ALGORITHMS FOR PARALLEL COMPUTERS

R. F. Churchhouse
University College, Cardiff, United Kingdom

ABSTRACT

Until relatively recently almost all the algorithms for use on computers had been designed on the (usually unstated) assumption that they were to be run on single processor, serial machines. With the introduction of vector processors, array processors and interconnected systems of mainframes, minis and micros, however, various forms of parallelism have become available. The advantage of parallelism is that it offers increased overall processing speed but it also raises some fundamental questions, including:

(i) which, if any, of the existing "serial" algorithms can be adapted for use in the parallel mode?
(ii) how close to optimal can such adapted algorithms be and, where relevant, what are the convergence criteria?
(iii) how can we design new algorithms specifically for parallel systems?
(iv) for multi-processor systems how can we handle the software aspects of the interprocessor communications?

Aspects of these questions illustrated by examples are considered in these lectures.

1. INTRODUCTION

Parallelism in a computer system means doing more than one thing at a time. The idea of parallelism in the context of computers can be traced back to a conversation of Charles Babbage in 1842 [1], as Professor Zacharov pointed out in his lectures to the 1982 CERN School of Computing [2]. As for its implementation: parallel handling of the bits of a word in arithmetic and logical operations was normal on most of the early electronic computers and parallel execution of I/O and computation was available on various general-purpose computers by about 1960. The overlapping of I/O and computation was built into the architecture of the machines and was achieved either by interrupts (hardware) or by polling (software).

For the purpose of these lectures "a parallel computer system" is defined to be a system of p(>1) communicating processors or processing elements (P.E's), not necessarily identical, all of which can be in operation simultaneously, possibly under some form of central control. It may also be assumed that each P.E. has some private memory and has access to some common memory. This is (deliberately) a very vague definition: the "processors or processing elements", for example, may be anything from Cray XMPs to "AND gates"; in particular they may be simple, but specialised, devices such as those shown in Figure 1. The communication paths may be radial, nearest-neighbour, anything-to-anything or designed around some fancy geometry; the central control may dictate all the operations and enforce strict synchronization, or it may be almost non-existent.

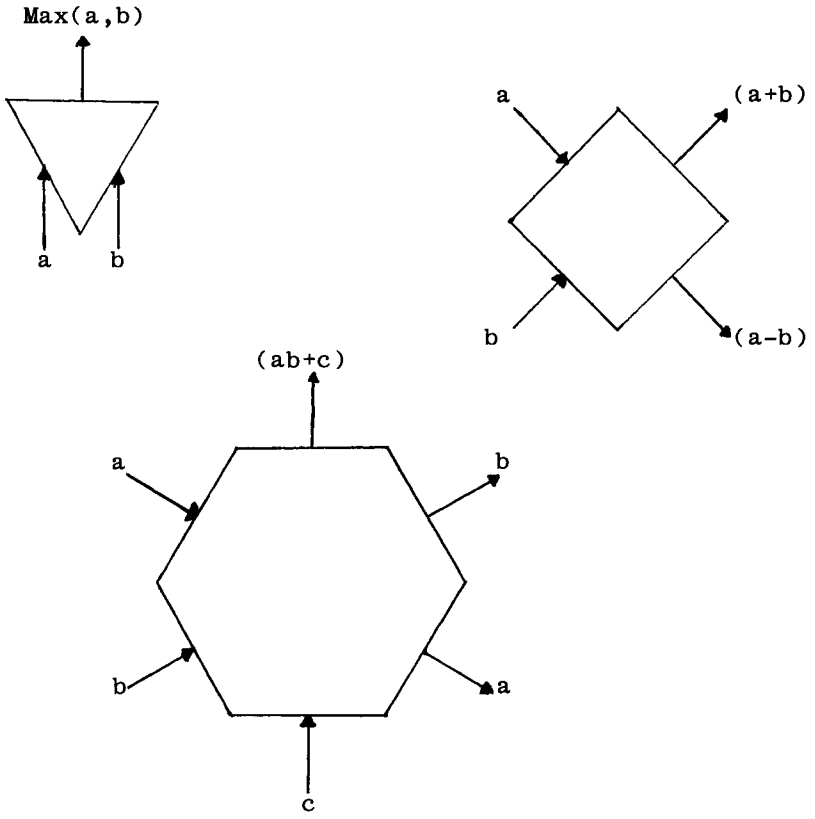


Figure 1.

If the P.E's all execute the same instructions in synchrony we have what Flynn [3] has called a Single-Instruction/Multiple-Data (SIMD) system. If the PE's can execute different instructions, whether synchronously or asynchronously, we have a Multiple-Instruction/Multiple-Data (MIMD) system. Examples of SIMD Systems include Illiac IV, the ICL-DAP, the Goodyear-NASA Massively Parallel Processor, the Cray-1 and the Cyber 205. We shall not be concerned with systems such as the Cray-1 or Cyber 205 or DAP in these lectures; they are (relatively) widely available and there are a very large number of papers on how to take advantage of their pipeline, vector or array facilities in a wide range of applications; see, for example [4, 5, 6].

MIMD Systems include multi-mini systems such as C.mmp, based on 16 PDP-11's, at Carnegie-Mellon University, the PACS-32 at Tsukuba University and the Neptune System at Loughborough University. We give more detail on these systems in Section 5.

A simple taxonomy of parallel computers which may be helpful, based on one given by Jordan [7], can be represented by the diagram below:

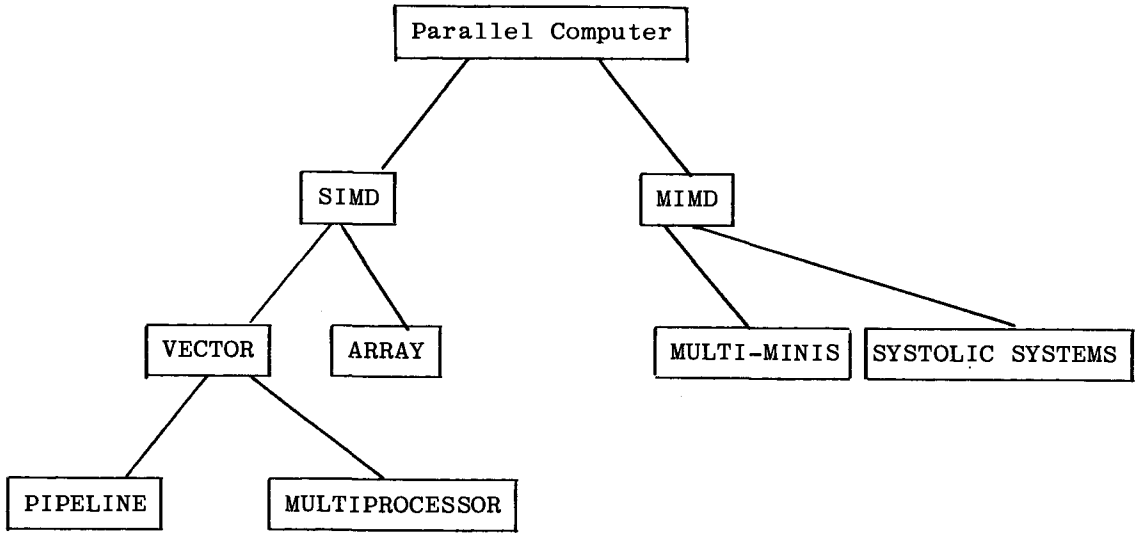


Figure 2.

The major reason for constructing parallel computer systems is, of course, that eventually the physical limitations imposed by the speed of light and switching times of elements will set a limit to the speed of computation of any serial computer, but this does not mean that the need for parallelism is based solely upon a desire to reduce the running-time of very long-running programs. There are other reasons, one is to increase availability and safety, whilst another arises in real-time environments where, say, a 10-millisecond response may be acceptable whereas a 100-millisecond response may be potentially disastrous.

There are then good reasons for trying to combine processors together and constructing algorithms to utilize them in parallel. Whether, however, we can increase the overall computation speed by an arbitrarily large factor by using a sufficiently large number of processors in parallel is an interesting question - the inter-processor communication times, which in the worse cases increase with p^2, may eventually dominate the computation times which can at best only be expected to decrease proportionally to p^{-1}. Even for quite modest multi-mini systems the communication times may be very significant. Having raised this question we will not consider it further here, but it is important to stress that the communication times may impose very significant overheads in practice.

In these lectures we shall look at some examples of parallel algorithms and some implementations of parallel systems but first we introduce some basic concepts.

2. BASIC CONCEPTS

A program is considered to be composed of a number of processes, which it controls; the program itself may be controlled by an operating system. At any given time any particular process may be in execution in at most one processor, not necessarily the same processor at different times.
A parallel algorithm is an algorithm for the execution of a program which involves at least intermittent running of two or more processes on two or more processors simultaneously.

A synchronized parallel algorithm is a parallel algorithm which involves at least one process which is not permitted to enter a certain stage of its activity until another process has completed a certain stage of its own activity.

If T_1 is the time taken to run the 'best' serial algorithm for a particular program on a single processor and if T_p is the time taken by a parallel algorithm for the same program on a p-processor system then:

the speed-up ratio is : $S_p = \frac{T_1}{T_p}$

and the efficiency of the parallel algorithm is: $E_p = \frac{S_p}{p}$.
Sometimes we are interested in the speed-up and efficiency of a parallel algorithm compared to a serial version of the same algorithm, rather than the "best" algorithm. In such a case we might use the terms "relative speed-up" and "relative efficiency".

Example Suppose we wish to compute xy+log(x+y) using two PE's; if the computation of xy takes t_1 units of time and the computation of log(x+y) takes t_2 units, where an addition takes 1 unit, and where (as is very likely) $t_1<t_2$, the computation is achieved in (t_2+1) time units by the synchronized parallel algorithm

Time	Activity
0	x,y→PE1; x,y→PE2
t_1	(xy) available from PE1; PE1 idles until input received from PE2.
t_2	log(x+y) sent from PE2 to PE1
(t_2+1)	(xy)+log(x+y) available from PE1

On a single PE the total time for this computation would be (t_1+t_2+1) units (ignoring storage times) so the speed-up ratio is $(t_1+t_2+1)/(t_2+1)$ in this case. Notice that this ratio must be less than 2, since we have assumed that $t_1<t_2$, and hence the efficiency is less than 1. This result is typical: for a p processor system the speed-up factor will almost always be less than p, and the efficiency less than 1. There are, however, cases where a relative speed-up factor greater than p, i.e. a relative efficiency greater than 1, can be achieved (see Section 4.6) .

An asynchronous parallel algorithm is a parallel algorithm with the following properties:

(i) there is a set of global variables to which all processes have access;

(ii) when a stage of a process has been completed the process first reads some global variables (possibly none) then, depending upon the values of these variables and the results just obtained from the last stage, the program modifies some global variables (possibly none) and then either activates its own next stage or terminates itself. In order to ensure logical correctness the operations on global variables may need to be programmed as critical sections [9].

Thus in asynchronous algorithms communications between processes are achieved by means of global variables. There is no explicit dependency between processes, as occurs in synchronized algorithms; asynchronous algorithms have the characteristic that they never wait for inputs but proceed according to the values of the global variables. Note, however, that some processes may have critical sections to which entry is blocked temporarily, as mentioned above [10]. Although the analogy isn't perfect we can think of synchrony as involving interrupts (hardware) and asynchrony as involving polling (software).

Apart from purely theoretical interest the main reason for the construction and analysis of asynchronous algorithms is that parallel algorithms are unlikely to succeed in keeping all P.E's equally busy and so the introduction of synchronization will inevitably lead to a slow-down in overall performance. Unfortunately a fully asynchronous parallel algorithm on several processors involving several global variables might well defy analysis, so that we would be unwise to use it in practice; it is, for instance, quite possible that the algorithm might converge with one assignment of processors and fail to converge with another assignment, even with identical initial inputs, as the experiment described in the next section illustrates.

3. SIMULATION OF THE NEWTON-RAPHSON ALGORITHM USING THREE P.E.'s

in (i) serial mode, (ii) synchronous parallel mode, (iii) asynchronous parallel mode.

In this simulation the Newton-Raphson algorithm, defined by

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)} \qquad (3.1)$$

is used to solve $f(x) = 0$, from a given starting value x_0, using three PEs which function as follows:

(i) Given x, PE1 computes f(x) in time t_1 units and sends it to PE3,

(ii) Given y, PE2 computes f'(y) in time t_2 units and sends it to PE3,

(iii) Given a,b,c PE3 computes $d=a-b/c$; if $|d-a|$ exceeds a preset threshold the value of d is sent to PE1 and PE2, otherwise the value of d is printed and the program halts. This all takes 1 unit of time.*

(*)There is no loss of generality in this, if we are prepared to accept non-integral values of t_1 and t_2; for if $t_3 \neq 1$ we divide t_1, t_2, t_3 by t_3 and the conclusions are unaltered.

(i) Serial mode. Given x_n as input PE1 computes $f(x_n)$, after which PE2 computes $f'(x_n)$, after which PE3 computes x_{n+1} from (3.1), tests $|x_{n+1}-x_n|$ and takes appropriate action. The time taken for each such iteration is (t_1+t_2+1) units. If k such iterations are required the total time for completion will be $T_{SM}=k(t_1+t_2+1)$ units.

(ii) Synchronous parallel mode x_n is provided as input to both PE1 and PE2 simultaneously. When both have completed their tasks the values of x_n, $f(x_n)$ and $f'(x_n)$ are used by PE3 as in the serial case. The time taken for each such iteration is

$$T_{SP} = (\text{Max}(t_1,t_2) + 1) \text{ units.}$$

The number of iterations required will also be k so the total time to completion will be

$$T_{SP} = k(\text{Max}(t_1,t_2) + 1) \text{ units.}$$

(iii) Asynchronous parallel mode PE1 and PE2 begin computing as soon as a new value of the input variable is made available to them by PE3 and they are ready to receive it. PE3 computes a new value, using (3.1) in the form

$$x_{n+1} = x_n - f(x_i)/f'(x_j) \quad * \qquad (3.2)$$

as soon as either PE1 or PE2 provides a new input, and tests $|x_{n+1}-x_n|$ etc. as in the serial case.

The time for each iteration will be at most

$$(\text{Min } (t_1,t_2)+1) \text{ unit}$$

but it may be as low as 2 units (e.g. if PE1 and PE2 complete at consecutive time steps). In this case we cannot predict how many iterations will be required; it is unlikely to be less than k and may be significantly more - including, as we see below, infinity (i.e. the algorithm never terminates).

A program to carry out this simulation under these three modes for various sets of values of t_1 and t_2 for the function $f(x) = x^2-2$ with $x_o=1.0$ produced the following results; the threshold was $\frac{1}{2} \times 10^{-7}$ and k=4, in this case.

t_1	t_2	T_{SM}	T_{SP}	T_{AS}	S_S	S_A
1	5	28	24	20	1.17	1.40
2	5	32	24	21	1.33	1.52
3	5	36	24	38	1.50	0.95
4	5	40	24	31	1.67	1.29
5	5	44	24	30	1.83	1.47
6	5	48	28	∞	1.71	0

(T_{SM},T_{SP},T_{AS} are times to completion in serial, synchronous parallel, asynchronous parallel modes; S_S and S_A are the relative speed-up factors for the synchronous and asynchronous modes).

*(the values of (n-i) and (n-j) are bounded, with explicit bounds, but that needn't concern us here).

In the case $t_1=6$, $t_2=5$ the algorithm fails to converge in the asynchronous parallel case; the system falls into an approximate cycle of length 36 time units during which six new approximations to the solution are obtained, none of them correct to 1 d.p. The reason for this is that the convergence of the N-R algorithm depends critically upon the value taken at each stage for $f(x_n)$ but depends to a much lesser extent on the value taken for $f'(x_n)$, unless this happens to be very small. Use of $f(x_n)$ rather than $f(x_{n+1})$ can cause the correction term to have the wrong sign whereas use of $f'(x_n)$ instead of $f'(x_{n+1})$ will usually leave the algorithm convergent but linearly, not quadratically. By taking $t_1>t_2$ it is inevitable that we will sometimes use a value for $f(x_n)$ which has the wrong sign, so destroying the convergence.

The main purpose of the example above is to illustrate an important point; an algorithm which is quite satisfactory on a serial machine, or even on a synchronized parallel system, may fail completely on an asynchronous system. Even if the asynchronous version ultimately terminates satisfactorily it may take longer than a synchronized, or even a serial, version. Whilst there are serial algorithms, such as the Gauss-Seidel and Jacobi methods for solving systems of linear equations, which can be run successfully on asynchronous parallel systems (see, for example the paper by Baudet [11]), in general this will not be the case. We conclude therefore that, in the majority of cases, algorithms for parallel computers, whether synchronous or not, will need to be specially designed and not merely be adaptations of existing serial algorithms.

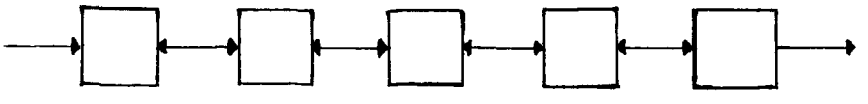
4. SOME COMMUNICATION GEOMETRIES AND PARALLEL ALGORITHMS

In this section two algorithms which have been adapted to MIMD systems are described. The first is a simple sorting algorithm which, in its parallel form, is naturally of synchronous type; the second is a generalisation of the Binary Search algorithm and will be given both in synchronous and asynchronous form.

As a preliminary it is convenient at this point to explain the nature of some communication geometries which occur frequently both in the literature, and in practice.

4.1 Linear communication

In these systems the P.E's are assumed to be arranged (from a geometrical, if not from a physical, point of view) in a line, viz:



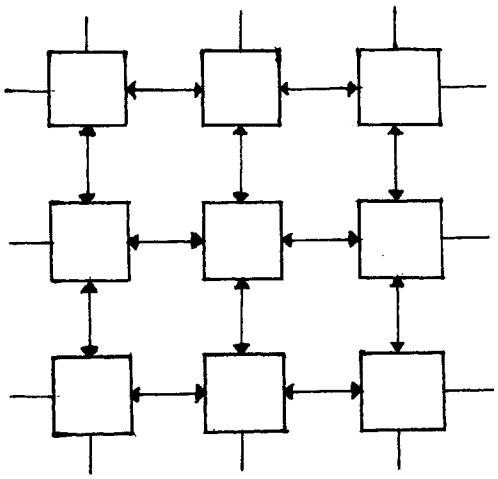
For $2 \leq k \leq (n-1)$ PE(k) can communicate with PE(k-1) and PE(k+1). PE(1) can communicate with PE(2) and PE(n) can communicate with PE(n-1).
If PE(1) and PE(n) can communicate with each other we have
Ring communication otherwise we have One-dimensional Linear Communication.

Simple though such a system is it suffices for many applications and its suitability for shift registers and pipelines is apparent.

4.2 Nearest-neighbours, two-dimensional arrays

In this system the PE's are envisaged as lying at the points to a two-dimensional rectangular lattice. Each row and each column has one-dimensional linear communication so that each P.E is connected to 4, 3 or 2 other PE's depending on whether it lies at an internal lattice point, a boundary lattice point or a corner lattice point.

Thus a 3x3 array can be represented



As in the linear-case, "end-around" communication 'horizontally' and/or 'vertically' is sometimes included, giving rise to 'cylindrical' or 'toroidal' communication geometries.

4.3 Two-dimensional hexagonal geometry

Two-dimensional space can be completely filled by (i) identical rectangles, (ii) identical equilateral triangles, (iii) identical hexagons. Kung [12] has utilised hexagonal symmetry to produce an elegant parallel processing system for multiplying two-dimensional matrices, as we shall see in Section 4.7 where this system will also be described.

4.4. The Odd-Even Transposition Sort

This is one of the simplest methods of sorting; on a uniprocessor it has the advantage of being easy to program and the disadvantage of being very inefficient, since the number of operations required to sort N items is $O(N^2)$, so that it is unlikely to be used if N is more than a few thousand. In a parallel mode however it is quite interesting because the sort can be easily performed by a collection of very simple PE's; with N such PE's the sorting time is reduced to O(N) units so that for special

purpose applications, where a few thousand items must be sorted quickly, a special-purpose device could be economic.

For simplicity we suppose that N, the number of items to be sorted, is even and write N=2m. The odd-even transposition sort proceeds as follows:

Odd phase: For i=1 to m compare the items in positions (2i-1) and (2i) and interchange them if they are in the wrong order.

Even phase: For i=1 to (m-1) compare the items in positions (2i) and (2i+1) and interchange them if they are in the wrong order.

Repeat both phases m times.

The list will then be in sorted order. The total number of comparisons/interchanges will be m(2m-1) so, if we define a comparison/interchange to take unit time the total time required will be about $\frac{1}{2}N^2$ inits.

If we connect together N PE's in a linear array, without end-around communication, as in 4.1, where PE(k), when activated, compares the item in its store with the item in the store of PE(k+1) and interchanges them if they are out of order then the synchronized parallel odd-even transposition sort on N PE's proceeds as follows:

Odd phase: Activate all odd-numbered PE's in parallel.

Even phase: Activate all even-numbered PE's, except PE(N), in parallel.

Repeat both phases m times

The list will then be in sorted order.

The total time required, ignoring any synchronization overheads, is N units. The speed-up factor is therefore about $\frac{1}{2}N$ and the efficiency about $1/2$.

The parallel system described above is a simple example of what Kung calls "a systolic architecture" [13]. "A systolic system consists of a set of interconnected cells, each capable of performing some simple operation.... Information in a systolic system flows between cells in a pipelined fashion and communication with the outside world occurs only at the "boundary cells". For example, in a systolic array, only those cells on the array boundaries may be I/O ports for the system". In a systolic system data and intermediate results flow between the PE's in a rhythmic fashion, analogous to the flow of blood within the body, hence the name.

Many standard sorting methods have been re-designed for parallel systems and sorting times of orders O(N), $O(N^{\frac{1}{2}})$, $O(\log^2 N)$ and O(logN) achieved, the faster methods usually requiring more PE's and/or more complex intercommunication and control. For a more sophisticated version of the odd-even sort see a recent paper by Wong and Ito [14], for other methods see [15].

4.5 Binary Search

This algorithm is perhaps the simplest both for finding an item in an ordered list and (under the name "Bisection Method") for finding a zero of a

continuous function to a specified precision, given that it is known to lie in some interval. The analysis is essentially the same for both cases; we shall describe it for the latter case. The algorithm is certain to succeed and although there are more efficient algorithms this one has the advantage that we can predict very accurately how many iterations will be required.

We suppose that we have a continuous function, f(x) and that we have two points x_o and x_1 such that $f(x_o)<0$ and $f(x_1)>0$; we therefore know that f(x) has a zero in the interval $\langle x_o, x_1 \rangle$ of length $|x_1-x_o|$. The Bisection Algorithm in serial form proceeds as follows:

(1) Evaluate $f(\frac{x_o+x_1}{2})$; if the value is zero go to the print routine, if it is negative replace x_o by $\frac{1}{2}(x_o+x_1)$, if it is positive replace x_1 by $\frac{1}{2}(x_o+x_1)$;

(2) If $|x_o-x_1|$ is sufficiently small go to the print routine, otherwise go back to (1).

The interval in which the zero lies is halved in length at each iteration so the number of iterations required to improve the accuracy from k to (k+n) decimal places will be

$$\lceil n \log_2 10 \rceil$$

(i.e. the integer greater than or equal to $\frac{10n}{3}$).

If we have a parallel system composed of p P.E's the obvious parallel equivalent of the Bisection Algorithm involves dividing the interval containing the zero into (p+1) sub-intervals of equal length at each iteration. The function, f(x), is evaluated at the (p+1) division points simultaneously and when all the evaluations have been completed the choice of the new interval is made. Synchronization is clearly essential for this algorithm and there will be time penalties associated both with the synchronization and with choice of the new interval. If we ignore these overheads the number of iterations required to improve the accuracy from k to (k+n) decimal places will be

$$\lceil n \log_{(p+1)} 10 \rceil$$

- but because of the overheads it will be more than this. The speed-up factor of this synchronized parallel algorithm is therefore at most

$$\frac{\lceil n \log_2 10 \rceil}{\lceil n \log_{(p+1)} 10 \rceil} \doteq \frac{\log_2 10}{\log_{(p+1)} 10} = \log_2 (p+1)$$

and the efficiency is $< \dfrac{\log_2 (p+1)}{p}$

which decreases as p increases, so this algorithm is likely to give better value for money, so to speak, when p is fairly small. For p=4, for example, it gives an efficiency of at most 0.57.

The Binary Search/Bisection Algorithm has received considerable attention in the literature and variations on the fully synchronized parallel version described have been published.

4.6 Binary Search : Asynchronous version using two P.E's.

If we have two P.E's we can execute the synchronized binary search algorithm of section 4.5, achieving a speed-up factor, ignoring overheads due to synchronization, etc., of $\log_2 3 (\doteq 1.59)$.

The search can, however, also be carried out asynchronously. Kung [10] describes the algorithm in the two P.E case in detail and a simplified version of this is given below.

We begin by assuming that we have two points A and D, a continuous function f(x) and that $f(A)f(D)<0$. Let $|D-A|=L$ and let Θ be the positive real number defined by

$$\Theta^2 + \Theta - 1 = 0$$

(so that Θ is $\frac{1}{2}(\sqrt{5}-1) = 0.618...$)

In the asynchronous algorithm to be described we shall, at any instant, know that the zero lies in a certain interval and be awaiting the evaluation of f(x) at two points; when either of these evaluations is complete the process either terminates or a new evaluation is begun at a point which is determined by:

(i) which of the two evaluations was completed,
(ii) which of two "states" (defined below) the system was in during the evaluation,
(iii) the value of f(x) at the point of evaluation.

The two states are defined with reference to the interval in which the zero is known to lie and the two points at which the evaluation is being performed. As above, let the interval in which the zero is known to lie be $\langle A,D\rangle$ and be of length L. Define points B,C inside $\langle A,D\rangle$ by

$$B = A + \Theta^2 L, \quad C = A + \Theta L.$$

From the definition of Θ it follows that $|CD|=\Theta^2 L$ and $|BD| = \Theta L$; less obviously, $|BC|=\Theta^3 L$. The values of f(x) at the points A and D are, of course known.

In state $S_1(L)$: we are awaiting the evaluation of f(x) at the point B and at some point, E, outside $\langle A,D\rangle$.

In state $S_2(L)$: we are awaiting the evaluation of f(x) at the points B,C.

The evaluations at the two points are carried out asynchronously by the two P.E's. On completion of an evaluation a P.E either terminates the algorithm or updates 5 global variables: (i) the system state, (ii) the end-points of the interval in which the zero is now known to lie, (iii) the values of f(x) at these two end-points.

The appropriate action to be taken when an evaluation is complete and

$|f(x)|$ is not yet sufficiently small to terminate the algorithm can best be described by a table. For convenience we suppose that $f(A)<0$ and $f(D)>0$ and indicate the evaluation just completed by a + or - sign and the evaluation still in progress by a ? sign. By $S_1(I)$, $S_2(I)$ we indicate that the zero lies in an interval of length I.

Case	Present State	Sign of f(x) at points				New State	New Interval
	$S_1(L)$	E	A	B	D		
1		?	-	-	+	$S_1(\theta L)$	<B,D>
2		?	-	+	+	$S_1(\theta^2 L)$	<A,B>
3		-	-	?	+	$S_2(L)$	<A,D>
4		+	-	?	+	(Implies more than one zero : ignore)	
	$S_2(L)$	A	B	C	D		
5		-	-	?	+	$S_2(\theta L)$	<B,D>
6		-	+	?	+	$S_1(\theta^2 L)$	<A,B>
7		-	?	-	+	$S_1(\theta^2 L)$	<C,D>
8		-	?	+	+	$S_2(\theta L)$	<A,C>

Thus, in Case 6, for example: the evaluation at B is completed first and $f(B)>0$ so the zero is now known to lie in the interval <AB> of length $\theta^2 L$ ($\doteq 0.382L$); the value of f(C) is still awaited and since C lies outside <AB> we are now in state $S_1(\theta^2 L)$. The PE which evaluated f(B) now proceeds to evaluate f(x) at the point P in <AB> defined by

$$P = A+\theta^2|AB| = A+\theta^4 L$$

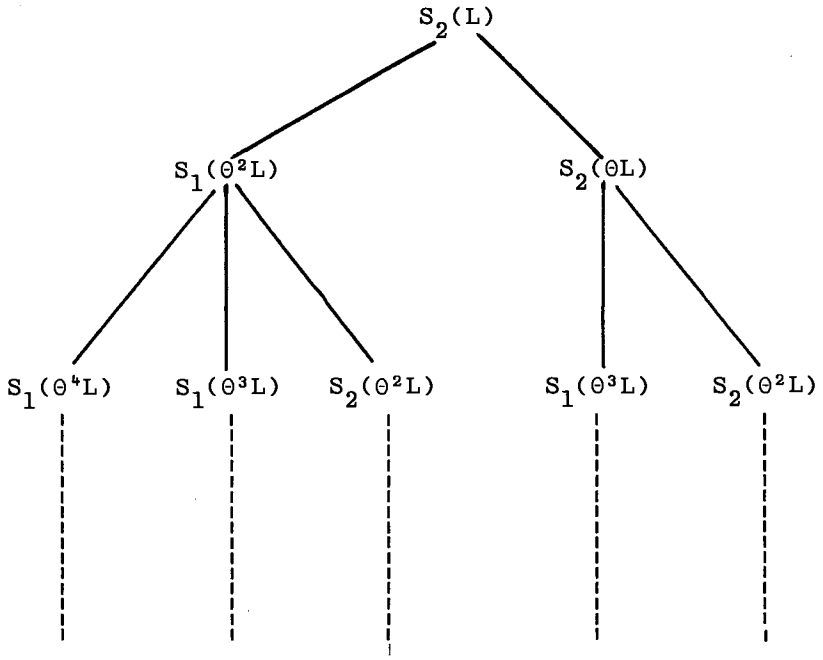
as required by state $S_1(\theta^2 L)$, and updates the five global variables accordingly. (We ignore the fact that the value of f(C) is irrelevant, in this version of the algorithm, in Case 6: more complex versions exploit this fact, see [10]).

The algorithm is certain to converge since the interval in which the zero is known to lie is reduced in length by a factor of θ (0.618...) in three of the seven cases, by a factor of θ^2(0.382...) in three of the remaining cases and is left unchanged in Case 3. Since, however, Case 3 represents a transition from $S_1(L)$ to $S_2(L)$ it cannot occur again immediately so that a reduction of interval length must follow after the next evaluation even in this case.

The speed at which the algorithm converges depends upon which particular path through the tree (which is derived from the table above and takes $S_2(L)$ as initial state) corresponds to the sequence of functional evaluations. Kung (op.cit) shows that the speed-up, compared to single P.E binary search lies in the interval

$$<2\log_2(\theta^{-1}),\ 4\log_2(\theta^{-1})> \text{ i.e. } <1.388,\ 2.777>$$

and that the asynchronised two P.E. algorithm is certainly faster than the



synchronised version if the overheads due to synchronization exceed 14% and could be as much as about 1.5 times faster even if the synchronization overheads are negligible.

This algorithm illustrates how complex the analysis of an asynchronous algorithm can be even for a two P.E system. As the number of P.E's increases so do the number of possible state-transitions and so too the complexity of the analysis also increases. Note, too, that even if all the P.E's are identical the computation times for f(x) at two points, x_1, and x_2 say, may be very different e.g. if f(x) is a slowly converging series, rather than a polynomial. On the other hand the synchronization overheads will also increase as the number of P.E's in a synchronized system is increased and it is not obvious what number of P.E's is optimal for the two types of algorithm, particularly if all the P.E's are identical (if they are not the synchronized algorithm may lose rather heavily, the overall speed being dependent on the slowest P.E).

4.7 Matrix multiplication by Systolic Arrays with Hexagonal Symmetry

Kung [12] has shown how an array based upon identical, very simple, P.E's can be used to multiply two two-dimensional matrices. The system is elegant and illustrates very nicely the systolic array concept.

The basic P.E can be described geometrically as a hexagon, with 3 input-faces and 3 output-faces as shown in Figure 3.

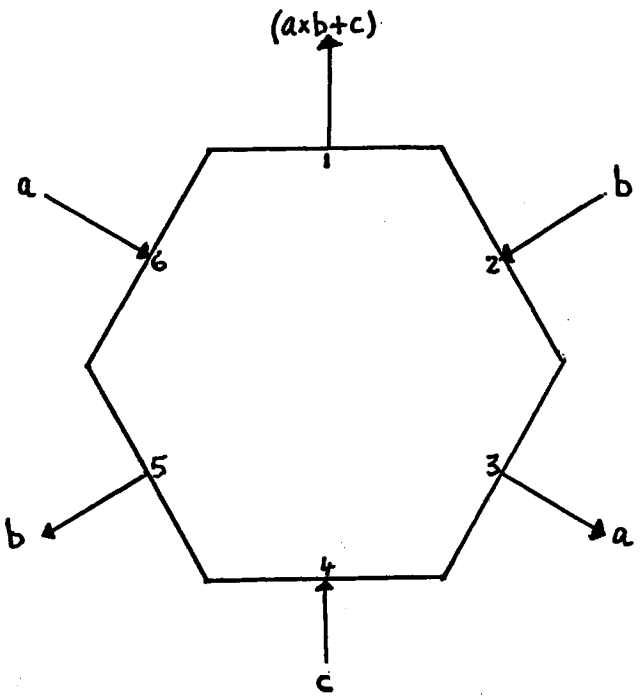


Figure 3.

The function of the P.E is very simple, viz: Inputs a, b and c arrive at faces 6, 2 and 4 and emerge as outputs a, b and (axb+c) at faces 3, 5 and 1 respectively one clock-pulse later.

Suppose now that we wish to multiply two 2x2 matrices

$$A = \begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix} \qquad B = \begin{pmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{pmatrix}$$

producing

$$C = \begin{pmatrix} c_{11} & c_{12} \\ c_{21} & c_{22} \end{pmatrix}$$

we may do this by building a system of 7 P.E's as shown in Figure 4.

The various data streams at time zero are shown in the figure. What happens at each subsequent clock-pulse is shown below:

Time	Activity
1	$C_{11} = a_{11}b_{11}$
2	$C_{11} = C_{11}+a_{12}b_{21}; C_{21}=a_{21}b_{11}; C_{12}=a_{11}b_{12}.$
3	$C_{21} = C_{21}+a_{22}b_{21}, C_{12}=C_{12}+a_{12}b_{22}; C_{22}=a_{21}b_{12}$

Thus with 7 P.E's we multiply two 2x2 matrices in 4 clock pulses. If the matrices are 3x3 we add another ring of 12 P.E's "outside" the outer 6 shown

in Figure 4. For nxn matrices we require

$$1 + 6 + 12 + .. + 6(n-1) = 3n^2-3n+1$$

P.E's and the computation is completed in O(n) clock pulses. On a uni-processor multiplication of two nxn matrices normally takes $O(n^3)$ operations (we assume that the matrices are "full" and ignore the possibility of using one of the ingenious, but complex, algorithms such as Strassen's [16].

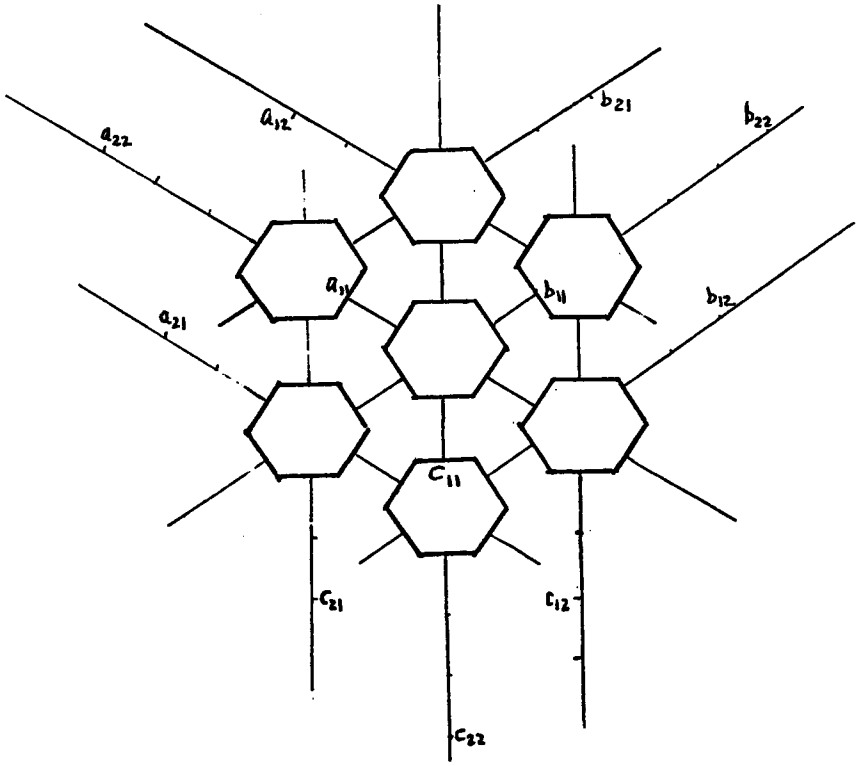


Figure 4.

Kung gives variations on this method for Band Matrices and applies the same system to the Computation of Discrete Fourier Transforms.

It is by no means inconceivable that applications exist where fast multiplication of two two-dimensional matrices of moderate order (eg n=20) is required in which case the building of a special systolic array based on less than $3n^2$ (1141 for n=20) identical simple P.E's of the type above might provide a satisfactory solution.

5. SOME IMPLEMENTATIONS OF PARALLEL ALGORITHMS ON MIMD SYSTEMS

The practical implementation of a parallel algorithm on an MIMD system is never easy and may be very difficult. Even when the system is homogeneous, i.e. when all the processors are identical, the standard operating system and compilers for a single processor will not be suitable for the multi-processor system so that until operating systems and compilers for languages which include parallelism become widely available, specially written programs at the machine-code level are likely to be required.

Despite these difficulties, however, MIMD systems have been constructed and a variety of algorithms written for them and tested to see how they perform in practice. The Department of Computer Science at Carnegie-Mellon University has played a major role in this respect and their reports,[e.g.13, 17] form a most interesting and valuable part of the research literature, which should be studied by anyone interested in the subject. Here we can only pick out a few examples.

In [17] Oleinick reports on the implementation and evaluation of parallel algorithms for (i) root-finding, and (ii) speech-recognition on the Carnegie-Mellon C.mmp system. The C.mmp system is described in detail in [18]. For our purposes it is sufficient to know that, at the time when [18] was written, C.mmp was a system of 16 processors that shared a common memory of 2.5 Megabytes. The 16 processors were completely asynchronous PDP-11s; 5 were PDP-11/20's and the remaining 11 were PDP-11/40's. Connection of the processors to the memory was via a 16 x 16 crosspoint switch, permitting up to 16 memory transactions simultaneously. I/O devices, unlike memory, were associated with specific processors so that use of a device by another processor required inter-processor communication. A general-purpose multiprogramming operating system, called Hydra [19], was written for the system; it was organized as a set of re-entrant procedures that could be executed by any of the processors; several processors could execute the same procedure simultaneously, this concurrency being accomplished by the use of *locks* around critical sections of the operating system.

The root-finding algorithm used by Oleinick was the 'natural' generalization of the binary-search algorithm when n P.E's are available, i.e. subdivision of the interval of uncertainty into (n+1) equal-length subintervals. This is not the optimal subdivision (see [17], for the general case, and section 4.6 above for n=2) but it is not far off and has the merit of simplicity. The problem solved can be stated:

"Given a random number h lying in <0,1> and a fixed small number ε find the value of x such that

$$|F(x)-h| = \left| \frac{1}{\sqrt{2\pi}} \int_x^{\infty} \exp(-\tfrac{1}{2}t^2)dt - h \right| \quad < \varepsilon"$$

Computation of the integral was done by means of the series

$$x + \frac{x^3}{3.5} + \frac{x^5}{3.5.7} + \frac{x^7}{3.5.7.9} + \ldots$$

for $x < 2.32$

and by means of the continued fraction

$$\frac{1}{x+} \quad \frac{1}{x+} \quad \frac{2}{x+} \quad \frac{3}{x+} \quad \ldots.$$

for $x > 2.32$.

Computation times varied for different values of x. At each stage the n processors computed the values of f(x) at the n division points. When all values were available the last processor to complete its computation tested for completion of the algorithm and either terminated or worked out the new sub-interval, "woke up" the "sleeping" processors and so activated the next stage. The algorithm is therefore of the synchronous type and the overall speed is bounded above by that of the slowest processor and subjected to the overhead of the synchronization and other delays.

Oleinick compared the actual speed-up achieved on n processors (n=2,3,...9) with the best-possible theoretical speed-up [$\log_2(n+1)$], found that the gap between expected and observed performance increased as n increased (e.g. 3 processors performed at about 80% of the expected level whereas 9 processors performed at about 70%) and analysed the causes of the degradation of performance. Seven factors were found:

(1) Synchronization: synchronization mechanisms covering a wide range of sophistication were tried, producing significantly different results as the granularity increased
(2) Calculation of F(x): the time required for this varied with x, up to a factor of 3.4
(3) Memory Contention: the average cycle length varied by a factor of up to 2.8
(4) Bottleneck of Scheduling processes in the Operating System: again a factor of up to 2.8 in delay time was observed
(5) Speed of the Processors: in the non-homogeneous C.mmp individual processor speeds varied by a factor of up to 1.6
(6) I/O Devices and Interrupts: could degrade computation of F(x) by a factor of up to 1.3
(7) Variations in memory speed: the 16 memory banks associated with the PE's were not identical (some were core, some semi-conductor), this caused a further degradation by a factor of up to 1.07.

Speech recognition systems offer the opportunity of exploiting parallelism but their complex control structures can impose a large synchronization overhead [20]. Oleinick [17] describes the use of the Carnegie-Mellon speech recognition system HARPY on C.mmp. HARPY can

recognise phrases and sentences from many speakers based on a finite vocabulary. A serial algorithm already existed and this was refined in various ways to produce four versions of a parallel algorithm, the fourth having some asynchronous sections. Using 8 processors speed-up was improved from 2.94 in the first version of the parallel algorithm to 4.29 in the fourth version. Significant improvements resulted from (i) balancing the work-load across all the processors, (ii) arranging for two sub-tasks to run asynchronously rather than, as hitherto, synchronously.

In another experiment C.mmp based on 4 processors recognised 15 trial utterances in 46 seconds, compared to 49 seconds on the (more powerful) uniprocessor DEC KL10. Using 7 processors C.mmp completed the recognition in 33 seconds. (On 1 processor C.mmp required 144 seconds).

This example illustrates both the effort that may be required to construct satisfactory parallel algorithms, particularly when the synchronization overheads are likely to be large, and the rewards that may be achieved.

5.2 PACS Hoshino and others at the University of Tsukuba have described a 32-element parallel microprocessor array for scientific calculations called PACS-32 [21] and have given details of its performance on a variety of problems including aerodynamics, Monte Carlo, nuclear reactor calculations and the solution of partial and ordinary differential equations.

PACS-32 is a MIMD system with both synchronous and asynchronous modes, built around 32 processing units, together with a control unit and a host computer as well as I/O devices and communication lines (see Figure 5).

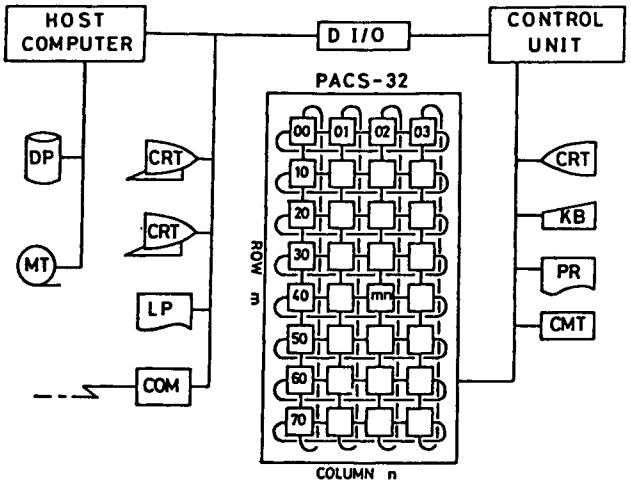


Figure 5. Configuration of PACS-32 system. (CRT = cathode ray tube, KB = keyboard, PR = printer, CMT = cassette magnetic tape, DI/O = digital input/output lines, PU = processing unit, LP = line printer, COM = communication interface, DP = disk pack memory, and MT = magnetic tape.) (From Hoshino et al. [21])

The 32 processing units (P.U) are represented geometrically as an 8 x 4 array, as shown in Figure 5 ; physically they are arranged in a 2 x 2 x 8 rectangular box. Each P.U, including those on the left-and right-hand sides, top and bottom, is connected to its 4 nearest neighbours (as indicated in Figure 5). Topologically therefore the 32 P.U's are arranged as a torus.

Each PU contains several elements, including a micro-processing unit (MPU), an arithmetic processing unit (APU), as well as local, result and communication memories. The MPU'S are Motorola MC6800 (8-bit) microprocessors and the APU's are Advanced Micro Device's Am9511's (16 and 32-bit arithmetic). (See Figure 6).

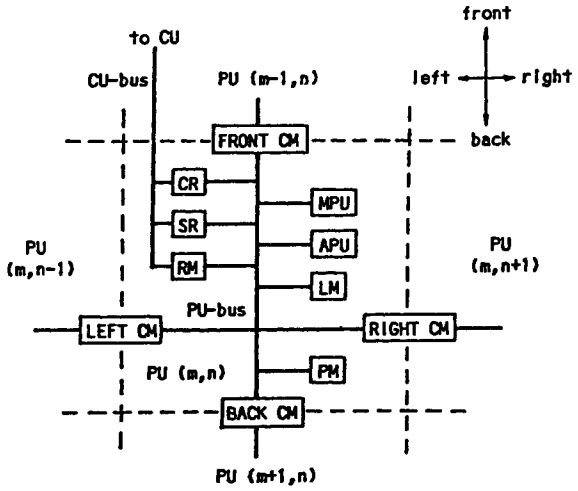


Figure 6. Configuration of processing unit (MPU = microprocessing unit, APU = arithmetic processing unit, LM = local memory, RM = result memory, FRONT CM, BACK CM, LEFT CM, and RIGHT CM = communication memories in front, back, left, and right, respectively, CR = control register, SR = status register, PU(m, n) = processing unit with coordinate (m,n), CU = control unit). (From Hoshino et al. [21])

Each PU has access to three banks of memory: Local Memory (5 kbytes) for local data and programs: Result Memory (1 kbyte) shared between the PU and Control Unit and used for communication between them; Communication Memory (2kbytes) shared between neighbouring PU's. In order to avoid access conflict in the Communication Memory all PU's are synchronized by the system clock; this clock signal originates in the control unit and is fed to all PU's but for the odd-numbered PU's it is fed 180^{o} out of phase and hence odd- and even-numbered PU's access the Communication Memories in

alternate half-cycles of the clock. Since the neighbours of PU(m,n) are PU(m,n±1) and PU(m,n±1) and since m+n determines whether a PU is "odd" or "even" the four neighbours of PU(m,n) are necessarily of opposite parity to PU(m,n) itself; hence conflicts in the Communication Memory are avoided.

The host computer is a Texas Instruments 990 Model 20. It is used to compile/assemble the source program, load the object code into the CU and PU, initiate parallel tasks, transfer and receive the data to and from the CU, and output the results.

Recognising that different applications would require different degrees of inter-processor communication and synchronization the designers of PACS-32 have provided facilities to allow considerable flexibility in these areas. Data transfer is available in two modes: *communication* and *non-communication;* in the latter case there is no data transfer among PU's; Monte-Carlo simulation for independent particles utilizes this mode. The *communication mode* is further subdivided into *synchronous* and *asynchronous* modes. In the *synchronous mode* all PU's are synchronized by the statement CALL SYNC. After the PU's have been synchronized, data are transferred to the Communication Memory, the transfers being organized to avoid memory contention. The *asynchronous mode* is itself subdivided into *conditional* and *unconditional* modes. In the conditional asynchronous mode data transfer occurs only if the data have been updated; in the unconditional asynchronous mode data are transferred regardless of whether they have been updated or not.

Programs for PACS-32 are usually written in FORTRAN for the serial part of the job and in SPLM, a high-level language for parallel-processing based on SIMPL [22], for the parallel parts of the job.

The performance of PACS-32 was evaluated over six major problems and the efficiency of the system, defined as the percentage of the time that the PU's were busy, worked out. The efficiency ranged from 78% to 99%. The author's final paragraph is worth quoting *verbatim:*

> "Computer scientists and engineers used to assume that overhead due to inter-processor communication would prohibit the practical application of a PACS-type processor array. Our experience has shown that this assumption can no longer be made."

6. PARALLEL LANGUAGES

None of the widely-used early programming languages, such as FORTRAN, ALGOL or COBOL, were designed to cope with parallel algorithms. With the introduction of multiprocessor systems however the possibility of parallel programming was raised and a number of proposals for handling concurrency, as in Concurrent Pascal [24], or parallelism (e.g. ALGOL 68 [25]) appeared. More recently ADA, which caters for parallelism by means of "Tasks" has been

introduced [26], but few people have so far had the opportunity of using it.

For the vector and array processors the manufacturers (CRAY, ICL etc.) have provided modified versions of FORTRAN which enable the programmers to exploit the particular features of these various machines, some of these versions of FORTRAN are very machine-specific, which is perhaps not surprising. Portability of programs from one such machine to another seems a long way off. For a good summary of the situation c. 1981 see [27].

For the MIMD systems the situation is even less satisfactory. Most of the systems built so far are essentially "one-off" and their creators have generally provided ad hoc solutions to the programming problem, often by means of additional features in FORTRAN. Fairly typical are those provided at Loughborough University in the UK for their NEPTUNE system [28]. NEPTUNE is a parallel system based upon four Texas Instruments 990/10 minis running under the DX10 Operating System. Pseudo-FORTRAN syntactic constructs have been provided to enable users to

(i) create and terminate parallel paths
(ii) state which data is shared between paths
(iii) ensure synchronization where required.

Thus for example the construct:

```
      $DOPAR 100 I = N1,N2,N3
       (code)
100   $PAREND
```

causes the generation of (N2-N1+1)/N3 paths each with a unique value of the loop index I. This is used for the generation and termination of paths with identical code.

On the other hand to generate paths with different code a construct of the form

```
      $FORK 101,102,103;150
101    (code 1)
      TO TO 150
102    (code 2)
      GO TO 150
103    (code 3)
150   $JOIN
```

is used.

Synchronization is provided by the user specifying a list of "resources" which may only be owned by one processor at any time. Such a resource is claimed and subsequently released by the construct

```
$ENTER (name 1)
$EXIT  (name 1)
```

If any other processor tries to claim this resource in the interim it is refused and the processor idles until the resource is available. It is clear that a deadlock situation could arise; the onus of preventing this is left to the programmer.

REFERENCES

(1) Menabrea, L.F, "Notions sur la machine analytique de M.Charles Babbage", Bibliotheque Universelle de Geneve, Serie 3, Tome 4, (1842), 352-376.

(2) Zacharov, V, "Parallelism and Array Processing", Proc. 1982 CERN School of Computing, 66-121.

(3) Flynn, M, "Some computer organisations and their effectiveness", I.E.E.E. Trans. Comp C-21,9 (1972), 948-960.

(4) Rodrigue, G (Ed), "Parallel Computations", Academic Press (1982).

(5) Evans, D.J. (Ed), "Parallel Processing Systems, An Advanced Course", Cambridge University Press (1982).

(6) Howlett, J. et al, "DAP in action", ICL Technical Journal, May 1983, 330-344.

(7) Jordan, T.L, "A guide to Parallel Computation and some Cray-1 Experience" in (4), 1-50.

(8) Barlow, R.H. and Evans, D.J, "Parallel Algorithms for the Iterative Solution to Linear Systems", Comp.J, 25 (1982), 56-60.

(9) Dijkstra, E.W. "Cooperating Sequential Processes" in "Programming Languages" (F.Gennys, Ed), Academic Press (1968), 43-112.

(10) Kung, H.T, "Synchronized and Asynchronous Algorithms for Multiprocessors" in "New Directions and Recent Results in Algorithms and Complexity" (J.F.Traub, Ed), Academic Press (1976), 153-200.

(11) Baudet, G.M, "Asynchronous Iterative Methods for Multiprocessors", J.A.C.M, 25 (1978), 226-244.

(12) Kung, H.T, "The Structure of Parallel Algorithms", Advances in Computers, 19 (1970), Academic Press, 65-112.

(13) Kung, H.T, "Why Systolic Architectures?" Carnegie-Mellon Dept. of Computer Science Report: CMU-CS-81-148 (1981).

(14) Wong, F.S. and Ito, M.R, "Parallel Sorting on a Re-circulating Systolic Sorter", Comp.J, 27 (1984), 260-269.

(15) Thompson, C.D and Kung, H.T, "Sorting on a Mesh-Connected Parallel Computer", CACM, 20 (1977), 263-271.

(16) Strassen, V, "Gaussian Elimination is not Optimal", Numerische Mathematik, 13 (1969), 354-356.

(17) Oleinick, P.N, "The Implementation and Evaluation of Parallel Aglorithms on C.mmp", Carnegie-Mellon University, Dept. of Computer Science Report, CMU-CS-77-151 (1978).

(18) Wulf, W.A and Bell, C.G, "C.mmp - A Multi-Mini Processor", AFIPS Proc. FJCC 1972, Vol. 41, 765-777.

(19) Wulf, W.A. et al, "HYDRA: The Kernel of a Multiprocessor Operating System", C.A.C.M. 17 (1974), 337-345.

(20) Lesser, V.R, "Parallel Processing in Speech Understanding Systems", Speech Recognition (1975), 481-499.

(21) Hoshino, T et al, "PACS: A parallel microprocessor array for scientific calculations", ACM Transactions on Computer Systems, 1, (1983),195-221.

(22) Basili, V.R and Turner, A.J, "SIMPL-T: A Structured Programming Language", Univ. of Maryland, Computer Science Center Report CN-14.1 (1975).

(23) Hoare, C.A.R., "Communicating Sequential Processes", C.A.C.M, 21 (1977), 666-677.

(24) Brinch Hansen, P: "The programming language Concurrent Pascal", IEEE Trans. Software Eng 1 (1975), 199-207.

(25) van Wijngaarden, A (Ed): "Report on the Algorithmic language ALGOL 68", Numer.Math 14 (1969), 79-218.

(26) Shumate, Ken; "Understanding ADA", Harper and Row (New York), 1984.

(27) Hockney, R.W. and Jesshope, C.R: "Parallel Computers", Adam Hilger Ltd. (Bristol), 1981.

(28) Barlow, R.H.et al: "The NEPTUNE parallel processing system", Dept. of Computer Studies, Loughborough University, UK (1981).