Scheduling algorithms for concurrent execution

Y. Wallach
Faculty of Electrical Engineering, Technion—Israel Institute of Technology, Technion City, Haifa, Israel 32000

The paper introduces a notation for describing algorithms in a pseudo-mathematical form. It uses ideas introduced by various higher level programming languages, and by extensions proposed for concurrent processing. This notation is shown to be suitable to describe both sequential and concurrent scheduling schemes for numerical quadrature and matrix inversion. Additionally, pseudo-programs are developed for concurrent calculation of a scalar product of two vectors and of linear recurrence relations, as needed for a not too well known integration method.

(Received September 1973)

1. Introduction

Real time, online computer control requires speed above those normally available on digital computers.

One approach to accelerating the computations would be to undertake them on a parallel processing system (PPS hereafter) which includes a number of processors working concurrently on banks of common memory through multiplexers. The associated operating system keeps pointers on 'executive points' of programs and queues the processors. Whenever a processor is 'freed', it enters a queue; whenever a 'job' is waiting, the first processor in the queue is allocated to do it.

The imminent availability of such systems (Lorin, 1972; Comte Corporation, 1975) raises the questions:

(a) what language should be used to describe algorithms as scheduled on a PPS?
(b) which algorithms are best suited for execution on a PPS and how should they be scheduled?

The aim of this paper is to develop a notation suitable for programming a PPS and apply it to the scheduling of two problems: quadrature and matrix inversion.

2. Notation

Various languages are used to express the programming of an algorithm. When dealing with concurrent execution of an algorithm, these languages are supplemented by certain extensions (Anderson, 1965; Dijkstra, 1968; Hansen, 1973). The same approach was applied by the author to other problems (Wallach, 1974) for a particular system (Wallach, 1977). The present paper though will use a different notation because of the following considerations.

An actual program includes declarations, temporary variables and (to be efficient) probably a few 'tricks'. All this obscures the idea behind the algorithm. On the other hand, the number of basic, new ideas introduced by higher level (sequential or concurrent) languages and essential to express numerical algorithms is limited. We will therefore adhere to the old-fashioned way of expressing algorithms in a mathematical notation with its Greek letters, powerful operators and interspersed sentences in English to which we add the following:

(a) The assignment operator is := whereas := will be used as a (Boolean) relational operator. Vectors are assigned by positions, e.g. (a, b, c) := (x, y, z) is equivalent to a := x, b := y and c := z.

(b) A Boolean expression is enclosed in braces.

(c) A label is the customary identifier, followed by a colon.

(d) A 'goto' operator is denoted by the '⇒' symbol.

(e) Instructions to be executed sequentially are separated by semicolons, concurrently by commas and both may be completed by one of the delimiters to be yet described.

Semicolons may be read 'followed by', commas as 'concurrently with'. In this paper, $S$ will stand for a single instruction, a block of instructions, or even for a complete subroutine.

(f) Any block of instructions which is to be treated as a single one is enclosed in a pair of square brackets. In the case of a concurrent block e.g. $[S_1, S_2, \ldots, S_n]$ it was pointed out (Hansen, 1973) that the instructions $S_i, i = 1, \ldots, n$ should be disjoint or noninteractive. Inside of a block there should be no mixing of semicolons and commas, since this would require either parenthesisation or setting up of precedences for the two modes of execution.

(g) If a block is preceded by symbol '≠' and a list of values, it is to be performed for all of them. The four possibilities:

\[
\# i := 1; 2; \ldots; n[S_i; S_{i+1}]
\]
\[
\# i := 1, 2, \ldots, n[S_i; S_{i+1}]
\]
\[
\# i := 1; 2; \ldots; n[S_i; S_{i+1}]
\]
\[
\# i := 1, 2, \ldots, n[S_i; S_{i+1}]
\]

(correspond to the following execution sequences:

\[
[S_1; S_{n+1}; S_2; S_{n+2}; \ldots; S_n; S_{2n}]
\]
\[
[S_1; S_{n+1}; S_2; S_{n+2}; \ldots; S_n; S_{2n}]
\]
\[
[S_1; S_{n+1}; S_2; S_{n+2}; \ldots; S_n; S_{2n}]
\]
\[
[S_1; S_{n+1}; S_2; S_{n+2}; \ldots; S_n; S_{2n}]
\]

respectively. The 'step' of the index value $i$ is always implied by its initial two values; the loop is completed upon $i > n$.

(h) A conditional expression uses arrows to denote 'then' (McCarthy, 1960) and dots to denote 'else', with the 'else' matching the last 'if'. Hence \{(p_1) ⇒ l. (p_2) ⇒ a := a + b\; \text{means (in ALGOL 60)}; \text{if } p \text{ then goto } l \text{ else if } p_2 \text{ then } a := a + b\}. \text{ Note, that if } p \text{ then if } q \text{ then } S_1 \text{ else } S_2 \text{ corresponds to } \{p \rightarrow \{q\} \rightarrow S_1 \rightarrow S_2\}; \text{ but that if } p \text{ then if } (q \text{ then } S_1 \text{ else } S_2) \text{ must be written: } \{p \rightarrow \{q\} \rightarrow S_1 \rightarrow S_2\}.

(i) Functions will be defined through a modified lambda notation (Carnap, 1958), e.g.

\[
\psi := (2x, y, z)[c := x^2 + y^2; d := 5; e := x^2 + y^2 - d] = (c, e)
\]

where the parentheses hold the parameters, function block and results respectively. Three dots inside the last pair of parentheses means that all results are required. A function is 'called' by appearing in an expression, e.g. $\psi(3, 4, 5)$ for $\psi$ of equation (9).

(j) A locking (Boolean) element $L_1$ may be associated with a statement $S$ as in

\[
\downarrow b; S
\]

If $b$ is found 'reset', it is first 'set', then $S$ is executed, and finally $b$ reset again. If $b$ was found 'set', $S$ cannot be
executed at this time. Locking operations on the same element \( b \) must exclude each other in time.

(k) The ‘wait’ operator @ as in

\[ \@\{B\}; S \]  

(11)

delays the execution of \( S \) until (boolean) condition \( B \) is satisfied.

We note that all eleven operators above have been defined as machine independent. The sequential operators correspond to those used on present day computers. Ways to implement parallel operators (comma, lock and wait) on multiprocessor configurations have been suggested previously. Locking may be effected by a ‘test-and-set’ instruction (Lorin, 1972), a modified interrupt (Carver Hill, 1973), as \( P \) and \( V \) operations on semaphores (Dijkstra, 1968) or through the WAIT, POST, ENQ and DEC macros (of OS/360). The same WAIT macro may serve as a slightly modified wait operator. A parallel block may require interleaving of data in memory and accessing it through multiplexers (Wallach, 1974). The lock and wait operators are useful in system programming (Hansen, 1973), but seldom for numerical work. All operators connected with parallelism are tailored for use on a MIMD-type of multiprocessor (Stone, 1973).

This completes the notation; its merits will be made clear by the algorithms of the following sections.

3. Quadrature methods

In some cases (e.g. online control of a plant modelled as a linear, time invariant system), evaluation of an integral

\[ y = \int_{a}^{b} \phi(x) \cdot dx \]  

(12)

takes too long on present day computers. Due to the complexity of \( \phi(x) \), even methods which use a minimum number of function evaluations (Wallach, 1969) would be too slow. Hence, the need for a concurrent solution on a pps.

A method (Adams, 1970) specifically designed for parallel evaluation of \( y \) used special graphs to describe the algorithm. In the notation of Section 2, it can be defined succinctly and clearly by the use of just three functions. \( \rho \) sums two areas \((s_1, s_2)\) to yield \( s_3 \) and checks convergence

\[ \rho := (\lambda_1, s_2, s_1, e) \]  

\[ (s_3 := s_1 + s_2; b := \{s < s_3, s_3 \}^e, s_3) \]  

(13)

\[ \Rightarrow (s_3, b); \]

\( v \) computes a midpoint \( o \), the function \( f_o \) at \( o \) and the areas on its two sides:

\[ v := (\lambda_2, f_o, z, f_o) \]  

\[ \left[ \begin{array}{l}
\{o := a + 0.55[z + a, 0.25[z - a]]
\end{array} \right.
\]

\( f_0 := \phi(x = o); \)

\[ (s_4 := r^*(f_0 + f_o), s_5 := r^*(f_0 + f_o)] \]

\[ \Rightarrow (o, f_o, s_4, s_5); \]

\( \tau \) is a recursive function (i.e. it calls itself) which yields the integral in the form of the area computed by \( \rho \) if convergence was obtained, but calls itself twice if it was not obtained. It is in this last case that the algorithm ‘forks’ into parallel paths.

\[ \tau := (\lambda_3, f_o, z, f_o, s_5, e) \]  

\[ (o, f_o, s_4, s_5) := v(a, f_o, z, f_o); \]

\[ (s_3, b) := \rho(s_4, s_5, s, e); \]

\[ \Rightarrow (b). \]

\[ \left[ \begin{array}{l}
\{u := \tau(a, f_o, o, f_o, s_4, e),
\end{array} \right.
\]

\[ v := \tau(o, f_o, z, f_o, s_5, e));
\]

\[ w := u + v]. \]

\[ w := s_3 \]

(14)

Very effective, automatic quadrature methods are based on shifting the integration interval \((a, z)\) to \((-1,1)\) with a subsequent integration of the polynomial which interpolates \( \phi(x) \) at the Chebyshev points

\[ t_j := \cos(\frac{\pi j}{n}), j = 0, 1, 2, \ldots, n. \]

The basic procedure (Clenshaw and Curtis, 1960) approximates the integrand \( \phi(x) \) by a series of Chebyshev polynomials

\[ \sum c_j T_j(x) \]  

and produces the value of the definite integral from

\[ F = 2(d_1 + d_3 + d_5 + \ldots) \]

where the coefficients \( d_j \) result from integrating the first polynomial to get \( \sum d_j T_j(x) \).

The algorithm is well known and documented, so that a similar but much less known method (Filippi, 1964; Davis and Rabinowitz, 1967) will be discussed. It approximates

\[ F = \int_{-1}^{1} \phi(x) \cdot dx \]  

(17)

and not \( \phi(x) \) as a Chebyshev series at the interpolation points \( t_j \).

Since no algorithm for this method has been published yet, the schedule for its concurrent execution will be commented extensively and exemplified throughout.

The number of subdivisions \( n \) should be chosen so that not all values of the sines and cosines will have to be recomputed. This is best done by choosing a value ‘max’ (e.g. \( max = 4 \)) and starting the program through

\[ k := 2^{max - 1}; n := 5 * k; m := 4; \]

(e.g. \( k = 8 \); \( n = 40 \); \( m = 4 \)). The initial values of \( cn_v = \cos (\pi j/n), sn_v = \sin (\pi j/n) \) and \( g_v = sn_v \phi(cn_v) \) are computed in the block:

\[ \#j := k, 2k, \ldots, n + 2 \]

\[ [i := n - j; cn_v := \cos (\pi j/n); sn_v := -sn_v; \]

\[ g_v := \sqrt{1 - cn_v^2} \]

(18)

\[ g_v := g_v \phi(cn_v) \]

(e.g. \( cn_8 = \cos 36^\circ \); \( sn_8 = \sin 36^\circ \); \( g_8 = \phi(36^\circ) \)) concurrently with \( [cn_{16}, cn_{24}, cn_{16}, sn_{16}, sn_{24}, sn_{16}, g_{16}, g_{24}, g_{16}, g_{24}] \).

Next, three values \( d_n, d_{n-1}, d_{n-2} \) are computed:

\[ lb := 1, 2, 3 \]

\[ e := 2*cn_{n-1}, c_{-1} := c_v := 0; \]

\[ \#k := 1, 2, \ldots, m; [c := e^*c_v - c_{n-k} + \ldots + \] \( g_{n-k}; \)

\[ c_{-1} := c_{n-k}; c := e]; \]

\[ d_{n+1-1} := (2*cn_{n-1}*c)((m + 1) \cdot (m + 1 - l)); \]

(19)

In the above, the \( c \)’s are computed from a marching recurrence relation. Since every \( c_k \) depends on \( c_{k-1} \) and \( c_{k-2} \), and since \( h \) loops from 1 to \( m \), it seems as if there is only one way to calculate the \( c \)’s: sequentially. However, it is shown in the Appendix that even such ‘inherently sequential’ calculations may be successfully parallelised.

If all three \( d \)’s are small, the value of the integral is approximately

\[ \int_{a}^{b} \phi(x) \cdot dx \approx (x - a) \cdot (d_1 + d_3 + \ldots + d_t) \]  

(20)

with \( r \leq n \). Otherwise, the \( m \) and \( k \) values are changed according to

\[ m := 2m + 1; k := k + 2; \]

(21)

and the new \( cn, sn, g \) are calculated not from equation (18) but through:

\[ cn := \sqrt{((1 + cn_{2k})/2); e := n - k; cn_e := -cn_e; \]

\[ sn := sn_e := \sqrt{((1 - cn_e^2)/2)}; g_e := sn_e \phi(cn_e); \]

\[ \#j := m - 1, m - 2, \ldots, 2 \]

\[ [\{rem 2 \neq 0 \}] \rightarrow \]
\[ l := j^*k; \quad cn_l := (cn_{l+k} + cn_{l-k})/(2*cn_k); \]
\[ (l < n/2) \rightarrow sn_l := sn_{2l}/(2*cn_l); \]
\[ sn_l := \sqrt{1 - cn_l^2}; \quad g_l := sn_l^*\phi(cn_l) \]
\[ ] \]
For \( m = 9, k = 4, e = 36 \) this computes first
\[ [cn_4; cn_{36}; sn_4; sn_{36}; g_{20}] \]
and then, concurrently \([cn_{28}; sn_{28}; g_{28}]\) with \([cn_{20}; sn_{20}; g_{20}]\) with \([cn_{12}; sn_{12}; g_{12}]\). Finally \( g_4 \) (e.g. \( g_4 \)) is computed and the program returns to label \( lb \):
\[ g_4 := sn_4^*\phi(cn_4); \quad > lb; \]
\[ (23) \]
This program exhibits so much parallelism that it should compare favourable with the previously mentioned concurrent method. This is especially true if the recurrent calculation of \( c_n \) is not done per equation (19) but in the way shown in the Appendix. A final remark concerns the evaluation of the integral equation (12) using the Romberg method. Both in the general case (Bauer et al., 1963; Krasun and Prager, 1965) and in the particular case of a ‘state model’ (Wallach, 1969), the notation of Section 2 lead easily to models for parallel execution.

4. Matrix inversion
One of the older algorithms for inverting a matrix \( A \) (Shipley and Coleman, 1959) is based on the theory of ‘diakoptics’ (Kron, 1939). It consists of choosing pivots \( A;i \) and substituting all elements \( A;k \) not on the pivoting row or column by
\[ A;jk := A;jk - A;ji*A;k;i \quad (24) \]
The element \( A;i \) is replaced by \( c := -1/A;i \) and the remaining \((2n - 2)\) elements of row \( i \) and column \( i \) are multiplied by \( c \). The pseudo-program of this algorithm is thus:
\[ \# i := 1; 2, \ldots \; n \]
\[ [c := -1/A;i;i; \# j := 1, 2, \ldots \; n \]
\[ [\# k := 1, 2, \ldots \; n \]
\[ [(j \neq i) \rightarrow A;jk := A;jk + c*A;ji*A;k;i]; \]
\[ \# j := 1, 2, \ldots \; m \]
\[ [A;i,j := c*A;i,j]; A;i,i := c \]
\[ ] \]
This substitution requires two multiplicative operations for each of the \((n^2 - 2n + 1)\) elements. The remaining \((2n - 2)\) elements are multiplied by a constant \( c \). Hence, the total number of multiplications is:
\[ \mu = n(2(n^2 - 2n + 1) + 2n - 2) = 2n^2(n - 1) \approx 2n^2 \]
This is twice the number required in Crout’s algorithm (Kunz, 1957), which probably explains why this algorithm was seldom used outside the field of the electrical power industry. It is similar to the elimination method of Gauss, and the main reason why we prefer it essentially the following. Equation (24) may be performed in any order, which means that processor scheduling and data storage will not be a problem (as it certainly is in other methods, e.g., those suggested for ILLIAC-IV).
It was noted (Pease, 1967) that: ‘With parallel processing, the total number of operations is not significant. What matters is the number of sets of operations, where each set involves those being done in parallel’. Since the entire inner loop (of \( j \) and \( k \)) may be performed concurrently, this set equals:
\[ \sigma \approx \frac{2n^3}{p} \]
where \( p \) is the number of processors. Hence, \( \sigma \) would be only
\[ \begin{array}{cccc}
\text{Fig. 1 Elements which change at iterations } i = 1, 2, 3 \\
\begin{array}{cccc}
\lfloor x x x x x x \ldots \rfloor & \lfloor x x x x x x \ldots \rfloor & \lfloor x x x x x x \ldots \rfloor \\
\{ i = 1 \} & \{ i = 2 \} & \{ i = 3 \}
\end{array}
\end{array} \]
\[ 2n \text{ for a system of } p = n^2 \text{ processors. The second advantage is the small } \sigma. \]
As a fringe benefit, note that the use of a minus on the pivoted term causes a symmetric matrix to remain symmetrical throughout the entire process.
Another advantage claimed for Kron’s algorithm is that it is useful in inverting sparse matrices. We will apply it to the best-known case, namely to a tridiagonal matrix which is basic to the numerical solution of elliptic, partial differential equations (Fox, 1962). We could adapt equation (25) to this case by modifying the Boolean condition for which the inner loop (of \( k \) and \( j \)) is performed. Since this point is traversed \( n^2 \) times (or by \( n^2 \) processors, once) and since, anyway, we have to assume the number of processors to be smaller than \( n^2 \), it is precisely checking of this Boolean condition that we would like to avoid. Fig. 1 and the following pseudo-program show how this can be done:
\[ \# i := 1; 2; \ldots \; n \]
\[ [c := -1/A;i;i; \# j := 1, 2, \ldots \; n \]
\[ [\# k := 1, 2, \ldots \; n \]
\[ [(j \neq i) \rightarrow A;jk := A;jk + c*A;ji*A;k;i]; \]
\[ \# j := 1, 2, \ldots \; l \]
\[ [(j = i) \rightarrow A;jm := c*A;jm; A;jm := c*A;jm; \]
\[ (j \neq i) \rightarrow \]
\[ A;jm := c*A;jm; A;jm := c*A;jm; \]
\[ \# j := 1, 2, \ldots \; l \]
\[ [A;i,j := c*A;i,j]; A;i,j := c*A;i,j; \]
\[ A;i,i := c \]
\[ ] \]
For its implementation, one row and one column of zeros should be added to the original matrix. The number of sets \( \sigma \) will be larger than that for an adapted equation (25), but there is enough parallelism left in equation (25) to keep all processors busy most of the time. The number of operations on elements is:
\[ 2^2 + 3^2 + 4^2 + \ldots + n^2 = -1 + \sum_{k=1}^{n} k^2 = -1 + \frac{n(n + 1)(2n + 1)}{6} \approx \frac{n^3}{3} \]
This is a large number, but, for large \( n \), most of the operations will be in the \( 2n \) multiplications of the inner loop (for \( j \) and \( k \)), which corresponds to equation (24).
Concurrency may also be used for inverting a reordered tridiagonal matrix or for solving lower and upper bidiagonal, as well as blockwise tridiagonal matrices (Even and Wallach, 1970).
The notation introduced in Section 2 may also be used to describe Gaussian elimination or the ‘bordering’ method on specially designed, parallel computer systems (Pease, 1967). For the ‘bordering’ method, and generally for linear algebraic problems, the ability to compute concurrently the scalar product of two vectors \( a \) and \( b \) is important. It should be noted that one method which should not be applied, is:
The results of performing equation (30) for more than one index concurrently would depend on the timing of the computers and make the computation indeterminate. To restore determinacy, \( s \) may be accessed only after it is locked, by \( d \), e.g.

\[
\begin{align*}
\# i &= 1, 2, \ldots, n[c_i := a_i b_i]; \\
\# j &= 0; 1; \ldots; (\log_2 n - 1) \\
[m] &= n/2^j; \\
\# i &= 1, 2, \ldots, m/2[c_i := c_1 + c_{m+1-i}]
\end{align*}
\]

(31)

Since this would slow down the process considerably, the following method is to be preferred for calculating the scalar product of \( a \times b \):

\[
\begin{align*}
\# i &= 1, 2, \ldots, n[c_i := a_i b_i]; \\
\# j &= 0; 1; \ldots; (\log_2 n - 1) \\
[m] &= n/2^j; \\
\# i &= 1, 2, \ldots, m/2[c_i := c_1 + c_{m+1-i}]
\end{align*}
\]

(32)

In equation (32), the initial dimension \( n \) of the two vectors should be a power of 2, i.e. \( n = 2^l \), but if it is not such, then we may set all

\[
a_h := b_h := 0 \text{ for } h = 2^l - n.
\]

(33)

Equation (32) halves successively the size of vector \( c \) and leaves the result in \( c_1 \).

Appendix

The recurrence relation used in equation (19) has the general form:

\[
c_k = p c_{k-1} + q c_{k-2} + r_k, h = 1, 2, \ldots, n
\]

(A-1)

where \( c_{-1}, c_0 \) are known initial values and \( p, q, r_k \) are known constants. Element \( c_h \) is to be computed—if possible, in a parallel way.

Lemma:

For any positive integer \( h \), \( c_h \) can be written as

\[
c_h = U c_{h-1} + V c_{h-2} + W
\]

(A-2)

where \( U, V, W \) are constants, i.e. \( c_h \) is a linear form in \( c_{-1} \) and \( c_0 \).

Proof (by induction):

For \( h = 1 \), the constants \( U, V, W \) are \( p, q \) and \( r_1 \) respectively.

For \( h = 2 \), they are found (by insertion) to be \((p^2 + q), (pq)\) and \((pr_1 + r_2)\) respectively. Assume next that equation (A-2) holds for some \( 2 < i < n \). Then

\[
c_{i-1} = U' c_{i} + V' c_{i-1} + W'
\]

(A-3)

\[
c_{i-2} = U'' c_{i} + V'' c_{i-1} + W''
\]

(A-4)

Inserting these two into equation (A-1) yields for \( h = i \) equation (A-2) with

\[
U = pU' + qU''
\]

(A-5)

\[
V = pV' + qV''
\]

(A-6)

\[
W = pW' + qW'' + r_k
\]

(A-7)

QED.

Let us split the \( n \) relations (A-1) into \( m \) contiguous blocks as depicted in the figure below. Block \( j \) starts with a \( c_h \) whose index is \( a \equiv I_j \) and ends with a \( c_h \) whose index is \( b \equiv I_{j+1} - 1 \). By a shift of origin, and by virtue of the lemma, for this block:

\[
c_{i-2} c_{i-1} c_i c_{i+1} c_{i+2} c_{i+3} c_{i+4} c_{i+5} c_{i+6} c_{i+7} c_{i+8} c_{i+9} c_{i+10} c_{i+11} c_{i+12}
\]

\[
l_1 = l_2 = l_3 = \ldots = l_m
\]

(A-8)

\[
c_{h-2} = U_j c_{h-1} + V_j c_{h-2} + W_j
\]

(A-9)

(Note that for \( b = i, a = 1 \) these equations reduce to (A-3) and (A-4).)

As is seen, the last two elements of a block are linear forms in the last two elements of the preceding block.

The algorithm for computing \( n \) concurrently is as follows: for each block \( I_a \) assume arbitrarily three pairs of initial values \((c_{-2}, c_1), (c_{-1}, c_0), (v, w)\), i.e. \( (1, 2, 3) \). For each pair a processor (concurrent with and independently of the other processors) applies equation (A-1) until the values \( c_{-2}, c_0, w \) are computed. As a result three linear equations (A-8) in the unknowns \( U_j, V_j, W_j \) are obtained and another three in \( U^*, V^*, W^* \). A judicious choice would be the following three pairs: \((1, 0), (0, 1), (0, 0)\) which yield:

\[
\begin{bmatrix}
1 & 0 & 1 \\
0 & 1 & 0 \\
0 & 0 & 1
\end{bmatrix}
\begin{bmatrix}
U_j \\
V_j \\
W_j
\end{bmatrix} =
\begin{bmatrix}
c_{-2}^1 \\
c_0^1 \\
c_2^1
\end{bmatrix}
\]

(A-10)

From these, the solution is

\[
U^*_k = c_{k+1}^1 - c_{k+1}^0
\]

(A-11)

\[
V^*_k = c_{k+1}^0 - c_{k+2}^0
\]

(A-12)

\[
W^*_k = c_{k+2}^0
\]

(A-13)

and similarly for \( U^*, V^*, W^* \).

Inserting the (known) values \( c_{k+1}, c_{k+2} \) into equations (A-8, A-9), yields \( c_{i-2} \) and \( c_{i-1} \). Proceeding sequentially in Fig. A-1 yields \( c_h \) in \( m - 1 \) steps. It should be noted that the three sweeps in each block may proceed concurrently, which increases parallelism even more.

Acknowledgements

The author wishes to thank Mr. V. Conrad, Dr. D. W. Martin and Professor M. Yoei for their help in preparing and revising the manuscript.

References


Book reviews


From 1 to 5 September 1975, the 4th annual symposium on the Mathematical Foundations of Computer Science (MFCS) took place in Mariánske Lázne, Czechoslovakia; its proceedings, edited by J. Bečvár appeared as Lecture Notes in Computer Science nr 32, Springer-Verlag. This meeting emphasised complexity theory.
The annual MFCS symposia are organised alternatively in Czechoslovakia and in Poland by the computer centres of the Academies of Science of these countries. The invited lectures and communications covered respectively constitute recent results in automata theory, complexity theory, Lindemayer systems, logic of computation and mathematical linguistics, on the one hand and the theory of data bases, methods for program proving, Petrinets and semantics of programming languages, on the other.
The proceedings contain a considerable number of communications. Being a 'correctness prover' myself, I restrict myself mainly to the subject implied: Mazurkiewicz presentation is related to work done earlier by H. Bekic. R. V. Freivald contributed: Minimal Gödel numbers and their identification in the limit—recursive function theory. I. M. Havel: Nondeterministically recognisable sets of languages—solid work. M. Karpinski: Decision algorithm for Havel's branching automata—if correct, then interesting. W. P. de Roever: First-order reduction of call-by-name to call-by-value—explains why call-by-name leads to a theory of termination which is exponentially more complicated than call-by-value. M. B. Trakhtenbrot: On representation of sequential and parallel functions—provides a glimpse of Russian work on a subject introduced mainly by Vuillemin. F. Kröger: Formalisation of algorithmic reasoning—already known, but in the right direction, as turned out later.
These symposia constitute to my knowledge the only regular meeting place between theoretical computer scientists of Eastern Europe and the Western world, and are therefore of vital importance to our East-European colleagues.

Willem P. de Roever, (Belfast and Amsterdam)

Computing Systems Hardware, by M. Wells, 1976; 245 pages. (Cambridge Computer Science Texts, 6, £4.00)

This paperback is one of the new breed of books about computer techniques which attempt to short-circuit much of the conventional analysis in order to give an overall picture of a complex subject in a manageable book. When one reads this book the first impression is of astonishment that the different subjects could have been fastened so closely together and be contained in just over 200 pages. The text is intended for second year undergraduates (third year in Scotland), studying computer science or computer engineering, and there is no doubt that it will be very useful for general reading in this area. However it is easy for someone who has struggled through the concepts of computer technology the hard way to consider the book an excellent summary. The danger is of course that like Pitman's shorthand it may be too condensed for the inexperienced reader, and it is fair to say that certain parts of the text are superficial in the sense that if the student went straight from the text to practical equipment he would be somewhat lost. A clear example is the brief way that TTL circuits are described, without reference to books like the Texas Instruments classic.

Having said this, there is a great deal to enjoy in the book, and it does approach its subject from the standpoint of information science. There are many different ways of looking at standard things, and some enjoyable prose. All in all a good book for use by anyone in conjunction with the services of a good lecturer. In the absence of a guide, then the reader would be prepared to buy a good dictionary of computing terms, because sometimes the specialist words come thick and fast.

F. G. Heath (Edinburgh)