewhat different results. For example, to solve the recurrence Eq. (1), FORA requires approximately $2\alpha + 1$ multiplications and $\alpha + 1$ additions. This is certainly an efficient implementation, but it is not the fastest possible. An obvious generalization, shown in Fig. 5, solves the same recurrence with only about $\alpha + 1$ additions and α multiplications. The penalty for this speed, however, seems to be in the number of required processing elements. FORA requires N processors, while the procedure in Fig. 5 requires about i processors for each x_i, for a total of $N^2/2$ processors for the same N-element sequence.

The same procedure diagrammed in Fig. 5 can be extended to cover any recurrence satisfying Class 2 of Table 1. Within this class one recurrence in particular that has received extensive study is the carry equation for binary addition of two binary numbers. Winograd [16] and Spira [17] have developed bounds on the minimum time to perform addition and Brent [18] has developed an approach similar to that shown in Fig. 5 for solving the carry recurrence and, thus, for performing addition in time close to Winograd's bounds with only $N\lceil \log_2 N \rceil$ processors (two-input logic elements). This is better than the procedure of Fig. 5, both in terms of time and the number of processors. However, the applicability of Brent's approach to other problems solvable by FORA is an open question.

If estimates of the actual times required to do an f or g evaluation [denoted $T(f)$ and $T(g)$ respectively] are available, a modification of FORA can also decrease total execution time. The number of function evaluation steps is higher than that for FORA, but the actual execution time is less. As an example, if $T(g) = 3T(f)$, a FORA computation of x_8 requires $T(f) + 3T(g) = 10\ T(f)$ units of time. A completely sequential evaluation of the same x_8 requires $8T(f)$ units of time. However, a combined parallel-sequential evaluation as pictured in Fig. 6 requires five function evaluation steps, but only $4T(f) + T(g) = 7T(f)$ units of time. This is less than for either FORA or the straight sequential solution.

In the process shown in Fig. 6, the function g can be used in parallel for any number of steps to compute new sets of parameter vectors. After any of these steps, parallel g evaluations can be replaced by sequential evaluations of f applied to the g-computed parameter vectors and a previously computed x_j. Figure 6 represents the case in which only one parallel g evaluation step is used. FORA, on the other hand, uses as many as possible, $\lceil \log_2 N \rceil$, followed by a single f evaluation. The actual point at which the switch between g and f evaluations occurs is completely open to choice. Consequently, if $T(f)$ and $T(g)$ are known, this point can be chosen to minimize total execution time. The following theorem defines the optimal choice of this switch point.

Theorem 3 The computation of x_N using k^* parallel g evaluations followed by sequential f evaluations is minimized in total execution time when k^* is either $\lceil \log_2(N(\log_e 2)T(f)/T(g)) \rceil$ or $\lfloor \log_2(N(\log_e 2)T(f)/T(g)) \rfloor$, whichever minimizes $kT(g) + NT(f)/2^k$.

Proof Clearly, after k parallel g evaluations there are still $N/2^k$ parameter vectors of the form $\mathbf{a}(j2^k, 2^k)$. If after the kth step we use only f evaluations, we need $T(f)N/2^k$ more units of time. A sequential combination of these vectors with the initial conditions gives a total execution time of

$$kT(g) + T(f)N/2^k. \qquad (15)$$

Setting the derivative with respect to k of (15) equal to zero, we obtain

$$T(g) - T(f)N(\log_e 2)2^{-k} = 0 \qquad (16)$$

or

$$k^* = \log_2(N(\log_e 2)T(f)/T(g)). \qquad (17)$$

Forcing k^* to be an integer gives the result stated in the theorem.

This theorem defines three regions for the value of k^*:

1. When $k^* \leq 0$, the all-serial algorithm is time-minimal;
2. when $0 < k^* < \alpha$, a partial-serial, partial-parallel algorithm is minimal; and
3. when $k^* \geq \alpha$, the all parallel FORA is minimal.

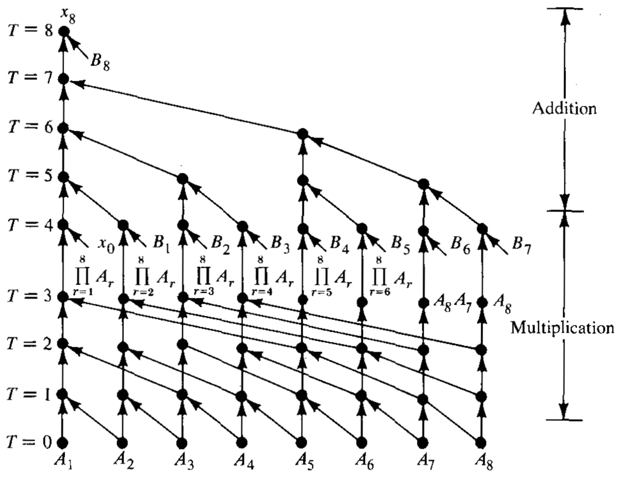


Figure 5 Fast computation of $x_i = A_i x_{i-1} + B_i$.

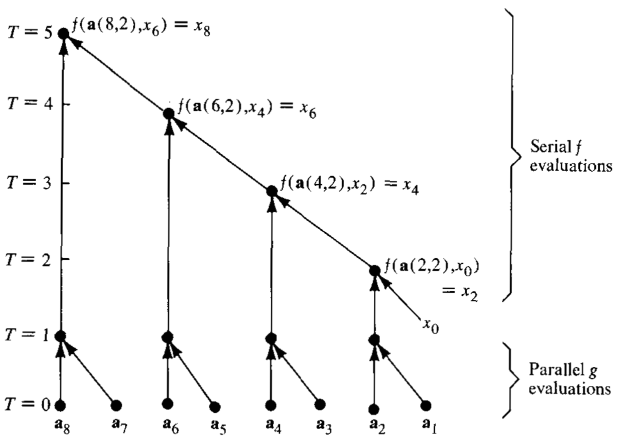


Figure 6 Minimal-time calculation of x_8 when $T(g) = 3T(f)$.

These three regions translate into three regions of the ratio $T(f)/T(g)$:

1. $T(f)/T(g) \leq 1/(N\log_e 2) \approx 1.45/N$: the serial algorithm is minimal:
2. $1.45/N \approx 1/N\log_e 2 < T(f)/T(g) < 1/\log_e 2 \approx 1.45$: a combined algorithm is minimal; and
3. $1.45 \approx 1/\log_e 2 \leq T(f)/T(g)$: an all-parallel algorithm is minimal.

The same procedure used in Fig. 6 to compute x_N alone can easily be extended to compute the entire series $x_1, \cdots, x_N$ with no loss in time. Figure 7 illustrates this extension for the case $N = 8$. The following program summarizes this procedure for any k^* between 0 and α. It is called FORAS (First Order Recurrence Algorithm Serial).

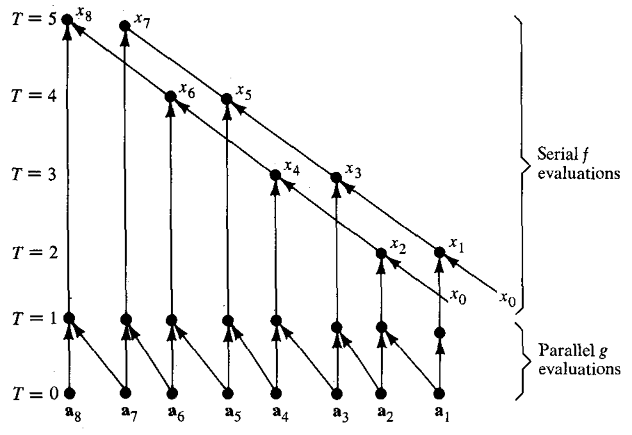

$T=5$
$T=4$
$T=3$
$T=2$
$T=1$
$T=0$
x_8 x_7 x_6 x_5 x_4 x_3 x_2 x_1 x_0 x_0
Serial f evaluations
Parallel g evaluations
$\mathbf{a}_8$ $\mathbf{a}_7$ $\mathbf{a}_6$ $\mathbf{a}_5$ $\mathbf{a}_4$ $\mathbf{a}_3$ $\mathbf{a}_2$ $\mathbf{a}_1$

Figure 7 Minimal-time calculation of $x_1, \cdots, x_8$ when $T(g) = 3T(f)$.

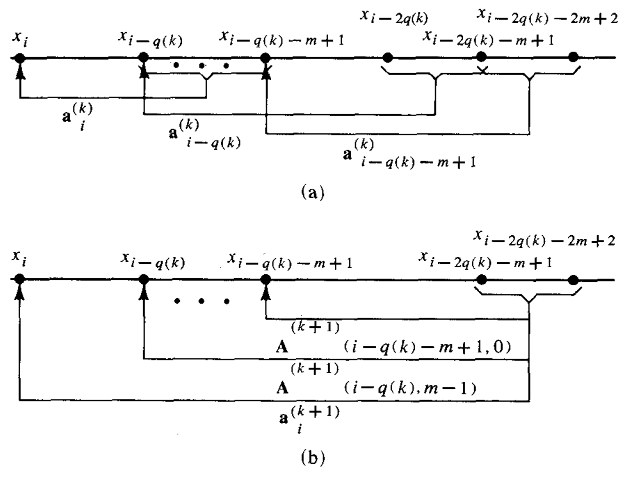


Figure 8 (a) Dependencies in computation of x_i after kth step. (b) Dependencies in computation of x_i at end of $(k+1)$th step.

```
procedure FORAS;
begin parallel array A(*);
  $\mathbf{A}(i) = \mathbf{a}_i, (1 \le i \le N)$;
  comment now compute $\mathbf{a}(i, 2^{k^*})$ for all $i$;
  for $k = 1$ step 1 until $k^*$ do
    $\mathbf{A}(i) = g(\mathbf{A}(i), \mathbf{A}(i - 2^{k-1})), (2^{k-1} < i \le N)$;
  comment now compute $x_1, \cdots, x_{2^{k^*}}$;
  $x_i = f(\mathbf{A}(i), x_0), (1 \le i \le 2^{k^*})$;
  comment now compute groups of $x_i$'s serially from previous groups. Each group has $2^{k^*}$ elements;
  for $j = 2^{k^*}$ step $2^{k^*}$ until $N - 2^{k^*}$ do
    $x_i = f(\mathbf{A}(i), x_{i-2^{k^*}}), (j < i \le j + 2^{k^*})$;
end FORAS;
```

The validity of FORAS follows immediately from the previous theorems.

General mth-order algorithm

The generality of the FORA algorithm stems from the existence of a companion for the r function. A similar situation exists for more general mth-order problems, although in this case two auxiliary functions, rather than one, are needed. These functions are defined as follows.

Definition 3 An mth-order r function has a companion set $\{g, h\}$ if there are functions g and h such that for all $\mathbf{a}_0, \mathbf{a}_1, \cdots, \mathbf{a}_m$ and $x_1, \cdots, x_m$,

$$f(\mathbf{a}_0, f(\mathbf{a}_1, x_1, \cdots, x_m), x_1, x_2, \cdots, x_{m-1}) = f(g(\mathbf{a}_0, \mathbf{a}_1), x_1, x_2, \cdots, x_m) \quad (18)$$

and

$$f(\mathbf{a}_0, f(\mathbf{a}_1, x_1, \cdots, x_m), \cdots, f(\mathbf{a}_m, x_1, \cdots, x_m)) = f(h(\mathbf{a}_0, \mathbf{a}_1, \mathbf{a}_2, \cdots, \mathbf{a}_m), x_1, \cdots, x_m). \quad (19)$$

The function g is similar to the definition of a companion function for a first-order problem; for example, it allows x_i to be expressed as a function of $x_{i-2}, \cdots, x_{i-m-1}$ as follows:

$$\begin{aligned} x_i &= f(\mathbf{a}_i, x_{i-1}, \cdots, x_{i-m}) \quad \text{(original recurrence)} \\ &= f(\mathbf{a}_i, f(\mathbf{a}_{i-1}, x_{i-2}, \cdots, x_{i-m-1}), x_{i-2}, \cdots, x_{i-m}) \quad \text{(substitution for } x_{i-1}) \\ &= f(g(\mathbf{a}_i, \mathbf{a}_{i-1}), x_{i-2}, \cdots, x_{i-m-1}). \quad (20) \end{aligned}$$

The function h permits us to rewrite a recurrence in which simultaneous substitutions are made for all unknowns as a single f evaluation involving a new parameter vector and the m common unknowns.

As with the first-order case, many of the problems encountered in practice have companion sets and are amenable to the solution technique described in this section. The general algorithm using these techniques is called MORA (mth-Order Recurrence Algorithm).

In operation, MORA proceeds much like FORA, with the computation at each step of a new set of parameter vectors $\{\mathbf{a}_i^{(k)} | 1 \le i \le N\}$ having the following properties:

$$\mathbf{a}_i^{(0)} = \mathbf{a}_i \text{ for } 1 \le i \le N, \quad (21)$$

$$x_i = f(\mathbf{a}_i^{(k)}, x_{i-q(k)}, \cdots, x_{i-q(k)-m+1}) \text{ for } q(k) < i \le N, \text{ and} \quad (22)$$

$$x_i = f(\mathbf{a}_i^{(k)}, x_0, x_{-1}, \cdots, x_{-m+1}) \text{ for } 1 \le i \le q(k), \quad (23)$$

where

$$q(k) = m2^k + 1 - m. \quad (24)$$

The general procedure for computing $\mathbf{a}_i^{(k+1)}$ from $\mathbf{a}_i^{(k)}$ is illustrated in Fig. 8. In this figure, the arrows represent the dependency of a sequence element x_j on m other x's, given the indicated parameter vector. To compute $\mathbf{a}_i^{(k+1)}$ from $\mathbf{a}_i^{(k)}$ we cannot simply substitute the expressions for $x_{i-q(k)}, \cdots, x_{i-q(k)-m+1}$ (involving $x_{i-2q(k)}, \cdots, x_{i-2q(k)-2m+2}$) into the expression for x_i. These expressions involve $2m-1$ unknowns while the function f can handle only m different x's. However, a sufficient number of applications of g in the fashion of equation (20) can produce new parameter vectors that express each $x_{i-q(k)-j}$, $0 < j < m-1$, in terms of the set $\{x_{i-2q(k)-m+1-j} | 0 \le j \le m-1\}$. For example,

$$x_{i-q(k)-m+2} = f(g(\mathbf{a}_{i-q(k)-m+2}^{(k)}, \mathbf{a}_{i-2q(k)-m+2}), x_{i-2q(k)-m+1}, \cdots, x_{i-2q(k)-2m+2}). \quad (25)$$

These new parameter vectors are denoted $\mathbf{A}^{(k+1)}(r, j)$, where

$$\mathbf{A}^{(k+1)}(r,j) = \begin{cases} \mathbf{a}_r^{(k)}, \ j=0 \\ g(\cdots g(g(\mathbf{a}_r^{(k)}, \mathbf{a}_{r-q(k)}), \mathbf{a}_{r-q(k)-1}), \cdots, \\ \quad \mathbf{a}_{r-q(k)-j+1}), \ j > 0. \end{cases} \quad (26)$$

Once a set $\mathbf{A}^{(k+1)}(i-q(k)-j, m-1-j)$, $0 \le j \le m-1$, of parameter vectors has been computed, the second function of the companion set, h, can be applied as in Eq. (19) to yield $\mathbf{a}_i^{(k+1)}$:

$$\mathbf{a}_i^{(k+1)} = h(\mathbf{a}_i^{(k)}, \mathbf{A}^{(k+1)}(i-q(k), m-1), \cdots, \mathbf{A}^{(k+1)}(i-q(k)-m+1, 0)). \quad (27)$$

This parameter vector expresses x_i as

$$x_i = f(\mathbf{a}_i^{(k+1)}, x_{i-2q(k)-m+1}, \cdots, x_{i-2q(k)-2m+2}). \quad (28)$$

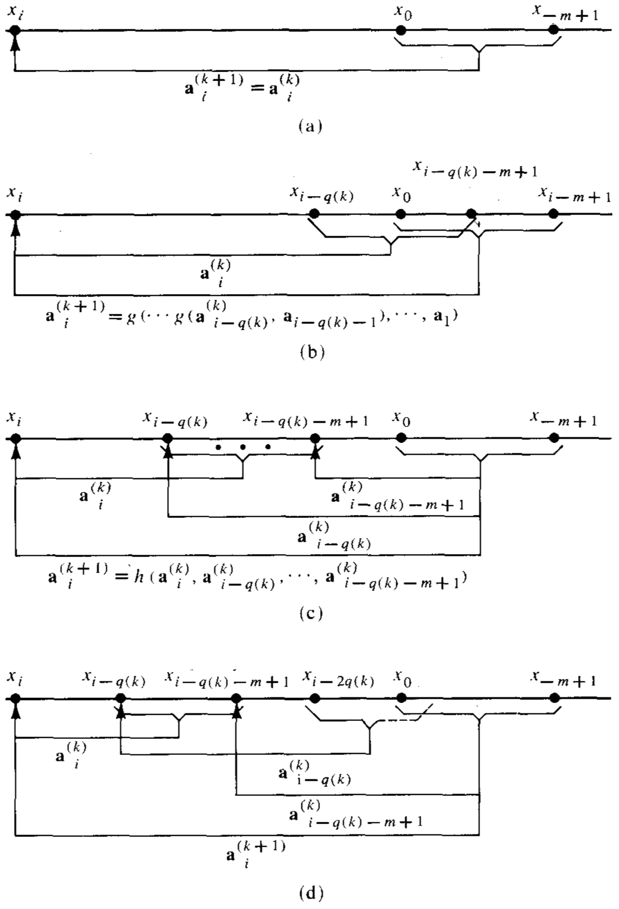
Thus $q(k+1) = 2q(k) + m - 1$ or, using the initial condition $q(0) = 1$, $q(k+1) = m2^{k+1} + 1 - m$.

The recurrence (27) is the heart of MORA. Each step of MORA uses it to compute in parallel a set $\{\mathbf{a}_i^{(k)} | 1 \le i \le N\}$. When $q(k) \ge N$, the procedure stops, and a single parallel f evaluation applied to these $\mathbf{a}_i^{(k)}$'s and the initial conditions $\{x_0, \cdots, x_{-m+1}\}$ yields the entire sequence $x_1, \cdots, x_N$.

There are, however, certain regions of i for each value of k in which the recurrence (27) must be slightly modified. The first four of the following five regions are illustrated in Figs 9(a)–9(d):

1. $1 \le i \le q(k)$
2. $q(k) + 1 \le i \le q(k) + m - 1$
3. $q(k) + m \le i \le 2q(k)$
4. $2q(k) + 1 \le i \le 2q(k) + m - 2 = q(k+1) - 1$
5. $q(k+1) \le i$.

We discuss each region separately, assuming that we are computing $\mathbf{a}_i^{(k+1)}$.



x_i
x_0
x_{-m+1}
$\mathbf{a}_i^{(k+1)} = \mathbf{a}_i^{(k)}$
(a)
$x_{i-q(k)-m+1}$
x_i
$x_{i-q(k)}$
x_0
x_{i-m+1}
$\mathbf{a}_i^{(k)}$
$\mathbf{a}_i^{(k+1)} = g(\cdots g(\mathbf{a}_{i-q(k)}^{(k)}, \mathbf{a}_{i-q(k)-1}), \cdots, \mathbf{a}_1)$
(b)
x_i
$x_{i-q(k)}$
$x_{i-q(k)-m+1}$
x_0
x_{-m+1}
$\mathbf{a}_i^{(k)}$
$\mathbf{a}_{i-q(k)-m+1}^{(k)}$
$\mathbf{a}_{i-q(k)}^{(k)}$
$\mathbf{a}_i^{(k+1)} = h(\mathbf{a}_i^{(k)}, \mathbf{a}_{i-q(k)}^{(k)}, \cdots, \mathbf{a}_{i-q(k)-m+1}^{(k)})$
(c)
x_i
$x_{i-q(k)}$
$x_{i-q(k)-m+1}$
$x_{i-2q(k)}$
x_0
x_{-m+1}
$\mathbf{a}_i^{(k)}$
$\mathbf{a}_{i-q(k)}^{(k)}$
$\mathbf{a}_{i-q(k)-m+1}^{(k)}$
$\mathbf{a}_i^{(k+1)}$
(d)

Figure 9 Regions of basic recurrence. (a) $1 \le i \le q(k)$, Region 1. (b) $q(k) + 1 \le i \le q(k) + m - 1$, Region 2. (c) $q(k) + m \le i \le 2q(k)$, Region 3. (d) $2q(k) + 1 \le i \le 2q(k) + m - 2 = q(k+1) - 1$, Region 4.

Region 1 $\ 1 \le i \le q(k)$

Here $\mathbf{a}_i^{(k)}$ expresses x_i directly in terms of $x_0, \cdots, x_{-m+1}$; thus $\mathbf{a}_i^{(k+1)} = \mathbf{a}_i^{(k)}$.

Region 2 $\ q(k) + 1 \le i \le q(k) + m - 1$

As pictured in Fig. 9(b), in this region there is at least one $x_{i-q(k)-j}$, $0 \le j \le m-1$, that is already an initial condition and as such does not allow any transformation involving the function h. However, $i - q(k)$ repeated applications of g, as in Eq. (20), directly express x_i in terms of $\{x_0, \cdots, x_{-m+1}\}$:

$$x_i = f(g(\cdots g(g(\mathbf{a}_i^{(k)}, \mathbf{a}_{i-q(k)}), \mathbf{a}_{i-q(k)-1}), \cdots, \mathbf{a}_1), x_0, \cdots, x_{-m+1}). \quad (29)$$

This composition of g is simply the parameter vector $\mathbf{A}^{(k+1)}(i, i-q(k))$ defined previously. Thus, to be consistent with Eq. (26),

$$\mathbf{a}_i^{(k+1)} = \mathbf{A}^{(k+1)}(i, i-q(k)). \quad (30)$$

Region 3 $q(k) + m \leq i \leq 2q(k)$
As pictured in Fig. 9(c), in this region all $\{\mathbf{a}_{i-q(k)-j}^{(k)} | 0 \leq j \leq m-1\}$ directly express the corresponding $x_{i-q(k)-j}$'s in terms of $\{x_0, \cdots, x_{-m+1}\}$. Consequently, $\mathbf{a}_i^{(k+1)}$ can be computed directly using the h function as follows:

$$\mathbf{a}_i^{(k+1)} = h(\mathbf{a}_i^{(k)}, \mathbf{a}_{i-q(k)}^{(k)}, \cdots, \mathbf{a}_{i-q(k)-m+1}^{(k)}). \tag{31}$$

Region 4 $2q(k) + 1 \leq i \leq 2q(k) + m - 2$
In this region some, but not all, of the $\mathbf{a}_{i-q(k)-j}^{(k)}$'s, $0 \leq j \leq m-1$, express $x_{i-q(k)-j}$ in terms of $\{x_0, x_{-1}, \cdots, x_{-m+1}\}$. Those $\mathbf{a}_{i-q(k)-j}^{(k)}$'s that do not so express their $x_{i-q(k)-j}$ must be transformed using g evaluations as in the general case into parameter vectors that do. The previously defined vector $\mathbf{A}^{(k+1)}(i - q(k) - j, i - 2q(k) - j)$ does this. Thus we need to compute $\mathbf{a}_i^{(k+1)}$ from

$$\begin{aligned}\mathbf{a}_i^{(k+1)} = h(&\mathbf{a}_i^{(k)}, \mathbf{A}^{(k+1)}(i-q(k), i-2q(k)),\\ &\mathbf{A}^{(k+1)}(i-q(k)-1, i-2q(k)-1), \cdots,\\ &\mathbf{A}^{(k+1)}(i-q(k)-j, 0), \mathbf{a}_{i-q(k)-j-1}^{(k+1)}, \cdots,\\ &\mathbf{a}_{i-q(k)-m+1}^{(k+1)}).\end{aligned} \tag{32}$$

Region 5 $q(k+1) \leq i \leq N$
In this region $\mathbf{a}_i^{(k+1)}$ may be computed directly from (27).

It would seem from the above discussion that MORA must have five separate subprogram segments—one for each region. If, however, the definition of $\mathbf{A}^{(k)}(i, j)$ is slightly modified to prevent referencing any $\mathbf{a}_r$, $r \leq 0$, only two regions need be considered.

This new definition of $\mathbf{A}^{(k+1)}(i, j)$ is

$$\mathbf{A}^{(k+1)}(i,j) = \begin{cases} \mathbf{a}_i^{(k)} \text{ for } j = 0, \\ \mathbf{A}^{(k+1)}(i, j-1) \text{ for } j > 0 \text{ and} \\ \quad q(k) + 1 \leq i \leq q(k) + j - 1, \\ g(\mathbf{A}^{(k+1)}(i, j-1), \mathbf{a}_{i-q(k)-j+1}) \text{ for } j > 0 \\ \quad \text{and } q(k) + j \leq i \leq N. \end{cases} \tag{33}$$

The properties of this $\mathbf{A}^{(k-1)}(i, j)$ are given in the following theorem, which can be proved by induction.

Theorem 4 For all i and j, $1 \leq i \leq N, 0 \leq j \leq m-1$,

$$\mathbf{A}^{(k-1)}(i, j) = \begin{cases} \mathbf{a}_i^{(k)}, \text{ for } 1 \leq i \leq q(k), \\ g(\cdots g(g(\mathbf{a}_i^{(k)}, \mathbf{a}_{i-q(k)}), \mathbf{a}_{i-q(k)-1}), \cdots, \mathbf{a}_1) \\ \quad \text{for } q(k) + 1 \leq i \leq q(k) + j - 1, \\ g(\cdots g(g(\mathbf{a}_i^{(k)}, \mathbf{a}_{i-q(k)}), \mathbf{a}_{i-q(k)-1}), \cdots, \\ \quad \mathbf{a}_{i-q(k)-j+1}) \text{ for } q(k) + j \leq i \leq N. \end{cases} \tag{34}$$

The following theorem defines the two regions that must be used to compute all $\mathbf{a}_i^{(k+1)}$. Its proof follows from Theorem 4 and the definitions of the five regions of i.

Theorem 5 For all $k > 0$,

$$\mathbf{a}_i^{(k+1)} = \begin{cases} \mathbf{A}^{(k+1)}(i, m-1) \text{ for } 1 \leq i \leq q(k) + m - 1, \\ h(\mathbf{a}_i^{(k+1)}, \mathbf{A}^{(k+1)}(i - q(k), m-1), \cdots, \\ \quad \mathbf{A}^{(k+1)}(i - q(k) - m + 1, 0)) \text{ for} \\ \quad q(k) + m \leq i \leq N. \end{cases} \tag{35}$$

From the definition of $\mathbf{A}^{(k)}(i, j)$ and Theorem 5, we can now construct MORA.

```
procedure MORA;
begin parallel array $\mathbf{A}(*, 0::m-1)$;
  $\mathbf{A}(i, 0) = \mathbf{a}_i$, $(1 \leq i \leq N)$;
  comment during the $(k+1)$th iteration of the following
     loop,
     1. $q = q(k) = m2^k - m + 1$,
     2. $\mathbf{A}(i, j)$ is the $\mathbf{A}^{(k+1)}(i, j)$ defined in Eq. (33);
  for $q = 1$ step $q + m - 1$ until $(N - m + 1)/2$ do
     begin comment the next loop computes $\mathbf{A}(i, j)$,
       $j > 0$; for $j = 1$ step 1 until $m - 1$ do
         begin $\mathbf{A}(i,j) = \mathbf{A}(i, j-1)$, $(1 \leq i \leq q + j - 1)$;
               $\mathbf{A}(i,j) = g(\mathbf{A}(i, j-1),$
                     $\mathbf{a}_{i-q-j+1})$, $(q + j \leq i \leq N)$;
         end;
       comment now compute $\mathbf{A}(i, 0) = \mathbf{a}_i^{(k+1)}$ using
         Eq. (27)
       $\mathbf{A}(i, 0) = h(\mathbf{A}(i, 0), \mathbf{A}(i - q, m - 1), \cdots,$
                 $\mathbf{A}(i - q - m + 1, 0))$, $(q + m \leq i \leq N)$;
       $\mathbf{A}(i, 0) = \mathbf{A}(1, m - 1)$, $(1 \leq i \leq q + m - 1)$;
     end;
  comment now apply initial conditions to compute all
     $x_i$'s;
  $x_i = f(\mathbf{A}(i, 0), x_0, \cdots, x_{-m+1})$, $(1 \leq i \leq N)$;
end MORA;
```

The validity of MORA follows directly from Theorem 5 and the above discussions.

The amount of time required by MORA is obtained directly. The outer loop is executed k times, where k is the smallest integer such that $N < m2^k + 1 - m$, or $k = \lceil log_2(N + m - 1)/m \rceil$. In each interation of this loop, there are $m - 1$ calls on g and one call on h. Thus the total time is $(m-1)kT(g) + kT(h) + T(f)$.

MORA solution to the second-order linear recurrence

One of the most common problems in applied mathematics is the solution of the three-term recurrence:

$$x_i = \mathbf{a}_i(1)x_{i-1} + \mathbf{a}_i(2)x_{i-2}. \tag{36}$$

The recurrence function in this case is

$$f(\mathbf{a}, x, y) = \mathbf{a}(1)x + \mathbf{a}(2)y. \tag{37}$$

A companion set exists for this function, and is easily computed by composition of the function with itself. The g function, for example, can be observed from

$$\begin{aligned} f(\mathbf{a}, f(\mathbf{b}, x, y), x) &= [\mathbf{a}(1)\mathbf{b}(1) + \mathbf{a}(2)]x + \mathbf{a}(1)\mathbf{b}(2)y \\ &= f([\mathbf{a}(1)\mathbf{b}(1) + \mathbf{a}(2), \\ &\qquad \mathbf{a}(1)\mathbf{b}(2)], x, y). \end{aligned} \tag{38}$$

Likewise, the h function can be derived from

$$\begin{aligned} &f(\mathbf{a}, f(\mathbf{b}, x, y), f(\mathbf{c}, x, y)) \\ &\quad= [\mathbf{a}(1)\mathbf{b}(1) + \mathbf{a}(2)\mathbf{c}(1)]x \\ &\qquad + [\mathbf{a}(1)\mathbf{b}(2) + \mathbf{a}(2)\mathbf{c}(2)]y \\ &\quad= f([\mathbf{a}(1)\mathbf{b}(1) + \mathbf{a}(2)\mathbf{c}(1), \mathbf{a}(1)\mathbf{b}(2) \\ &\qquad + \mathbf{a}(2)\mathbf{c}(2)], x, y). \end{aligned} \tag{39}$$

Substituting these functions into MORA yields SORA, a second-order recurrence algorithm. The following program lists this algorithm in an ALGOL-like format. The notation $\mathbf{A}(i, j)(k)$ refers to the kth element of the parameter vector $\mathbf{A}(i, j)$.

```
procedure SORA;
begin parallel array $\mathbf{A}(*; 0::1)$;
$\mathbf{A}(i, 0) = \mathbf{a}_i, (1 \le i \le N)$;
for $q = 1$ step $q + 1$ until $(N - 1)/2$ do
   begin $\mathbf{A}(i, 1) = \mathbf{A}(i, 0), (1 \le i \le q)$;
   $\mathbf{A}(i, 1) = (\mathbf{A}(i, 0)(1)\ \mathbf{a}_{i-q}(1) + \mathbf{A}(i, 0)(2),$
            $\mathbf{A}(i, 0)(1)\ \mathbf{a}_{i-q}(2)), (q < i \le N)$;
   $\mathbf{A}(i, 0) = (\mathbf{A}(i, 0)(1)\ \mathbf{A}(i - q, 1)\ (1)$
            $+ \mathbf{A}(i, 0)(2)\ \mathbf{A}(i - q - 1, 0)(1),$
            $\mathbf{A}(i, 0)(1)\ \mathbf{A}(i - q, 1)(2)$
            $+ \mathbf{A}(i, 0)(2)\ \mathbf{A}(i - q - 1, 0)(2)),$
            $(q + 1 < i \le N)$;
   $\mathbf{A}(i, 0) = \mathbf{A}(i, 1), (1 \le i \le q + 1)$;
   end;
$x_i = \mathbf{A}(i, 0)(1)\ x_0 + \mathbf{A}(i, 0)(2)\ x_{-1}, (1 \le i \le N)$;
end SORA;
```

With an approximation of $\lceil \log_2(N + 1)/2 \rceil$ as $\alpha - 1$, this algorithm requires $6\alpha - 4$ multiplications and $3\alpha - 2$ additions to compute an N-element sequence.

SORA is not the only known parallel algorithm for solving Eq. (36). There are at least two others (cf. Stone [6]). The first is simply to express (36) as

$$\begin{bmatrix} x_i \\ x_{i-1} \end{bmatrix} = \begin{bmatrix} \mathbf{a}_i & \mathbf{b}_i \\ 1 & 0 \end{bmatrix} \begin{bmatrix} x_{i-1} \\ x_{i-2} \end{bmatrix} = \mathbf{M}_i \begin{bmatrix} x_{i-1} \\ x_{i-2} \end{bmatrix}. \tag{40}$$

The parallel log-product algorithm on the matrices $\mathbf{M}_1, \cdots, \mathbf{M}_N$, followed by a matrix-vector product of the form $x_i = (\mathbf{M}_i\mathbf{M}_{i-1} \cdots \mathbf{M}_1)[x_0 x_{-1}]^{\mathrm{T}}$ computes the same sequence in $8\alpha - 6$ multiplications and $4\alpha - 3$ additions. This is roughly one-third slower than SORA.

Noting an observation by Euler [19], Stone [6] derived an entirely different algorithm for solving (36)—one based directly on the technique of recursive doubling and the analytic solution to (36). Stone's algorithm requires about $8\alpha - 4$ multiplications and $3\alpha - 1$ additions, faster than the matrix approach, but still slower than SORA.

As can be seen from the above comments, SORA is considerably faster than either of the other approaches. Whether or not SORA is the fastest parallel algorithm that solves (36) is an open, and interesting, question.

Mth-order linear nonhomogeneous recurrences

The most common recurrence encountered in practice is the general mth nonhomogeneous recurrence:

$$x_i = \sum_{j=1}^{m} \mathbf{a}_i(j)x_{i-j} + \mathbf{a}_i(m + 1), \tag{41}$$

where $\mathbf{a}_i(j)$ is the jth component of $\mathbf{a}_i$. This recurrence has the companion set

$$g(\mathbf{a}, \mathbf{b})(j) = \begin{cases} \mathbf{a}(1)\mathbf{b}(j) + \mathbf{a}(j + 1) \text{ for } 1 \le j \le m - 1, \\ \mathbf{a}(1)\mathbf{b}(m) \text{ for } j = m, \\ \mathbf{a}(m + 1) + \mathbf{a}(1)\mathbf{b}(m + 1) \text{ for } j = m + 1, \end{cases}$$

and

$$h(\mathbf{a}_0, \cdots, \mathbf{a}_{m+1})(j) = \begin{cases} \sum_{r=1}^{m} \mathbf{a}_0(r)\mathbf{a}_r(j) \text{ for } 1 \le j \le m \\ \sum_{r=1}^{m} \mathbf{a}_0(r)\mathbf{a}_r(m + 1) \\ \quad + \mathbf{a}_0(m + 1) \text{ for } j = m + 1. \end{cases}$$

The only other known algorithms for solving such problems are generalizations of Stone's second-order algorithm [for the case when all $\mathbf{a}_i(m + 1)$ equal zero], and the solution of matrix products of the form

$$\begin{bmatrix} x_i \\ x_{i-1} \\ \vdots \\ x_{i-m+1} \\ 1 \end{bmatrix} = \begin{bmatrix} \mathbf{a}_{-i}(1) & \cdots & & \mathbf{a}_{-i}(m + 1) \\ 1 & 0 \cdots 0 & & 0 \\ & & \vdots & \\ 0 & 0 \cdots 1 & & 0 \\ 0 & 0 \cdots 0 & & 1 \end{bmatrix} \begin{bmatrix} x_{i-1} \\ \vdots \\ x_{i-m+1} \\ x_{i-m} \\ 1 \end{bmatrix}. \tag{42}$$

As with the second-order case, both of these algorithms appear to be slower than the MORA equivalent.

Conclusions

As demonstrated by the algorithms developed in this paper, the existence of composition properties in recurrence functions permits the direct construction of elegant and efficient parallel programs for the solution of a wide class of recurrence problems. This is of both theoretical and practical importance. The demonstration of the usefulness of composition properties helps form a theoretical basis for the understanding of parallelism and its application to apparently serial processes. From a practical standpoint, the definition of companion functions and the standard formats of FORA and MORA allow a semi-automated approach to the construction of parallel algorithms. The test for the existence of a companion set to a new recurrence function is relatively direct; observation and rearrangement of simple compositions of the function with itself are usually sufficient. Further, it is not hard to imagine that future compilers for SIMD computers will assume many of these capabilities. Such a compiler could take, for example, a DO loop in a FORTRAN-like language, determine that the loop represents the solution to some recurrence, use a package of algebraic substitution routines to test for the existence of a companion set, and directly construct the appropriate parallel program from the MORA template—all without programmer intervention.

Several other crucial research topics are opened up by the results of this paper, including searches for other composition principles, analyses of numerical stability of these algorithms, minimal time for the solution of recurrences, and the minimal parallelism needed to solve them. Several starts into these areas have begun. For example, the "cyclic reduction" technique [3, 4] for the solution of Poisson's equation suggests a new composition principle that computes sequences of $\mathbf{a}_i^{(k)}$'s expressing x_i as

$$x_i = f(\mathbf{a}_i^{(k)}, x_{i-q(k)}, x_{i-2q(k)}, \cdots, x_{i-mq(k)}). \quad (43)$$

In terms of numerical stability, initial studies documented in the author's thesis [10] show that bounds on the absolute error in the algorithm SORA (and several other parallel algorithms) grow at the same rate as that derivable for a standard sequential solution. Other studies [11,12] have placed lower bounds on the minimal parallelism needed under certain circumstances to solve various subclasses of recurrence problems.

Acknowledgments

This work is part of a Ph.D. thesis developed at Stanford University under an IBM Resident Fellowship and National Science Foundation Grant GJ1180. I thank Professor H. S. Stone of Stanford University for his continual advice and encouragement during the development of the work described in this paper, and the International Business Machines Corp. and the National Science Foundation for their financial assistance. A summary version of this paper was presented at the Seventh Annual Princeton Conference on Information Sciences and Systems in March 1973.

References

1. R. Karp, R. Miller and S. Winograd, "The Organization of Computations for Uniform Recurrence Equations," *J. ACM* **14,** 563 (1967).
2. I. Munro and M. Paterson, "Optimal Algorithms for Parallel Polynomial Evaluation," *Conference Record of 12th Annual Symposium on Switching and Automata Theory*, IEEE Pub. 71 C 45-C, pp. 132-139, 1971.
3. O. Buneman, "A Compact Non-iterative Poisson Solver," Report 294, Stanford University Institute for Plasma Research, Stanford, California, 1969.
4. B. L. Buzbee, G. Golub and C. Neilson, "On Direct Methods for Solving Poisson's Equations," *SIAM J. Numer. Anal.* **7,** 627 (1970).
5. P. A. Gilmore, "Structuring of Parallel Algorithms," *J. ACM* **15,** 176 (1968).
6. H. S. Stone, "An Efficient Parallel Algorithm for the Solution of a Tridiagonal Linear System of Equations," *J. ACM* **20,** 27 (1973).
7. R. Karp and W. Miranker, "Parallel Minimax Search for a Maximum," *J. Combinatorial Theory* **4,** 19 (1968).
8. H. W. Gschwind, *Design of Digital Computers*, Springer-Verlag, New York, N.Y., 1967.
9. P. Kogge and H. S. Stone, "A Parallel Algorithm for the Efficient Solution of a General Class of Recurrence Equations," *IEEE Trans. Computers* **C-22,** 786 (1973).
10. P. M. Kogge, "The Numerical Stability of Parallel Algorithms for Solving Recurrence Problems," Report 44, Digital Systems Laboratory, Stanford University, California, September 1972.
11. P. M. Kogge, "Minimal Parallelism in the Solution of Recurrence Problems," Report 45, Digital Systems Laboratory, Stanford, California, September 1972.
12. Y. Muraoka, "Parallelism, Exposure and Exploitation in Programs," Report 424, Dept. of Computer Science, University of Illinois, Urbana, February 1971.
13. H. Trout, *Parallel Techniques*, Ph.D. Thesis, Dept. of Computer Science, Univ. of Illinois, Urbana, Illinois, October 1972.
14. G. H. Barnes et al., "The ILLIAC IV Computer," *IEEE Trans. Computers* **C-17,** 746 (1968).
15. D. Kuck, "ILLIAC IV Software and Applications Programming," *IEEE Trans. Computers* **C-17,** 758 (1968).
16. S. Winograd, "On the Time Required to Perform Addition," *J. ACM* **12,** 277 (1965).
17. P. Spira, "The Time Required for Group Multiplication," *J. ACM* **16,** 235 (1969).
18. R. Brent, "On the Addition of Binary Numbers," *IEEE Trans. Computers* **C-19,** 758 (1970).
19. L. Euler, *Introductio in Analysin Infinitorum*, Lausanne, Sections 359-361, 1748.
20. D. E. Knuth, "Seminumerical Algorithms," *The Art of Computer Programming*, Vol. 2, Addison-Wesley Publishing Co., Inc., Reading, Mass., 1969.

Received March 9, 1973; revised October 18, 1973

The author is at the IBM Federal Systems Division Electronics Systems Center in Owego, New York 13827.