PARALLELISM AND ARRAY PROCESSING

V. Zacharov
CERN, Geneva, Switzerland

ABSTRACT

Modern computing, as well as the historical development of computing, has been dominated by sequential monoproducts. Yet there is the alternative of parallelism, where several processes may be in concurrent execution. This alternative is discussed in a series of lectures, in which the main developments involving parallelism are considered, both from the standpoint of computing systems and that of applications that can exploit such systems.

The lectures seek to discuss parallelism in a historical context, and to identify all the main aspects of concurrency in computation right up to the present time. Included will be consideration of the important question as to what use parallelism might be in the field of data processing.

1. INTRODUCTION

The history of developments in the automation of computational processes, of which electronic computing represents only a rather recent period, has always been characterized at any given stage by practical limitations. There have always been problems for which even the most powerful of computational systems have simply been inadequate; probably this will always be the case. Although there are many different factors which are responsible for the limitations which exist at a given time, the one which is typically the most important is speed. Indeed, it was precisely the inadequate speed of human beings, rather than any other factor, that led to so many attempts in the past to mechanize or to automate the computational processes that needed to be solved. This was certainly one of the main motivations behind the pioneering innovations in the nineteenth century of Charles Babbage, although the question of mistakes in computation was very important as well, and also the case in the developments of the early electronic and electro-mechanical stored-programme computers during the 1935–1945 period.

The position can be illustrated well by the kind of problem that is said to have led Konrad Zuse to his pioneering work in automatic computing. In the early thirties, the practical limit in computing stress distributions in static load-bearing structures was the solution of less than six simultaneous linear equations. Yet the design of a railway station roof in Germany about 1935 involved 30 equations, an almost impossible task even with teams of human beings aided by mechanical calculating machines. Certainly this was one of the categories of problem that was intended to be solved by the first machine designed by Zuse in 1937.

Thus the search for higher and higher computing speeds has proceeded ceaselessly: hand calculation has been supplanted by mechanical, mechanical by electro-mechanical, electro-mechanical by electronic, and the process continues. Even now, with electronic computers operating with cycle times as short as 10 nanoseconds, there are many attempts to develop circuits and processor components that operate still faster. Current developments to implement a computer based on the Josephson effect are not motivated by the wish just to utilize a new technology, but because the computational problems are there to be solved. A cryogenic, Josephson-junction computer could have characteristic circuit-times of about 1 nanosecond, including switching time and circuit delays, and so might operate 20 times faster than current uni-processing computers. There are also many other attempts to increase computational speeds by technological means.

Technology is not the only way, however, to achieve higher computing speeds. No matter how important is the influence of technology in implementing computer systems, there is always another option which can offer increased speed, even with a fixed operation speed for individual components. This option is parallelism.
Parallelism in computing systems is simply doing more than one thing at a time, and clearly there is no limit in principle to the number of concurrent actions. So, potentially at any rate, parallelism offers an arbitrary degree of improvement in computing speed. This is a most important point when considering limitations of computing systems, since, although conventional sequential processors can be speeded up just by increasing further the speeds of circuit components, we are approaching certain intrinsic physical limitations — such as the propagation speed of signals in physical media. There is a long way to go before we reach these limits, but they are there nonetheless. Even in the proposed Josephson computer, for example, we should remember that the signal propagation delays associated with the packaging and inter-coupling of a chip, in the order of 600 picoseconds, are already much greater than the on-chip switching times.

Another factor to take into account when considering limitations of computing systems is that of the system organization or architecture. This is a much less definite aspect to take into account than hardware parameters or performance, but it is clear that the purely sequential nature of most conventional computing systems imposes certain constraints not only on how problems are executed in such systems, but upon the associated algorithms and programs used. Although it may not be exploited in that way, parallelism at least offers the possibility of organizing both computing systems and problems for them in a way quite different from, and sometimes more effective than, that for sequential processing.

Parallelism is a term which implies that more than one operation of some sort is performed at each instant, and that the operations are of the same kind and proceed together in synchronism. Actually, this is not only not always the case in non-sequential computer systems, but the term “parallelism” is itself used to describe many different forms of system implementation and problem formulation. Perhaps a better word to describe what we shall be discussing here is concurrency, which certainly means performing more than one action at a given instant, but does not suggest any synchronism or complete overlap between any of these actions. We shall see later indeed that there is a wide variety of possibilities for concurrent systems.

Another question that arises when computational parallelism is discussed is that of array processing. Indeed, this expression is often used synonymously with parallel processing or with the term vector processing. It is true that very many concurrent computing systems have processors that are organized in regular arrays, or have the capability of processing regular arrays of operands with only a few instructions (sometimes one instruction only). Similarly, it is the case that many computational problems can best be expressed algorithmically in parallel form by operations on regular arrays of variables, not necessarily single dimensional in form. Nevertheless, this is not always the case, and many systems and problems exist where the parallelism is not of such a regular kind. In our discussion here, we shall try always to make a distinction between array processing and other forms of parallelism. Vector processing is the most common form of parallelism encountered in modern computing systems, simply because it is relatively easy to build a certain kind of processor allowing the concurrent processing of vector pairs with relatively little extra hardware compared with that required for sequential (non-concurrent) processing. The technique is called “pipelining”, and we shall discuss it later on, with some specific examples. However, not only is it not the only technique available, but, as we shall see, it may be anything but an effective solution for the wrong kind of problem.

We have defined the scope of this discussion as that of concurrency in computation. However, we have not mentioned what operations in a computational process are executed concurrently, and this is because there is no real limitation. We shall see that many different levels of parallelism may exist, and that a given parallel-processing system may implement more than one level, perhaps with concurrency in executing different parallel operations at different levels. And so complexity becomes an important issue, which we shall touch upon briefly. It is probably true that the most important factor limiting computational speed at the present time is not, as it has been so many times in the past, availability of appropriate computational hardware, but rather the difficulty of managing the complexity of algorithm formulation and of programming.
2. A HISTORICAL PERSPECTIVE

Computational parallelism is not a new idea and it seems to have occurred, in one form or another, to nearly all the pioneers of computing from the earliest times. For us, so many of whom are concerned with programmable microcircuit devices, it is important to recall the very many good ideas of parallel processing that could not be implemented in the past, or only implemented with poor effectiveness, because of the difficulties of hardware limitations. Although modern microprocessors are still relatively simple devices functionally, they offer a freedom for implementing system architecture that is unprecedented. And so, it is instructive to look at least at the principal advances in introducing parallelism in previous computer systems.

The idea of parallelism certainly appears to have been considered by Charles Babbage in his design of the Analytical Engine, one of the very first stored-program computers. For although no mention of concurrent arithmetic operations can be found in Babbage's own writings, nor in the accounts of the Lady Lovelace, the possibility is mentioned explicitly by Menabrea in 1842, following Babbage's visit to Italy and his discussions there about the Analytical Engine. In the Bibliothèque Universelle de Genève, one can read an account of this machine by Menabrea:

"D'ailleurs, lorsque l'on devra faire une longue série de calculs identiques, comme ceux qu'exige la formation de tables numériques, on pourra mettre en jeu la machine de manière à donner plusieurs résultats à la fois, ce qui abrégera de beaucoup l'ensemble des opérations."

And Babbage had also conceived the idea of parallelism in single arithmetic operations rather than sequential operations, even for his Difference Engine.

Rather later, in relation to punched-card machines, one of the earliest pioneers, Herman Hollerith, used parallelism in both tabulating and sorting machines, where several data items stored on a punched card (still used today in unchanged form in certain computer installations!) could be processed concurrently. Of course, the processing involved was only of a very elementary kind, yet it was a major innovation at the time, and one which made possible the rapid processing of the US censuses at the turn of the century.

By the time of the earliest electro-mechanical and electronic calculating engines, in the period 1937–1946, the notion of parallelism was well established. Zuse, whom we have already mentioned, gave in his diary of 1937 an explicit reference to the point that carrying out elementary operations in parallel is a way of achieving faster speeds of computation. The very first electronic, general-purpose calculator, the ENIAC only allowed the concurrent execution of different arithmetic operations, but also permitted these operations to proceed in parallel with input or output on punched-card equipment. This was made possible on the ENIAC by having several different operational units, in turn inter-coupled by several data highways. It is interesting to note that the ENIAC also had a separate bus for broadcasting commands to the different functional units, so enabling initiation of concurrent operations.

Another very important early (1945), general-purpose computer, the Bell Laboratories relay machine, had several areas of concurrency. For example, the reading of externally stored commands (on paper tape) could proceed concurrently with calculation, and the searching for operands in stored tables, also on paper tape, could be overlapped with processor operations. More significant in a historical context, however, is that there actually were two separate processors that could share input-output and stored tables. In this way, it was possible to use two processors concurrently to execute different tasks of the same problem. It is interesting to note that the Harvard Mark II machine also was a twin processor, which could be used to execute cooperative tasks concurrently.

Those responsible for the designs of the very first electronic computers with internally stored (and hence machine modifiable) programs, namely von Neumann and Turing, were well aware of the significance of parallelism both in performing elementary operations and in overlapping them with other operations such as input and output. However, the extent of concurrency in actual early machines was limited more by engineering considerations than by the aspirations of some of the designers. There was a reluctance to increase the number of electronic tubes in processors to the extent required for parallel arithmetic, but in any case many of the early memories, such as mercury delay lines, were essentially serial in character, and so better matched to bit-serial operations.

One very interesting feature of Turing's design for the ACE computer at the National Physical Laboratory (NPL) was that, although numbers were stored serially in delay lines in groups of 32, it was possible in a single instruction to add all the numbers in line A to corresponding ones in line B to give a vector sum in line C of the form $A + B \rightarrow C$. Moreover, by using a short line storing only a single operand $S$, the accumulated sum of all the 32 operands in line A could be implemented by a single instruction $A + S \rightarrow S$. By 1947 or so, codes had been developed at NPL with subroutines for a whole range of vector operations on arrays of operands, such as vector sum, inner product and others.
An ACE machine was not built until very much later than 1947 but, fortunately, another machine, the Pilot ACE (as it was called), was built in the interim with many of the important features of the Turing design, becoming fully operational in 1952. Like the ACE, the pilot machine had delay-line storage and arithmetic units of different operation times, and so it was realized very early that operands needed to be placed in the correct positions in the stores in relation to the instructions for which they were needed. Otherwise the instruction execution would need to be delayed until the corresponding operands became available. The technique of getting all the operands and instructions in the right relative positions was termed "optimum coding", and was well-known at the time (to the point of obsession) to all those who programmed any recirculating memories, such as delay lines or drums. Of course it is still known to those who deal with rotating stores, only they call it something else now!

To get the very best "optimum" code was rather difficult and, in the words of someone at the time, required a virtuoso programmer. Nevertheless, not only was such coding regularly produced, but we see one of the very first examples anywhere of real parallelism between different arithmetic operations. An excellent example of this (possibly the very first) appears in a paper by Wilkinson\(^7\) on the extraction of latent roots and vectors of matrices. In that paper, Wilkinson observes that "... the multiplier works in parallel with the remainder of the equipment, and many of the arithmetic facilities are connected with short storage units which are not used by the multiplier, so that a number of instructions may be carried out while multiplication is proceeding". He then goes on to give an example of calculating the \(i\)-th inner product involved in forming the \(i\)-th element of \(Ax\) in a way that would have brought joy to anyone who tried to solve similar problems on the CDC 7600. The example is as follows:

The cycle then proceeds as follows, once for each value of \(j\) from 1 to \(n\):

1. **A** Order the multiplication \(a_y \times x_j\).
   - While this is proceeding continue with the following:
     1. Build up the instruction for extracting the next element of \(A\), extract it, and store it in a short unit.
     2. Build up the instruction for extracting \(x_{j+1}\), extract it and store it (when \(j = n\) it extracts the number in the store next to \(x_n\), but as this is not used it does not matter).
     3. Add \(a_y\) to the accumulating grand sum of the elements of \(A\).
     4. Construct the corrections to the product \(a_y \times x_j\) (necessary because the Pilot ACE multiplier does not take account of signs).
     5. Add the correction to the product as produced by the multiplier which will have completed its operation by now.
     6. Add the resulting double-length product to the accumulating double-length scalar product.
   - Put new \(a_y\) in the position occupied by the previous \(a_y\) and the new \(x_j\) in the position occupied by the previous \(x_j\).
   - Is the present value of \(j\) equal to \(n\)?
     - If so go to position (B). If not form the new value of \(j\), i.e. \((j + 1)\), and return to (A).

2. **B** The \(i\)-th element of \(Ax\) is now complete; it is rounded off and put into its correct storage position.

What Wilkinson describes here is of course the kernel of the calculation, important because it is executed \(n^2\) times compared with \(n\) for the instructions outside the segment (A). And the optimum coding technique enabled a total execution time of only 4 milliseconds to be achieved, although multiplication alone took over 2 milliseconds, and access time to the instruction store was 1 millisecond. This enormous improvement in execution time was made possible by exploiting parallelism between functional units within the processor.

Another kind of concurrency, namely that between arithmetic processing and other computer activities, was also known and indeed exploited at about the same time. By 1951, the UNIVAC I allowed the overlapping of asynchronous I/O operations, memory accesses and central processor execution. And that great leap forward in 1953 by IBM into the domain of electronic computing, the 701, also allowed concurrency between instruction execution and transfers between I/O devices and memory. Since main storage was electrostatic, using cathode-ray tubes, and these needed to be refreshed, further parallelism was introduced with the capability to "steal" cycles from the processor while memory regeneration took place. Although only used for internal purposes, this is one of the very first examples of memory cycle-stealing that later became so common for direct memory access (DMA) in minicomputers.
By about 1956 the exploitation of computers for large computations had become extensive, and there was a very clear need for much higher speeds than hitherto possible. Part of this need was satisfied by the availability of new technology, in particular logic circuits based on the junction transistor, and memory based on ferrite cores. The then new technology led directly to improvements in performance of about an order of magnitude compared with machines existing prior to 1956. But this was not enough for many applications, and here again concurrency was exploited to achieve even faster computational speeds.

One of the ways of introducing further parallelism was to use several independent memory units instead of one only, and to have separate access to each. The idea was simply to fetch operands concurrently from all the separate memory units, so improving the total operand throughput potentially by the same factor as the number of units. Such a parallel memory organization first appeared in the STRETCH computer of IBM and the ILLIAC II. In the STRETCH computer in particular, the memory organization was highly parallel and arranged in an interleaved manner separately for instructions and for operands. Here, there were six memory units, interleaved so as to have their addresses distributed modulo 2 for instructions stored in the first two units, and modulo 4 for operands stored in the other 4. Since the memory had 2 microsecond cycle time and instructions required only half-words, an effective rate of 0.5 microsecond could be achieved both for operands and instructions under favourable circumstances.

The STRETCH computer was a very advanced machine for its time, the first one being delivered in about 1960 (although the project had started in 1956). What was so remarkable was that, although it used advanced technology to achieve a factor ten in performance compared with its predecessors, in particular the IBM 704, another factor of about ten was achieved by parallelism, of which interleaved memory was only one aspect. In STRETCH, all I/O operations could proceed in parallel with processor activity but, more significantly, many operations could execute concurrently within the processor. This was probably the first time such processor concurrency was included as a fundamental part of the design rather than as an incidental result there to be exploited (as in the earlier examples of optimum coding). And to obtain two orders of magnitude performance improvement within only one step in the evolution of computer systems was not only unprecedented since the move from mechanical to electronic machines, but it will probably never occur again!

To achieve such a performance improvement in the processor, the STRETCH computer allowed the fetching and the processing of instructions to proceed in parallel with fetching of operands and execution of instructions. And, since four instructions could be fetched simultaneously and there were several independent logical and arithmetic units, it was possible to have several (actually as many as six) instructions concurrently in various stages of execution. A very important feature indeed to enable processor concurrency to be achieved was the pre-accessing of both operands and further instructions ahead of the instructions being currently executed. This was the first example of such a "look-ahead" capability.

Although the development of computer architectures and hardware implementations has been, and still continues to be, one of evolution, the period until 1960 was, in some senses, a special one. As our brief review of that period shows, very many of the ideas—indeed most of them—which are now considered to be important in parallel computers, had not only been thought of, but had been incorporated into actual machines in one way or another.

What characterizes the period after 1960, in so far as we are considering concurrency, is that all the various methods for implementing parallelism, hitherto occurring in only a rather sporadic way, were brought together in certain machines and systematized. In these machines indeed, parallelism was made a central property of the machine architecture. At the same time, the attack on applications problems which were somehow parallel in nature also began to be systematized.

The decade of 1960–1970 was a transitional one for large-scale computation, marking the decline of conventional uniprocessing and the beginning of what are now termed vector or parallel processors. In this transition, two main lines of development emerged, the first involving a massive overlap of operations of the same kind in what are called "pipeline" units, the second using many separate processing elements (PE).
3. THE DEVELOPMENT OF PARALLEL MACHINES

As we have seen from the preceding section, the introduction of parallelism into computer implementation occurred almost everywhere where possible. And so there is no simple and single line of development leading to modern parallel processing machines. Before considering this development further, therefore, it is important to recognize that many levels exist where parallelism may be used. Let us summarize the main categories:

i) **Within functional units**: Arithmetic, logical and other operations can be implemented in bit-serial mode, parallel by bit groups, or on whole operands concurrently. Clearly there are limits to what can be done, and there is the question of cost and reliability of the extra hardware involved. However, this category does not generally affect the way in which a problem is formulated, although it does determine the speed of execution.

Another area of parallelism in this category is the concurrent access to several interleaved memory units.

ii) **Within processing elements**: At this level, the most obvious form of concurrency is between different operations executing in parallel on different operands. For this to be exploited, the problem formulation (the detailed machine coding in this case) needs to be appropriate to the PE organization.

Another kind of concurrency is where there is only a single instruction at one time, but the functional unit to which the instruction refers, for example a multiplier, may be able to process a stream of operands in an overlapped or pipelined fashion. Pipelining is a very powerful feature indeed, which we will discuss later. It can give significant improvements in execution speed even for simple sequences of entirely serial code.

iii) **Within uniprocessing computers**: Even with only a single processor, there are many activities that can proceed concurrently. An obvious example already mentioned is memory access, and another is I/O. In some ways, the concurrent operation of input-output is just a detail of system organization, since it is a way of keeping the operational memory full (or to empty it); if main memory were big enough, many of the I/O problems would vanish. However, when several uniprocessors need to inter-communicate in order concurrently to execute different tasks of the same job, then parallel I/O or inter-memory communication is a vital requirement — and a difficult problem to solve.

iv) **Many-processor systems**: An obvious category of concurrency is where a computer system contains several (often many) processors, either sharing main memory or not, and inter-communicating by some means. In the most usual example, the separate processors are used to execute quite independent jobs. This is the well-known multi-processor architecture, and the concurrency here is used not to speed up the execution of individual jobs, but the global throughput of the whole system.

Much more interesting from the standpoint of our discussion here is where the different processors are used either to execute separate cooperating tasks within a given job or where even separate code sequences within a given program are executed concurrently.

There is a very great difference here between computers where all the processing elements are identical, and operate in lock step, and where they differ.

The way in which all the different levels of concurrency are exploited differs from one category to another. In the case of functional parallelism, it is necessary to structure the machine code appropriately, and so the characteristics of any compiler used are of paramount importance. At the other extreme, in multi-processing systems, it is the operating system which does the job of allocating resources among concurrently executing jobs and between processing and I/O. In all the other categories, of central importance is the structure of the job itself and so the choice of algorithms used becomes the significant issue. We shall return to this most important question later.

At the present time, several machines exist whose architecture and implementation have been expressly chosen to provide massive parallel-processing capability. We shall describe some of them later. However, in the previous two decades, a number of earlier computer systems were developed where concurrency was a central feature in implementation, and where nearly all the basic ideas on which modern machines are based were tried out. In these machines, it is not always easy to separate out any one feature as being most significant, since several levels of concurrency were often implemented in the same system. Nevertheless, there were perhaps three separate lines of development whose evolution has been more or less distinct, namely pipelining, the use of many functional units within a uniprocessor, and systems of many cooperating processors. And there is one other category which might be mentioned separately, that of special-purpose computing systems. Let us briefly consider these four categories.

### 3.1 Pipelining

In many computational processes, the total process can be made to take place in a number of discrete steps or segments. For example, the processing of an instruction might be segmented into the separate phases of fetching the instruction, decoding, calculating the operand addresses, fetching the operands and executing the instruction. In
Fig. 1 the difference is shown diagramatically between some unsegmented process P and the same process separated into four sequential segments. There is no reason why the total time T for an unsegmented process P should differ from the sum of the separate segment times \( t_1 + t_2 + t_3 + t_4 \). The idea of pipelining is simply that if all the segments S are implemented by physically separate sub-units, then they can operate together, and several processes may proceed concurrently in an overlapped fashion. Fig. 1(b) represents thus a processing pipe in which there may be up to four concurrent processes P at any given instant.

(a)

(b)

---

The advantage of a pipeline can be seen from Fig. 2 which shows several processes P in concurrent execution. In the first clock period only the first process is executing, but by clock period 5 there are four processes in the pipeline and the first is complete (assuming the same time for all segments). Clearly the maximum rate of issue of complete processes from a full pipeline is one for every clock period, a potential improvement of \( m \) compared with unsegmented processing, where \( m \) is the number of segments.

---

<table>
<thead>
<tr>
<th>SEGMENT</th>
<th>( S_4 )</th>
<th>( S_3 )</th>
<th>( S_2 )</th>
<th>( S_1 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>sub-process</td>
<td>( P_1 ) ( P_2 ) ( P_3 ) ( P_4 ) ( P_5 ) ( \ldots ) ( \ldots )</td>
<td>( P_1 ) ( P_2 ) ( P_3 ) ( P_4 ) ( P_5 ) ( \ldots ) ( \ldots )</td>
<td>( P_1 ) ( P_2 ) ( P_3 ) ( P_4 ) ( P_5 ) ( \ldots ) ( \ldots )</td>
<td>( P_1 ) ( P_2 ) ( P_3 ) ( P_4 ) ( P_5 ) ( \ldots ) ( \ldots )</td>
</tr>
</tbody>
</table>

Clock period: 1 2 3 4 5 6 7 8

---

Fig. 2 Timing diagram for a 4-segment pipeline
The example above is that of pipelining in instruction processing, and one of the earliest implementations of this idea (in rudimentary form) was in the STRETCH computer. But pipelining can also be used with even greater effectiveness in arithmetic processes. For example, a floating-point addition may be segmented into, say, four execution phases: compare exponents, normalize mantissa, add, normalize result. There is then in principle no limit to the number of operands that can be processed in an overlapped way at the maximum pipeline rate, an encouraging prospect, bringing to mind the alliterative words of Alexander Pope:

"And the smooth stream in smoother numbers flows"

*An Essay on Criticism*

A machine which used pipelining in this fashion for instruction processing was the Ferranti ATLAS computer developed jointly with Manchester University and first delivered in 1963. In this machine, the floating-point addition time of 6 microseconds for a single pair of operands, could be reduced to 1.6 microseconds on average for a long stream.

Pipelining has been employed at more than one level in machines, and in different configurations. In the preceding examples the method has been used either to overlap the processing of successive instructions (in the I-unit) or the execution of successive operations of the same kind on different operands (in an E-unit). Of course, one can have both, and this was probably done for the first time in the IBM 360/91 in the mid-sixties. However, a very interesting application of pipelining of a somewhat different kind, also used in the 360/91, is at the micro-system level, for example within a single arithmetic operation. Here, for example in floating-point multiplication or division, the function can be implemented as a successive or iterative execution of a series of microfunctions; in the case of multiplication, the microfunctions might be additions (or subtractions) of partial summands deriving from using one or more bits of the multiplier. Then the same hardware segments are used cyclically to process different parts of operands; but only one instruction is being executed at any given time. So if, for example in 16-bit multiplications, 4-bit groups are being processed, one might require four cycles through a pipeline of 5 segments to generate the final product. Such a scheme is shown in Fig. 3, where \( S \) decodes each 4-bit group and provides 3 appropriately scaled multiplicands to be summed (with proper signs) in a carry-save adder CSA-1. In further cycles partial sums of accumulated operands are added in CSA-2 and CSA-3, but at the last cycle (i.e. after 8 minor cycles) the final outputs of CSA-3 are summed in a carry-propagating adder \( A \) to give a 32-bit product at the output.

![Fig. 3 An iterative pipeline multiplier](image-url)
We should remark here that the iterative scheme of Fig. 3 for pipeline multiplication is a development of an early proposal by Wallace, to use carry-save adders interconnected in a binary tree to perform all additions in successive levels until only two numbers remain; these are then added finally in a carry-propagating adder. The essential feature which enables pipelining to proceed here is that the output of each level should not change during the period of each minor cycle for which this output is needed as an input to the successive level. This feature is provided by "latching" the output of each CSA, a technique first introduced by Earle.

A further remark to make is that iterative pipelining within a single arithmetic function is not restricted to the simple operations of addition and multiplication. There is an elegant extension of the technique to the evaluation of more complicated functions, such as exponential, logarithm, ratio and square root. This is based on the iterative co-transformations of a number pair \((x,y)\) such that some chosen bivariate function \(F(x,y)\) remains invariant; in this process \(x\) is directed towards \(x_n\), where \(y_n\) is the required result. The pipelining technique has been employed here to implement a functional unit of two segments, one for successive evaluation of \(x\), the other for \(y\), the process continuing iteratively until convergence (usually when \(x_n = 1\)). For example, the 360/91 used such a scheme for division.

So far in our discussion of pipelining, we have considered machines where pipelining has been used to speed up arithmetic operations on sequences of operands not necessarily associated with one another. Moreover, the pipeline units themselves have been auxiliary to non-pipelined functional units, they have been fixed in the operations they can perform and also, generally, of fixed execution time. They are termed static uni-functional pipelines.

More recently, actually since about 1970, a number of machines have appeared where pipelining is a central rather than auxiliary feature of the architecture, and where sometimes the pipelines can be used to implement a range of different functions. Perhaps the best example of such an early implementation for general purposes is the STAR-100 computer, manufactured by CDC and first delivered in 1974.

### 3.1.1 The STAR-100 Machine

We shall not discuss here the whole machine, but only the pipelining features and those aspects necessary to support high processing throughput through the pipes. There are three important aspects: the pipeline functional units themselves, the streaming of operands from memory, and the question of processing arrays.

The basic organization of the STAR-100 processor is shown in Fig. 4, but omitting certain details. There are actually four main units: memory and storage-access control (SAC), two floating-point pipelines, a string unit, a stream control unit. The memory has up to 64 kwords of 512 bits, with a cycle time of 1.28 microseconds, and there is a paging mechanism. The stream unit controls the whole system, but also performs simple logical operations and arithmetic on scalar operands. It also contains a set of 256 high-speed registers, termed Register Files, to and from which most operations communicate, and a stack for 64 instructions which allows look-ahead for up to 16 instructions. Within the stream unit are shift and alignment networks for input and output operands, allowing the selection and manipulation of operands and their necessary routing. As the name implies, the Stream Unit has as a major function the task of streaming operands (and instructions) in a continuous flow between memory and the pipelines. Much of this function is performed by microcode stored in a special high-speed memory unit which can be loaded from main memory or from a separate maintenance station.

Apart from certain register to register operations on scalar operands, all the processing in the STAR-100 is performed in one of three pipes, all of which can be in operation concurrently. It should be said also that the other major units of STAR-100, namely memory and Stream Control also function concurrently.

The String Unit is used mainly to process decimal or binary strings. However it also plays a most important role in processing what are known as control vectors, which define which operands are to be processed or which results are to be stored in arrays of operations performed by the two floating-point pipelines.

The two arithmetic pipeline units are shown in Fig. 5 and Fig. 6. Both of them receive input from two streams of operands \(A\) and \(B\), and the results appear in a stream at \(C\). To save time taken in routing results through the Register File, which may be needed as input operands at \(A\) or \(B\), there is an optional data path (termed a short stop) enabling \(C\) to be coupled directly back to the input. A basic component of both pipelines is a five-segment floating-point adder pipeline, with the option of post-normalization or not. A feature of this adder is that it can operate on one or two streams of 64-bit operands or, optionally, on up to four streams of 32-bit operands, where each stream processing rate can be attained. Since the minor-cycle time in STAR-100 is 40 nanoseconds, processing rates of 50 million operations can be obtained for 32-bit operands, and since also both pipelines can function concurrently, there is a potential maximum throughput of 100 million floating-point operations per second (100 MFLOPS)— hence STAR-100.
MEMORY
512-bit words 1-28 microsecond cycle

ADDRESS DATA
WRITE BUFFER 16 WORDS
READ BUFFER 16 WORDS

VIRTUAL ADDRESS TRANSLATION 16 REGISTERS

DATA

INPUT / OUTPUT 64 WORDS

1/O CHANNELS

ADDRESS

ALIGNMENT NETWORK

ALIGNMENT NETWORK

INSTRUCTION STACK 32 WORDS

REGISTER LOGICAL AND SHIFT UNIT

C

B

A

REGISTER FILE 256 WORDS

INTERRUPT COUNTERS

LOAD/STORE UNIT

MICROCODE

PIPEGINE 1
+
-
X

PIPEGINE 2
+
-
÷
MULTI-PURPOSE

STRING UNIT

Fig. 4 The STAR-100 Processor
Fig. 5 STAR-100 Floating pipeline Unit 1

Fig. 6 STAR-100 Floating pipeline Unit 2
The two pipeline units differ in certain respects. The first unit also performs high-speed multiplication, while the second has additional units performing scalar division (by a single-segment iterative scheme) and multi-purpose functions respectively. The multi-purpose unit is pipelined in 24 segments and, depending upon control, can perform such functions as square root, vector division and vector multiplication.

As can be seen from Fig. 5, pipeline multiplication is performed separately on two pairs of 32-bit operands, so that there can be two independent streams concurrently in a single unit, or a concurrent processing of parts of a 64-bit multiply. To perform a 64-bit multiplication, the operands A and B are partitioned respectively into \((A_1 \cdot 2^W)\) and \((B_1 \cdot 2^W)\) where \(W\) is the width of the multiplier. The product \(A_1 \cdot B_1 + (A_1 \cdot B_2 + A_2 \cdot B_1)2^W + (A_2 B_2)2^W\) is performed in two cycles: \(A_1 \cdot B_1\) and \(A_1 \cdot B_2\) during the first, \(A_2 \cdot B_1\) and \(A_2 \cdot B_2\) during the second. The partial sums and partial carries are merged in the 64-bit merge unit into final sums and carries which are added finally to produce the required product.

The basic 32-bit multiplier is shown in Fig. 7, which represents a multi-segment pipeline of carry-save adders with intermediate latches. In the floating-point format of STAR-100, 48 bits are used as mantissa in 64-bit operands and 24 bits for 32-bit words. The multiplier operands are decoded into 12 groups of two bits and, depending upon the combinations of bits in these groups, so twelve appropriate multiples of the multiplicand are added (or subtracted) at the first level of carry-save adders. Successive adders then reduce the number of partial addends until only two remain, which are added (or merged and added) outside the multiplier. Intermediate latches are shown where they are necessary to staticize the intermediate results generated at different times in relation to the clock period.

Fig. 7 The STAR-100 32-bit multiplication pipeline
We note that the scheme of Fig. 7 is just a development of the original proposal by Wallace\textsuperscript{10}, but here for two-bit groups of the multiplier operand. The STAR-100 hardware implementation was with integrated circuitry based on emitter-coupled logic, but it is interesting to note that it is now possible to find VLSI multipliers for 16-bit operands based on the "Wallace tree" (of course, still rather slower than the STAR-100 multiplier!).

As has been mentioned, the peak rate of operand processing is 100 MFlops, implying that, no matter what intermediate buffering or ordering of operands is provided, it is necessary to provide for a correspondingly high aggregate rate of operands from main memory (and storage of results). Indeed, the peak rate is eight 32-bit or four 64-bit operands, concurrent with a stream of two or four results every 40 nanoseconds and with a stream of associated control elements at the same rate.

To provide for the very high memory bandwidth, two features of concurrency were introduced in the STAR-100 memory. The first is a very wide access path, and the second is a highly interleaved memory. The memory is organized into 32 banks with an access path of 512 bits. Thus, in the cycle time of 1.28 microseconds, a "superword" of 512 bits can be transferred to or from memory into each of the 32 banks concurrently, an aggregate rate of 1600 megabytes per second. In the minor cycle time of 40 nanoseconds, the 512-bit superword is indeed adequate to provide the required rate of 4 X 64-bit input operands plus 2 X 64-bit results and some 128 bits for the control vectors and whatever general input-output there might be to memory. Each 40 nanoseconds, successive banks of memory are accessed. The storage-access control unit takes care of the mapping of 512-bit superwords onto sets of narrower operands, which may be 64-bit words, half-words, bytes or even bits.

Clearly, in order to make possible such sustained and high computational rates, it is essential to organize storage of successive operands in successively interleaved memory banks. In STAR-100, all operands of vector instructions must be stored so that consecutive operands are located strictly sequentially in successive banks; the stride here is unity.

Although the memory access path is 512 bits wide, the operand paths to the Stream Unit are 128 bits. There may be four such paths concurrently active, three for reading and one writing, so matching the peak requirement in streaming operations of 4 X 64-bit input operands, 2 results and 128 bits for other requirements. In this case, since there is only a single or a few vector instructions for the whole stream of operands, there is also no problem of streaming instructions.

Since vector operands need to be in contiguous locations in interleaved memory, a problem arises about how to process either regular vectors of other than stride unity or vectors whose elements are scattered quite irregularly in memory. This problem can be tackled by the association of a control vector with certain operands in vector instructions. Figure 8 shows an example of this for multiplication of two operand vectors A and B, with the results in vector C, controlled by a vector V. V is simply a bit string, and results are stored only in the positions where V has a logical 1. Thus, for positions in the results vector C where V has logical 0, the corresponding positions will contain whatever operands were there previously to execution of the vector multiplication instruction; these positions are indicated by asterisks in Fig. 8. In this case it is important to stress that the unwanted results, for example A_2 B_2, A_5 B_6 and A_8 B_7 are actually computed, but they are not stored in corresponding positions in the result vector. Here, the introduction of a control vector and the wasted computation is an effective process in most cases where there are long vectors, simply because less time is lost in this case than by fetching a set of shorter vectors. This is because there is a long "start-up" time for the first result to appear in any pipeline operation, independent of the vector length.

\[ \begin{array}{cccccccc}
A & A_1 & A_2 & A_3 & A_4 & A_5 & A_6 & A_7 & A_8 \\
B & B_1 & B_2 & B_3 & B_4 & B_5 & B_6 & B_7 & B_8 \\
V & 1 & 0 & 1 & 1 & 1 & 0 & 1 & 0 \\
C & A_1, B_1 & * & A_2, B_2 & A_3, B_3 & A_4, B_4 & * & A_5, B_5 & * \\
\end{array} \]

Fig. 8 A controlled vector multiply in STAR-100
For the case of very irregularly distributed vector elements, or in the case of sparse vectors, it may be better (or
sometimes even necessary) to collect all the required elements in densely-packed short vectors, which may then be
processed at maximum speed by streaming through the pipelines. In STAR-100 there are two ways in which this can
be achieved. In a compress operation, a source vector $A$ (with elements $A_1, A_2, ..., A_n$ in contiguous locations) can be
compressed into a shorter target vector $B$, depending upon a bit vector $BIT; BIT$ has $n$ elements, and only those
elements $A_i$ are stored in $B$ where $BIT_i = 1$. The STAR-100 instruction is of the form

$$B = Q8 \text{VCMPRS}(A, BIT; B)$$

which is one of a whole class of macro-commands which can be embedded in STAR-100-FORTRAN code. There is a
corresponding $Q8$ instruction for expand of compressed vectors.

An even more powerful command gives STAR-100 the capability of scattering or gathering vectors to an index
list. Here, if $IL$ is an integer data vector, each element of which is the relative address of the positions of the
successive elements of a sparse vector, then the elements of a source vector $A$ can be gathered into successive and
contiguous positions of some vector $B$ by the command

$$B = Q8 \text{VGATHR}(A, IL; B)$$

This is the equivalent of a FORTRAN loop:

$$DO \ J = 1,N$$
$$B(J) = A(IL(J))$$
$$\text{CONTINUE}$$

The corresponding scatter operation is $Q8 \text{VSCATR}$, which accomplishes:

$$DO \ J = 1,N$$
$$A(IL(J)) = B(J)$$
$$\text{CONTINUE}$$

It is not our purpose here to discuss the overall performance of STAR-100, but only to consider the pipelining
aspects. And there can be no doubt that the STAR-100 machine had the most advanced and effective pipelining
processor of its period, with a high degree of concurrency in associated units. Nevertheless two factors need to be
mentioned in assessing STAR-100 performance, the first concerning the time for operations on scalars or short
vectors, the second concerning operand access.

Although the peak processing rate in STAR-100 pipelines could approach two results each minor cycle of
40 nanoseconds, nevertheless the time taken for the first result to emerge could be as much as 100 times
longer. Thus, for computations on scalars or on sets of short vectors, STAR-100 was actually a rather slow
machine. Moreover, because of the relatively slow access time to main memory for a randomly placed operand (1.28
microsecond), a memory reference could be rather slow, as for example in certain cases of gathering or scattering of
unfortunately distributed elements of vectors. It is only on intensive computations involving well-organised long
vectors that the STAR-100 offered distinct advantages over other contemporary processors.

As a final remark concerning pipelining, we observe that a single instruction specifying the starting address and
lengths of all the operands can suffice to process a very long stream indeed in a sequence of arithmetic
operations. Thus the usual measure of performance in conventional machines, namely MIPS (million instructions
per second) is totally inadequate. Although it by no means represents the whole picture, a much better measure here
would be the MFLOP. We also remark that STAR-100 was explicitly built for processing vector streams very fast
and with a very compact set of vector commands. It is a very good example of an array processor, even though it
does not use an array of processors.

3.2 Multiple functional units

Quite distinct from the concurrency of pipelining, we can achieve higher computing speeds by multiple functional
units within a single processor. In pipelining, we saw that the increase in speed was due to the partial overlap of
successive operations of the same kind, either by issuing several similar consecutive instructions or by a single "vector" instruction specifying the range of operands to be streamed through the pipeline. In the case of multiple functional units, each unit has to be activated by a separate instruction, specifying the corresponding operands, but now not only may the separate functions be entirely different from one another but they may overlap completely in execution. It is also not excluded that several functional units may simply be repeated, for example having two adders instead of one.

Of course, with multiple functional units there is a very important problem to solve, namely that of synchronism between units, a problem which, by and large, does not exist in pipeline processors. Synchronisation in pipelines is achieved effectively by the data structure of operands. That is an over-simplification, but once the operands are in correct order and a pipeline is activated, the relative order of results and their time sequence is assured (in the absence of interrupts). In multiple-function processors, different functions may take a different time not only to execute their function, but to obtain the necessary operands. And since some of the input operands to one function may be the results from others, we can see that the problem of synchronism is a much more complex one to solve. So the whole problem of effectively utilizing the parallelism of multiple functional units reduces to that of handling the complexity of the associated programming; it is not an easy task to solve.

3.2.1 The CDC 6600 and 7600

We have already mentioned machines which could execute more than one processor instruction concurrently, for example the STRETCH. This was also a feature of the IBM 360/91 machine, a very powerful processor indeed for its time. The first time, however, where multiple functional units were used as the central architectural feature of a uniprocessing machine was the CDC 6600\(^{13}\), which appeared in late 1964.

The 6600 was a machine that incorporated parallelism at many levels. At one of these, to handle input-output, there were 10 separate I/O processors to support the aggregate data flow between 12 external data channels and central memory. The memory itself was heavily interleaved in 32 banks of 4096 words, allowing consecutive accesses every 100 nanoseconds. There was also an instruction stack of 32 registers which could be filled concurrently with execution. Most important to our discussion of parallelism, however, was that the central processor had multiple functional units, although none of them was pipelined.

The 6600 processor centred around 24 operating registers to which 10 independent functional units were coupled. The functional units all performed register to register instructions, main memory being accessed only through the operating registers. The units were for shift, floating add, fixed add, divide, logical operations, branch, multiply and increment, these last two units being replicated. To handle the problem of synchronism and resource scheduling there was a "scoreboard" to ensure that any register reserved for use by a functional unit that had not completed its execution could not be filled by another operand, for example from central memory.

In executing a code segment of the form \( R = (A \cdot B) + (C/D) \), the 6600 could perform the fetching of C and D operands concurrently with the multiplication of \( A \cdot B \), whose operands had been fetched previously in consecutive cycles (since there were two increment units capable of fetching). The final add could not of course proceed until division was complete. In this way the execution time for this code, compared with what it might be in a strictly sequential processor, could be about halved as a result of parallelism.

In a successor to the 6600, the implementation of a still more powerful machine, the CDC 7600\(^{14}\), first delivered in 1969, used both parallel functional units and pipelining to achieve very much higher speeds. In architectural terms, the 7600 is very similar to the 6600, although implemented with much faster circuitry. The minor-cycle time was reduced from 100 to 27.5 nanoseconds.

The principal innovation in the 7600 was a very fast memory module, termed small core memory (SCM), of up to 64 kwords with 275 nanoseconds cycle time. SCM was interposed between a much larger main memory (LCM) and the operational registers of the processor unit of which, as in the 6600, there were three sets of 8, termed the X, A and B registers. Although SCM was interleaved 16 ways, successive accesses could occur only every minor cycle.

In the 7600, there are nine functional units for all the arithmetic operations on integers and floating operands, as well as logical and shift functions, increment, population count and normalize. All these units can execute concurrently with each other, provided of course that operands are available and registers free to receive the corresponding results. The units execute significantly faster than on the 6600, but a major difference is that they are all pipelined. It is thus possible, in principle at least, to process successive instructions in each unit separated by a single minor cycle. To enable this to be done, there is a stack of 12 registers which can hold up to 48 instructions. The instruction stack is a most important feature of the processor, for if some important inner loop
Fig. 9 An instruction trace for product of matrix by vector
kernel of code can be contained entirely within the stack, then execution can proceed at a rate governed by the access
time to the registers (27.5 nanoseconds) instead of by the time to fetch from SCM (about 300 nanoseconds). So, in
principle, if no other limitations or conflicts occur, a peak processing rate of about 35 Mips could be achieved on the
7600. In practice however, owing to the necessity for branching and for other reasons, such a very high rate can
rarely be obtained. Nevertheless, by very careful hand coding in the 7600 assembly language, and by choosing a
problem which lends itself to optimization of function unit utilization, processing rates close to this theoretical
maximum have been achieved. For example, in the CERN subroutine library there is a package\textsuperscript{15} for the
multiplication of a real matrix of three rows with a real column vector. The inner loop of this program accumulates two
terms of all the three scalar products for each cycle, which takes only 32 minor cycles. Figure 9 shows a trace of the
instructions in execution during each clock period for this inner loop, as well as the utilization of all the X, A and B
registers. The loop executes between clock periods 265 and 296, and it can be seen that there is no clock period
without concurrent action in the arithmetic registers X, and only four clock periods have no new instruction
issued. Thus 28 instructions are executed in 32 minor cycles, a rate of about 30 Mips.

Of course the example of Fig. 9 is difficult to achieve and, in order to store all the necessary commands in the
instruction stack and not to have too many operands for the available X registers, it is necessary to limit the order of
matrix to three. Matrixes of greater dimension need to be processed in bands. As a measure of the effectiveness of
parallelism here, it can be remarked that the strictly sequential execution of the code shown would take 140 minor
cycles, a factor of about four longer.

3.3 The programming problem

We shall discuss the question of applications programming later, both from the algorithmic standpoint and the
coding. But it is relevant at this stage to mention just one software development in exploiting parallelism, since it
touches on two of the machines that we have just mentioned, the 7600 and the STAR-100.

We have considered, so far, mainly the architectural features of machines which possess parallelism in some
form or another, and we have seen, potentially at any rate, that concurrency offers the possibility of a significant
increase in computing speed. With the advent of the first machines of this kind for general use, particularly in the
case of the 7600, there was the realization that, although the hardware allowed concurrency in operations, the
associated task of programming was very complex, and something would have to be done to reduce this complexity for
the ordinary programmer. In the example cited previously, for products of vectors with matrixes on the 7600, we saw
that the complexity of coding necessary to issue a new instruction almost every cycle was very great, and not
something that could be produced by the typical programmer. How then to exploit the hardware concurrency, while
shielding the programmer from the complexity of coding? Possibly the earliest attempt to solve this problem was the
utilization of libraries of subroutines, particularly the development in 1971 of STACKLIBE\textsuperscript{16} (spelt mainly without an
E, even in the original report).

The essential idea of Stacklib was that certain operations on vectors (and matrixes) are frequently encountered
and that, for these particular operations, very efficient code could be written once and for all in assembly
language. These operations, of which there are about 300, were invoked by special subroutine calls which could be
embedded in arbitrary FORTRAN code, and they exploited the parallelism of the 7600 with very high efficiency
indeed. For many of these operations, on long vectors, instruction rates of near the peak speed of 35 Mips can be
achieved, with a correspondingly high rate of execution (approaching 13 Mflops in certain cases).

What made the Stacklib routines so efficient, apart from the coding itself, was the storage of loop sequences in
the 7600 instruction stack and the streaming of operands from SCM. The typical operations performed are of two
types, either Dyadic, represented by a FORTRAN loop

\begin{verbatim}
DO I = 1,N 
W(L*1) = U(J*1)#V(K*1) 
CONTINUE
\end{verbatim}

where: # is any arithmetic operation 
W result vector 
U,V input vector (or scalar) 
I a loop index (not necessarily with origin 1) 
J,K,L vector increments (positive or negative)
or Triadic, represented by

\[
\begin{align*}
\text{DO} & \quad I = 1, N \\
W(M \ast I) &= U(J \ast I) \# V(K \ast I) \# Y(L \ast I) \\
\text{CONTINUE}
\end{align*}
\]

Both dyadic and triadic operations are performed by a single subroutine call. For example, the equivalent of the inner product FORTRAN calculation

\[
\begin{align*}
\text{DO} & \quad I = 1, 50 \\
\text{DO} & \quad J = 1, 50 \\
C(I,J) &= 0.0 \\
\text{DO} & \quad K = 1, 50 \\
C(I,J) &= C(I,J) + A(I,K) \times B(K,J) \\
\text{CONTINUE}
\end{align*}
\]

would be

\[
\begin{align*}
\text{DO} & \quad I = 1, 50 \\
\text{DO} & \quad J = 1, 50 \\
\text{QVDOT}(C(I,J), A(I,1), B(1,J), 50, 0, 50, 1) \\
\text{CONTINUE}
\end{align*}
\]

This Stacklib routine executes three times faster on the 7600 than the ordinary FORTRAN object code, at a rate of 13 Mflops.

In general, triadic routines execute faster than two dyadic ones, owing to the saving of intermediate results in fast registers.

Stacklib was developed at the Lawrence Livermore Laboratory, which took delivery not only of the first 7600 but also the first STAR-100 (in 1974). So it was not surprising that very many of the routines had corresponding equivalents in STAR-100. In fact, not only did these STAR-100 routines exist, in the form of subroutines beginning with the characters Q8, but a whole set of emulation routines were developed for the 7600. An interesting result is that the performance can be compared directly on the two machines for vector operations and this has been done. While the peak processing speed in Mflops on the two machines is theoretically about 100:13, the actual ratio is in general significantly worse for the Stacklib routines, sometimes even worse than a unity ratio. It turns out, despite the long pipelines of STAR-100, the concurrency of the 7600 is, in some respects, significantly greater.

3.4 Many-processor systems

We have considered two processor architectures which afford a considerable increase in computing speed for the right kind of problem compared with a purely sequential execution. They both meet somewhat different requirements and make different compromises. Pipeline architecture is manifestly better for operations on long vectors, while multiple functional units perform better for problems that can be organized in rather short fast instruction loops. Nevertheless, there is another alternative which is not only architecturally different, but which may perform exceptionally fast for certain kinds of problem. This is the use of many processing elements (PE) of the same kind, usually arranged in a regular array, either linear or square, and operating under a master control.

To see the motivation for using such a processor array, we need only consider the addition of two \( n \times n \) matrices. Here, the \( n^2 \) additions could be performed completely in parallel in a single addition time, were there available \( n^2 \) adders. Another example might be in pattern recognition where the identical transformation needs to be carried out on every picture element (pixel), and this can be done for as many elements as there are processors.
The idea of using regular arrays of processors appears to go back to Unger in 1958, although there is mention in his paper of earlier work by Huffman at MIT, unfortunately without an explicit reference. The primary motivation for the Unger proposal was to solve problems of pattern recognition, for which purpose he suggested a rectangular array of several hundred bit-serial processors all under the control of a master processor. Each PE would communicate with the four adjacent elements in horizontal and vertical directions, but linking to the four diagonal neighbours could be arranged easily.

The Unger paper is a remarkable one for its time, because it suggests not only the architecture of subsequent machines like SOLOMON, ILLIAC IV and the DAP, but because it clearly foresees the essential role of monolithic integrated circuitry in implementing any machine with so many components. Moreover, very many of the contemporary attempts to implement pattern-recognition computers reflect precisely the architecture suggested by Unger (and indeed made with VLSI components!). And finally, although the specific proposal was for a two-dimensional array of single-bit processors, there is the explicit mention that the arrays could be with more dimensions or of different shape, and that the processors might operate on bit-parallel words.

The first machine actually designed with processor-array architecture was the SOLOMON computer, along the lines proposed by Slotnick in 1962. It was to be a $32 \times 32$ array of inter-coupled processors, each operating in bit-serial mode and with 4096 words of local storage. All internal PEs could communicate with their four nearest neighbours, while the edge elements could be used for input-output. The PEs were controlled by a single master processor which could transmit both commands and data words to them. The commands would usually be broadcast, in the sense that all PEs would perform the same instruction at the same instant, but on different operands. There was the possibility however that different PEs could perform different instructions concurrently.

The main application of SOLOMON was to be in the solution of linear systems of equations and in the numerical solution of partial differential equations, principally by finite differences. But the choice of bit-serial operations and the technology available to implement the hardware did not result in a very effective solution, and a complete machine was never built. A much more powerful machine based on the SOLOMON idea, the ILLIAC IV, was produced however, and this overcame many of the principal difficulties. Nevertheless, the ILLIAC IV did not appear until about ten years later!

ILLIAC IV, although sharing the architectural idea of SOLOMON, was in all other respects a very much more powerful machine. In particular, the PEs could operate on 64-bit words in parallel, and the control processor, instead of being a fairly limited machine, was, in ILLIAC IV, a Burroughs B6500. Although a PE had not the comprehensive command repertoire of the 6600, each one had an approximate speed of more than one 6600 even allowing for the parallelism of multiple functional units in the 6600. The ILLIAC IV was a formidable machine indeed, and one cannot resist recalling (in another context, of course!) the words of St. Matthew (vi. 28):

"And yet I say unto you, that even Solomon in all his glory was not arrayed like one of these"...

Unfortunately there was only one of them, for ILLIAC IV was premature in hardware implementation and suffered from many difficulties arising from being first in using bipolar monolithic technology on a massive scale, and from other problems. Actually, only one quadrant of $8 \times 8$ processors was ever built. Part of the problem in exploiting the very considerable power of ILLIAC IV was that each PE could communicate directly only with its four nearest neighbours (i.e. $\pm 1, \pm 8$); other transfers of operands needed to be propagated through successive links between adjacent PEs.

Despite the difficulties of ILLIAC IV, there is no doubt that it was very effective in solving a large number of applications problems. Moreover, a wide range of software techniques and numerical algorithms were developed which have had perhaps even more influence on subsequent exploitation of parallelism than the machine itself. A very great deal of thought was devoted also to the problem of data structuring, even before ILLIAC IV became fully operational. This pioneering work, for example that of Kuck, showed that it was as important to choose an effective algorithm for data structures as it was for the computation itself.

The principal difficulty of ILLIAC IV was not the aggregate processor power available, but that it was difficult to get the required operands in and out of the PEs fast enough. The problem lay in the basic structure of memory in relation to the processors, with each PE having direct access only to its corresponding 4096-word memory. To solve this problem Burroughs undertook to build another machine, the BSP, which would be an outcome of the ILLIAC IV experience but free from many of the difficulties. Of course, each PE in the BSP, as well as the memory, could be implemented in much more reliable and indeed faster VLSI circuitry. And the control processor and associated file
memory could also be much more powerful and reliable. But the principal difference was simply that all the parallel PEs are coupled to main memory not in a set of corresponding banks, but through an alignment network. This is pictured in Fig. 10, which shows that all the PEs in concurrent operation can access any part of central memory for input operands or placement of results. Because alignment networks get rapidly more complex as the number of connections increases, the number of PEs was limited to 16. The memory cycle is 160 nanoseconds and most PE operations take two cycles. So there is a maximum processing speed of 50 Mflops. For triadic operations involving three input operands and one result, it can be seen that there is a balance between memory access and processing speed.

The total scheme of Fig. 10 is very flexible indeed, and further enhanced in performance by the fact that the total loop (shown with arrows) is pipelined; operand fetching, instruction decoding, processing and result storage are all overlapped. And one of the best aspects of the BSP was that as well as being a processor array, it had features for array processing. Like the ILLIAC IV, the BSP processors operate in lock-step, executing the same operation in parallel on different operands. But, unlike ILLIAC IV, the BSP allowed streaming operations with single macro instructions, with linear vectors of arbitrary length. Indeed, in the BSP, the handling of vectors with lengths not some integer multiple of the number of PEs, is done entirely by the control hardware. In one respect also, the BSP streaming arrangement is even more powerful than that in the pipeline machines mentioned, in that successive operands need not be in successive locations in interleaved memory. This is achieved not only by the alignment networks, but by having 17 memory access ports (a prime number!). The chance of access conflict with regular arrays of operands is thus small.

The BSP was built and partially tested, but was withdrawn as a product for sale in late 1980. Had it appeared for general use it might have been a rather powerful machine capable of implementing a wide range of applications. Two other machines need to be mentioned however, since they too are based on the original Unger idea, and both have been built. They are the DAP and the MPP.
The ICL DAP was first delivered in 1980 and consists of a 64 × 64 array of single-bit PEs of relatively modest speed. Each PE can communicate with its nearest few neighbours and with 4096 single-bit elements in DAP memory. There is thus a total of 2 Mbytes of DAP memory, which is part of a larger memory in a host machine of the ICL 2900 series. The arrangement is shown in Fig. 11. In addition to the connection of a PE with nearest neighbours for operand transfer, there is the facility for transferring a ripple carry from the adder in one PE to its neighbour in a row. In this way, arithmetic can be carried out either by all the single PEs acting in lock-step on successive bits of words stored "vertically", or by rows of 64 PEs acting on successive "horizontal" 64-bit-slices; in the second case, 64 words can be processed completely in parallel.

It can be seen that the most powerful feature of the DAP is that the problem of coupling the processor array to memory is largely solved, since the data can be structured both horizontally and vertically, or the structure can be transformed in some intermediate way by the attached host processor. Nevertheless, for floating-point arithmetic on long words, DAP is not very well suited. For 32-bit operands, the addition time for 4096 elements is about 150 microseconds, and 280 microseconds are needed for multiplication, giving peak rates of about 27 and 15 Mflops respectively. However, for operations on bit arrays, for example pattern recognition, DAP performs very well.

Perhaps the greatest criticisms of DAP are that vector streaming operations are not implemented in hardware and that, in any case, it would be very difficult to attain the very high operand throughput required by all the PEs processing in parallel. The memory and I/O throughput of the host 2900 are not adequate to support such high bandwidth streaming, and in any case the 2900 can only access storage by cycle stealing with the DAP. It seems a pity that DAP could not be implemented as a separate back-end to one or more attached hosts, and with many concurrently active ports to the DAP memory. A very powerful machine indeed could be made using the DAP processing scheme, but with a number of other processors surrounding and accessing the memory so as to input and output operands and to re-align the data structure as necessary. These "data-structuring" processors need only be quite modest, since they only require to handle bit strings and not perform much arithmetic.

The other array computer mentioned, the Goodyear massively parallel processor (MPP), is due to be delivered about now (mid 1982). It consists of very large number (128 × 128) of single-bit PEs, designed all to operate in parallel on concurrent data streams of the same kind. Indeed the principal application of MPP is concurrently to process the pixels from satellite images. For this purpose, 8-bit integer operation is usually adequate and 6000 or more MOPs can be obtained for additions in this mode.

Each PE in the MPP is considerably more powerful and faster than those in the DAP, and the level of integration much higher. Nevertheless the PEs are coupled in the classical way only to nearest neighbours, since any form of alignment network to memory was considered (and indeed is) far too difficult to implement. The problem of data structuring was tackled in two ways: first, by having a programmable topography for the edges of the PE array, so that data of any width can be moved across the array in any direction; second, by externally re-ordering data arrays by attached processors. In this second arrangement, there is a common 2^19-bit memory buffer for data arrays which has several independent access ports in different directions. Data can be transferred at a sustained speed of 160 Mbytes per second, and can be read out in one direction at this rate and re-written in another.

Although the MPP is intended for image processing, it can be used also for floating-point operations on arrays. For additions of 32-bit vectors, speeds of up to 430 Mflops may be attained, with only half this speed for multiplications.

3.5 Special Purpose Computers

Since the stored-program computer came into being, despite the intrinsic generality of a programmable device, there have been many developments of special-purpose machines, and this is true as much for concurrent systems as for sequential-processing ones. One of the very earliest proposals for such a computer appeared in 1952, intended for the solution of elliptic partial differential equations. It was called DINA\(^{24}\), and it is pertinent to remark that the motivation to build it was the same as for machines such as ILLIAC IV. The reason for a special-purpose architecture was simply that much higher processing speeds might be obtained for the particular application.

DINA was based on an implementation of the finite difference method for solution of the equation \(\nabla^2 u = F\), where \(F\) could be a number of different functions (including time-dependent ones). In this way the solution required successive addition operations for values of the potential \(u\) at nodes of a Cartesian mesh. This was to be done in DINA using iteration through two arithmetic units (with bit-serial arithmetic), the units having adders and shifters. A remarkable feature of DINA was that these units were to be pipelined and to function synchronously in parallel. It is probably the first example of a proposal for pipelined architecture.
DAP INSTRUCTION BROADCAST AND MASTER CONTROL UNIT

Fig. 11 The DAP memory architecture
Since the early days of DINA there have been many other proposals for special-purpose systems, some of which have been implemented. An example of growing importance is the inner-product computer, an early version of which was suggested by Swartzlander\textsuperscript{25}. This proposes an iterative multiplier and an adder in a pipeline configuration to accumulate successive terms of the result. There are many applications of such devices, one of which is in transaxial x-ray tomography. This involves an image reconstruction algorithm using convolution of each density profile with some pre-specified kernel, and each convolution requires a large number of inner products. For a typical image, forty thousand 200-element inner products might be needed.

It is relevant to remark here that the advent of VLSI circuitry has made possible the implementation of fairly inexpensive special-purpose processors which employ parallelism, and that these are now beginning to appear in a fairly widespread way. A good example is in signal processing, where certain of the latest high-speed modems now use pipeline processors of 20 Mflops capability for Fourier transforms.

The foregoing examples of special-purpose computers are for very limited applications and do not involve much hardware. There are many other instances. However there are several examples also of very large systems which, although intended for a specific purpose, are nevertheless of fairly general properties. A very important category of such machines derive, just like SOLOMON, ILLIAC IV and DAP, from the original Unger suggestion for a regular processor array. These machines stem from the important realization that, provided memory and processors can be interconnected in an appropriate configuration, then very many data blocks can be inspected (or processed) in parallel to see which of them conform to a set of logical templates or keys, usually a set of masked bit strings. This is the basis of so-called associative processing, a concept that is probably due to Slade and McMahon in 1957\textsuperscript{26}. In these machines, the basic operation is that each word in a given field in memory is compared in parallel with a query word defined in the processor array. The essence of the problem thus is not so much the processing, which is fairly straightforward, but the interconnection of processor array with memory.

Perhaps the most successful example of associative processor arrays and one of the first to be put to practical use is the STARAN system\textsuperscript{27}, the first production model of which appeared in 1972. Figure 12 shows the arrangement of one module of STARAN, of which there can be up to 32. All the modules are controlled by a separate control unit and are attached to an external host computer. The basic module is an array of 256-bit processors which couples to a 256 × 256-bit memory module. The two are intercoupled by means of an alignment network, termed here a flip network, whose function it is, in a single memory cycle, to access 256 cells either in bit-slice, word-slice or mixed mode (words and bits). It can be seen that the STARAN is a very powerful machine indeed for pattern recognition, and several systems have been delivered already. More recent plans are to implement a large version of STARAN, in which each module would have a memory array of 256 × 9216 cells.

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{staran_diagram.png}
\caption{The STARAN associative array}
\end{figure}
In connection with this discussion of associative processing, we should mention that many applications require the systematic associative search through large blocks of data of a similar kind. In this case, it does not matter where the data are stored or in what order; simply that the data should be presented sequentially to a parallel array of processors (or sets of processor arrays, where several reference words can be compared concurrently to the same data). A very attractive possibility here is to make use of rotating memory for the block storage (disk or drum), but to have one processing element per track. Then, a complete search of all the data can be performed in a single revolution. This idea of a logic per track device was first proposed by Slotnick\(^2\) and it is one of a class of block-oriented associative processors. Many machines have been built based on this idea.

A rather more complex processor array PEPE\(^3\) also has a large number (288) of PEs which, like STARAN, have an associative processing capability. But PEPE also has memory directly attached to each PE, and an additional arithmetic processor. PEPE is designed so that each PE operates on the data within its own memory only, but the processing can be associative, purely arithmetic or correlative; there is a correlation unit which compares processed data with previously predicted data. All the processing units in the PEs operate in lock step, but on different data. The main application of PEPE is the concurrent tracking of different objects on the basis of radar information, each PE tracking a different object. This is not, however, the only application by any means.

A very important feature of PEPE, distinguishing it from other processor array systems, is that it has two instruction streams, the first for the arithmetic unit and the second for the correlation unit. And the correlation unit can be used (together with others by inter-communicating data) to perform logical operations of an imprecise kind. For example, an operation can be performed to determine whether some set of parameters lie within certain bounds.

A further machine that should be mentioned here is the ASC\(^4\), not so much because its architecture is special purpose, but because the applications to which all the machines delivered were put were rather specialized, mainly in the processing of seismic data.

The ASC was first delivered in about 1973, with a prototype machine available some years earlier. It is a pipeline machine, very similar to STAR-100 in many respects and was delivered in more copies than the STAR-100. There could be up to four pipelines in ASC, each with 32 or 64-bit arithmetic and a 60 nanosecond segment time. Up to 36 instructions could be in various stages of processing concurrently in four pipes, and various segments of vector streams could be distributed among pipelines.

The principal differences between the ASC and the STAR were that there was no control vector or sparse vector handling commands on the ASC, but strides of other than unity were allowed. Also, certain operations, such as division, needed to be synthesized on the ASC from more primitive operations in the pipelines. The STAR had a richer basic instruction set, particularly for vector streaming, but the ASC did have certain powerful single vector operations (such as inner product). For some stream operations on 32-bit operands, the ASC could achieve processing rates in four pipes of 64 Mflops.

### 3.5.1 Add-on Processors

In completing this discussion of special-purpose parallel processors, mention must be made of a class of machines which are designed to be used in conjunction with some host computer in order to perform a limited number of operations on data arrays. Almost without exception, such add-on machines are based on pipelining and derive their data from the memory of the host machine. Also, nearly all of them are intended to solve problems arising in geophysical work, such as oil prospecting or seismic applications.

Some of the earliest examples of add-on array processors were from the manufacturers of large main-frame computers. For example, in the period 1972–1977, IBM made the 2938 and 3838, as an optional processor to be coupled to an IBM host system by a channel connection. The 3838\(^3\) consisted of a pipelined floating-point multiplier and two pipelined adders, as well as a number of other arithmetic units including a sine-cosine generator (to support Fast Fourier Transforms). The 3838 was not directly programmable in the usual sense, but had an extensive set of commands implemented internally by microcode. These commands could be invoked from the host machine in an appropriate sequence, and they controlled also the data flow between host memory and 3838 working store. The command repertoire was very comprehensive, ranging from division and multiplication of vectors (element by element) by scalars or vectors, right through to FFT algorithms on real or complex input data.

The 3838 was a rather powerful machine, but unfortunately only of use to IBM installations, and also fairly costly. Much more universal and far less costly was the AP–120B of Floating Point Systems\(^3\), now being replaced by a somewhat more comprehensive FPS–164.
The AP-120 B is basically a processor of two pipeline units, a three-segment multiplier and a two-segment adder, both with a segment time of 167 nanoseconds. In the processor are memories for operands and program, as well as a table memory for needed constants. The AP-120B is attached to some host computer, which may simply be a minicomputer, from which it derives data in the form of operand streams and programs in the form of a sequence of CALL statements. Each call is compiled by a FORTRAN compiler into a set of instructions for the AP, which are essentially microcoded. The microcode is then used to control all the units within the AP concurrently.

The AP-120B does not process vectors directly from host computer memory. Instead, operands are transferred first to the AP and there processed. In this way, intermediate results can be used together with new operands without waiting, and the AP can function autonomously of the host computer. Leaving out some details, a typical command sequence for inner product might be as follows:

\[
\begin{align*}
\text{CALL APCLR} & \quad \text{Clear array processor} \\
\text{CALL APPUT(A,I,N)} & \quad \text{Transfer first array to AP} \\
\text{CALL APPUT(B,J,N)} & \quad \text{Transfer second array to AP} \\
\text{CALL APWO} & \quad \text{Wait for Data} \\
\text{CALL DOTPR(A,I,B,J,C,N)} & \quad \text{Perform A} \cdot \text{B} \\
\text{CALL APWR} & \quad \text{Wait for completion} \\
\text{CALL APGET(C)} & \quad \text{Return result to host}
\end{align*}
\]

This sequence, equivalent to

\[\sum_{m=1}^{N} A_{ml} \cdot B_{ml} \rightarrow C,\]

is executed at an asymptotic rate (for long vectors) of about 6 Mflops.

The AP-120B has been a highly successful machine, particularly applied to problems of signal and image processing, where FFT algorithms play an important part. Some of this success is certainly due to the low cost of the add-on hardware, but the principal reason is that it is the most effective solution for a certain class of problems which can exploit concurrency. This point has been realized by the manufacturers of large main-frames, and we may now expect them to produce add-on processors of the 3838 type, but much less expensive. Indeed, in a recent development, Hitachi have announced an “internal” add-on pipeline unit called an integrated array processor (IAP). The IAP is an optional part of the Hitachi S8 or S9 CPU, and is accessible through FORTRAN-compatible source code to perform a set of array arithmetic (there are up to 46 additional machine instructions). In the IAP, there is the concept of a control vector (as in the STAR), but general sparse matrices are difficult to handle.

Many other add-on processors involving concurrency have been proposed, including for example an auxiliary associative parallel processor, to mention only one attractive possibility. What has inhibited many of these proposals up to now is the cost and that there has been very little user experience to gauge the potential benefits. The position is now changing in both respects, since VLSI technology is reducing hardware costs, and users are gaining experience of large machines with concurrency. We may now expect very many more new developments along the lines of what Hitachi and FPS are doing.

It should be stressed that there is a big difference between what Hitachi are doing, and what is done with the FPS and the 3838. The former is coupled directly to the memory of a large processor, while the latter are at the end of a channel.
4. SOME LIMITATIONS TO CONCURRENCE

In the last few pages we have seen that concurrency in hardware can be exploited in many different ways to achieve faster speeds in arithmetic and logical computing operations, ranging from the most elementary operations to quite complicated ones. The question may reasonably be posed however as to any limitations that might exist to the process of incorporating ever more parallelism in our computing systems. Intuitively we would expect such limitations to exist, since we have seen how concurrency is generally implemented at the price of increased complexity, certainly at the hardware level. Let us therefore look at some of the principal barriers to employing parallelism.

4.1 Limitations within functional units

Nearly all functional units have been the target of attempts to implement them with as much concurrency as possible. Indeed it is probably more exact to say that all the units of any conventional computer have been candidates in this process. A good example of an elementary operation here is shifting since, as long ago as the ACE proposals by Turing, there was a suggestion for a parallel shifter (called then a “rotater”) despite the natural adoption of serial shifting in delay-line registers. Shifting, whether it be circular or logical (with empty outputs filled by zeros), is basically the problem of moving a bit pattern from one n-bit register to another by a specified number of places, in a defined direction, in accordance with an m-bit shift-distance count. To do this serially is simple with hardware, but slow (on average), since each bit shifted requires a clock pulse. It can however be done entirely in parallel, that is always within one clock pulse irrespective of shift distance, provided that an appropriate alignment network is interposed between the two registers. The problem here is that such an alignment network would need to have n output connections for every bit of the input register, a very complex affair indeed. In practice this is considered too complicated, and so a compromise is adopted of having a binary or quaternary shift tree. In such a tree, the input word is shifted say 0, 1, 2 or 3 bits at the first level, controlled by the two least significant bits of the shift count, 0, 4, 8, 12 bits at the second level, 0, 16, 32, 48 at the third level, and so on. For a 64-bit shifter, three quaternary levels would thus be needed with a maximum speed of 3 clock periods irrespective of shift length.

In the case of integer addition the situation is not quite so clear. At least, in the example of shifting, concurrency can be used to bring down the operation time to that of a single logic gate (neglecting propagation delays of interconnections, and the decoding time of the shift count—not always negligible!!). For addition this is just not possible.

A basic adder is shown in Fig. 13, in which successive digits of two numbers \( A_n \ldots A_1 A_0 \) and \( B_n \ldots B_1 B_0 \) are added in the single-bit adder elements \( \Sigma \). \( \Sigma \) differs from all others in the absence of an input carry, but that is just a detail. All the bit adders can be implemented with combinatorial logic elements in a time of about two gate delays (2\( \delta \)). The two simple implementations of an adder are: entirely sequential, where a single bit adder is used iteratively to add successive digits, or with \( n \) adders, as shown in Fig. 13. In the first case, there must be a delay in feeding back the carry from given bit addition to the successive bit position, typically 2\( \delta \) and so an \( n \)-bit addition would take about 4\( n \delta \) to perform. In the second case, the time would be about halved, since any adder can operate as soon as the carry is available from a preceding stage. For this reason the scheme of Fig. 13 is called a ripple-carry adder.

\[ \begin{array}{c}
\text{Fig. 13 A basic parallel adder}
\end{array} \]
The serial adder is clearly economical but slow, while the ripple-carry adder is much more costly and not much faster. In this example of concurrency, the extra hardware parallelism does not gain us very much. There is, however, a further possibility, and that is to make use of the knowledge about how carry digits propagate. There are many schemes to do this and we will mention only two.

4.1.1 The carry-look-ahead adder\(^{33}\)

If we consider any given stage in an \(n\)-bit addition, we observe there are only three possibilities for carry propagation: if the input bits \(A_i, B_i\) are ones, then the carry to the successive stage is always 1 irrespective of the value of \(C_{i-1}\); if the input bits are zeroes, the carry will always be zero; if \(A_i\) and \(B_i\) are different, then the carry will simply be a copy of the input carry \((C_i = C_{i-1})\). It turns out therefore that a combinatorial circuit can always be built to compute the carry at the input to any stage as a function of all the previous input bit pairs. The trouble is however that such a combinatorial circuit would, for large \(n\), require not only very many logic elements (and so large propagation time), but the number of input lines to certain elements (the fan-in) would need to be impossibly large for modern technology.

In practice one cannot win by the pure carry-look-ahead scheme, except for small \(n\), no matter how good it sounds in principle. What can be done is to make a compromise by segmenting the input bits into blocks, performing the carry-look-ahead scheme within each block, but generating a real carry which ripples through from block to block. Or one can try carry-look-ahead at more than one level, by using “super-blocks”. In any event, the limit is set by the restrictions on fan-in and fan-out of modern technology. For a 64-bit adder, it seems that the best one can achieve is about 10 gate delays, but at the expense of nearly 200 logic elements.

4.1.2 Conditional-sum adders\(^{34}\)

This is a completely different scheme from the previous one, but also divides the input operands into segments of only a few bits each. The method depends upon the simple fact that the carry from any segment can only be zero or one. So both sums are computed “conditionally” and then the correct one is selected as soon as the input carry becomes known.

The logic scheme to implement a conditional-sum adder is a complicated one, which is further complicated by requiring a large fan out (typically \(n/2\) for \(n\)-bit operands) for some of the logic gates. But practical adders of this type can achieve addition times of \(O(\delta \log_2 2n)\), provided \(n\) is not too large.

4.1.3 Ultimate limitations

We have seen in the two foregoing examples of adder that considerable improvements can be achieved in speed provided that the complexity of circuitry is increased. We can return therefore to the basic question as to the ultimate limitations on the process, in this case addition of two binary numbers. This general question was answered in a classical paper by Winograd\(^{35}\), who found that there is a lower bound on the addition of two numbers modulo \(\mu\); it scales like \(\log \log \alpha(\mu)\), where \(\alpha(\mu)\) is the largest power of a prime which divides \(\mu\). In practice, because there is a limitation on fan-in of practical logic elements (\(r\)), this in turn imposes a limit to the number of segments into which an adding process (of which the conditional-sum adder appears to be the best) must be divided. So the Winograd formula in this case reduces to \(\left\lceil \log_2, 2\left\lceil \log_r \alpha(N) \right\rceil \right\rceil \delta\) for binary radix addition. For a fan-in \(r = 4\), gate delay \(\delta = 1\) nanosecond, and 32-bit operands, the best possible addition time turns out to be about 3 nanoseconds. It is most interesting to note that a real computer with the same practical circuit parameters actually achieves about 5 nanoseconds — which is pretty good.

One other observation to make is that the Winograd limit depends upon the radix chosen. This is nearly always \(\mu = 2\), but an improvement in addition time might be achieved by choosing a greater value of \(\mu\). Again this is a clear example of how to obtain increased operational speed, but at the cost of complexity, in this case the complexity of encoding for higher radix representation. In practice this complexity would lose more than one wins in the addition time!

A similar analysis for the lower bound in operation time has been performed\(^{36}\) for multiplication modulo \(n\), in which the surprising result is obtained that this bound does not differ greatly from that for addition. But this is only the case if the number code chosen is not radix-2, and the circuit complexity is much too great to be implemented in any practical arrangement. In practice we have seen that multiplication is performed like addition with radix-2, and that the complexity is limited by the practical fan-in and fan-out ratios that VLSI circuits can handle.

N.B. \(\lceil x \rceil\) is the ceiling function and \(\left\lfloor x \right\rfloor\) denotes the smallest integer \(y \geq x\)
4.2 Limitations in the number of functional units

We saw, for example in the 7600 computer, that a number of different functional units could give improvements in computational speed even for a single stream of instructions, as a result of concurrency inherent in the algorithm used for processing. Indeed the aim of Stacklib was precisely to identify such algorithms, and then to code them for the 7600 in an optimum fashion. Nevertheless, although there is no limit in principle to the number of different functional units within a given processor, there are clear limits to what is reasonable in any practical computer. There is no point in having many idle units because there are either no operands or instructions available at a given moment. In the end, the real limitation is set not by hardware, for it is becoming very easy indeed to replicate VLSI units, but the algorithms that can be found embodying parallel code segments. We shall return to this point later.

4.3 Memory and data-streaming

Parallelism in memory is used as a method of improving the average access time both for operands and for instructions, principally by the technique of interleaving. Of course, high-speed registers, caches and other buffering techniques are also essential, but these only speed up the peak in data streaming; what they cannot do, except for relatively short segments, is to speed up the average operand throughput. Thus, the question arises as to what limitations there are in the interleaving strategy.

Figure 14(a) shows a 4 X 4 array X stored sequentially in a memory of four interleaved units. Clearly, if individual rows or diagonals are required, then the elements can be streamed at the peak memory rate (four elements per memory cycle). However, if columns are required, then the maximum rate is only one element per memory cycle. In another arrangement (b) the storage is skewed, so allowing fast access either to rows or to columns; but now diagonals cannot be accessed without conflicts. To overcome this problem, one may have skewed storage, in which the number of memory units interleaved is greater than the number of row elements, as in (c). Then rows, columns or diagonals can all be fetched at the peak rate.

![Fig. 14 Some array storage choices: (a) straight, (b) skewed, (c) skewed with m ≠ n](image)

Clearly there is a problem with interleaved memory if arrays stored there are indexed by some stride which is a factor of the number of memory units. This is particularly a hazard if the number of processors in a parallel array has some simple relationship to the number of memory units. To avoid this, the number of memory units is sometimes chosen to be relatively prime to the number of processors. In the end, however, for a fixed coupling between interleaved memory and processor array, there is no way to achieve a conflict-free access in all cases. The only way out here is to use an alignment network between processor(s) and memory.

We shall not discuss alignment networks here, since it is a complex subject in its own right, but we will just remark that we encounter analogous problems of complexity here as we do with other parallel structures. Although it is indeed true that it is possible to devise some alignment network for each concrete configuration of memory units and processors, so as to be able to map one set of nodes onto the others without any access conflicts, it is invariably the case that there is a price to pay in implementation of the alignment network itself. Usually this price is the complexity of the network arising from practical limitations on such factors as fan-in and fan-out of circuits, delay time of different routes and the non-negligible switching time of the network itself.
We should remark here also, in relation to alignment networks, that it is not only whole words that need to be assembled in their required order into arrays, but also other data units such as half-words, bytes and even bits. Moreover, in many instances we need to be able to broadcast (say constants), so that there is a requirement for more than just a one-to-one connection of inputs to outputs.

Just to illustrate the problem, even the most regular and simple network for connecting all inputs to outputs in a one-to-one relationship, namely the crossbar arrangement shown in Fig. 15, requires \( O(n^2) \) gates to implement for \( n \) inputs and outputs, but the time taken for data transmission is \( O(\log n) \) as is the time taken to set up the alignment. The crossbar arrangement is thus a very bad solution for large systems.

![Fig. 15 A crossbar alignment network](image)

Other systems such as the Batcher\(^{37} \) network, the Omega\(^{38} \) network and others, can either perform the switching in shorter time or use fewer gates to implement, but only at the cost of increased complexity. It seems the best one can do for a one-to-one network is about \( O(n \log n) \) gates.

### 4.4 Operand flow and data dependencies

Each process in a computer is expressed in algorithmic form, which we can assume for the purpose of this discussion is deterministic, correct and implementable. But once the algorithm has been chosen and coded in the best possible way (we ignore here any compiler inefficiencies), there can be limitations in the number of separate steps or operations that can be executed concurrently, even if there is no problem either in the number of processors (or operational units) available or in memory access.

Consider, for example, the process illustrated in Fig. 16, namely the addition of a series of 8 operands \( \sum_{i=1}^{8} x_i \). Here, irrespective of the ordering of operands, one cannot evaluate the expression in less than 3 levels of a binary tree, never mind how many adders we may have available. Moreover, operations at one level cannot begin until all the results of previous levels are complete. It is thus easy to see that any expression of this type \( E(n) \) just cannot be evaluated in less than \( O(\log n) \) operation times, and that more than \( n/2 \) processors also give no further improvement.
There are other arithmetic expressions which can only be evaluated in much longer times and can exploit only far fewer concurrent operational units. For example, if the algorithm chosen to evaluate the polynomial of degree \( n \)

\[ P_n(x) = \sum_{i=0}^{n} a_i x^i \]

is the well-known (and usual) Horner's rule

\[ P_i = a_i + x \cdot P_{i+1} \quad \text{for} \quad i = n-1, n-2, ..., 0 \]

and

\[ P_n = a_n \]

then only one functional unit can be used at a time (alternately multiplication and addition), regardless of the number of units available. The evaluation time here is thus \( O(2n) \).

Of course, there are means to reduce the computation time in certain cases, where many functional units are available. For example, we can make use of properties such as associativity or commutativity to reduce tree height. One such illustration is shown in Fig. 17, where (a) shows the best that can be done to evaluate the expression \( A(BCD + E) \), namely 4 time units using a single functional unit at a time. By using the distribution of multiplication over addition, however, we can improve the time to 3 units here by evaluating \( ABCD + AE \). This time, we should note that we require 3 concurrent functional units.

Much work has been done in the area of tree-height reduction using well-known properties of arithmetic operators, but in the end, no matter how much parallelism we have, there is always a lower bound. It appears that, for arithmetic expressions, we can never have less than about \( O(\log n) \) levels, for which we would need at least \( O(n) \) processors.

We see thus that the availability of operands to process, and an unlimited number of processors to perform operations, does not, in itself, guarantee that we can improve execution speed beyond certain limits, for a given algorithm. What we can do, however, is to choose a better algorithm, at least in some cases.

The example of polynomial evaluation is a special case of linear recurrences, which are very important in computational processes. The case of computing the inner product of vectors is another important example. For such recurrences, there are general theoretical arguments\(^{39} \) to suppose that, subject to a number of important conditions, we might evaluate the expressions of order \( n \) in \( O(\log n) \) time steps using only \( P = O(n) \) processors. Unfortunately, it is one thing to identify theoretical bounds, and quite another to find concrete algorithms that come some way near these bounds. Nevertheless, in the two examples cited, methods have been found which come fairly close.
In one good example of what can be done, Maruyama\textsuperscript{40} has shown that polynomials of degree $n$ might be evaluated in $\log n + \sqrt{2} \log n + O(1)$ time steps if an unlimited number of processors were available. However, he also describes a number of algorithms, based on computational trees implementing generalizations of Horners' rule, which come fairly close to the theoretical lower bound. For example, for a polynomial of degree about 1000, 14 time steps are required compared with the theoretical bound of 11. This is not only an excellent result, but should be compared with the time required to evaluate the same polynomial on a uniprocessing computer!

The other very important case, of inner products of two vectors, the theoretical lower bound of $O(\log n)$ can indeed be attained. Here, however, the important problem is what happens when we want to perform a large number of inner products on a smaller set of vectors, as is the case for example when we require the product of two $n \times n$ matrixes. In this case, instead of algorithms which minimize the time for evaluating the single inner product, what we seek is a more global algorithm for optimizing the whole process of matrix multiplication.

Just to illustrate what might be done, let us mention two distinctly different approaches. In the first, the strategy adopted is to minimize the total number of multiplications involved, which is usually $n^3$ for the $n \times n$ matrix product. This has been done by Winograd\textsuperscript{41} who, for two vectors $x = (x_1, ..., x_n)$ and $y = (y_1, ..., y_n)$ pre-computes the quantities

$$\xi = \sum_{j=1}^{n/2} x_{2j-1} \cdot x_{2j}$$

$$\eta = \sum_{j=1}^{n/2} y_{2j-1} \cdot y_{2j}$$

and then calculates the inner product from:

$$(x \cdot y) = \sum_{j=1}^{n/2} (x_{2j-1} + y_{2j}) \cdot (x_{2j} + y_{2j-1}) = \xi - \eta$$

This expression is for $n$ even, with a similar one for odd values of $n$. 
Although the Winograd method does not reduce the number of operations for a single inner product, what it does do is to reduce the number of multiplications required for the \( n \times n \) matrix product from \( n^3 \) to \( n^3/2 \). There are also significant gains to be made in an analogous way in matrix inversion (using Gaussian elimination) and in solutions of systems of linear equations.

Perhaps more central to the exploitation of parallelism, however, are methods which do not reduce the number of arithmetic operations in the matrix multiplication, but rather do more of them in parallel. Thus, in this second approach, instead of computing a single term of the product matrix at a time, namely a single evaluation of an inner product, we seek to compute more than one. To make this clear, let us see the three main possibilities. In the first technique, the product \( Z \) of two \( (n \times n) \) matrices has elements

\[
z_{i,j} = \sum_{k=1}^{n} x_{i,k} \cdot y_{k,j}
\]

which are evaluated by the FORTRAN loop

```
DO I = 1,N
  DO J = 1,N
    DO K = 1,N
      Z(I,J) = Z(I,J) + X(I,K) \cdot Y(K,J)
    CONTINUE
  END DO
END DO
```

(1)

For a serial machine there would be \( n^3 \) multiplications and \( n^3 \) additions executed sequentially. For a parallel machine, we would at least evaluate the innermost loop (i.e. a single inner product) by a tree of the type shown in Fig. 16, and in this way compute each of \( n^2 \) inner products in \( O(\log n) \) time steps rather than \( 2n \). So we would need \( O(n^2 \log n) \) rather than \( 2n^3 \) units of time.

But, of course, we can do even better, either by computing \( n \) inner products concurrently or by building up all \( n^3 \) products in lock step. The first of these, the so-called middle product technique is equivalent to interchanging the I and K loops in (1) and then evaluating the assignment statement in parallel, \( n \) at a time. We would thus have

```
DO J = 1,N
  DO I = 1,N
    DO K = 1,N
      Z(I,J) = Z(I,J) + X(I,K) \cdot Y(K,J)
    CONTINUE
  END DO
END DO
```

with the parallel (vectorized) evaluation of all iterations of the innermost loop. And we observe that this evaluation is a triadic one, and so well suited to machines which can overlap the addition and multiplication in successive iterations.

The last possibility moves the loop over K in (1) to the outside to give

```
DO K = 1,N
  DO I = 1,N
    DO J = 1,N
      Z(I,J) = Z(I,J) + X(I,K) \cdot Y(K,J)
    CONTINUE
  END DO
END DO
```

and then, for each value of K, evaluates the assignment statement for every combination of I and J. Thus, in this outer product method there are \( n \) evaluations of \( n^2 \) terms in each of the \( n^2 \) inner products in parallel, namely the parallel execution of all iterations in both inner loops.

In the end, however, there is a limit to what parallelism can offer to increase the speed of matrix multiplication. If one had enough parallelism, one could in principle evaluate all the required products in parallel, namely all \( n^3 \) concurrently, and then sum all \( n \) terms of the \( n^2 \) inner products in \( \log n \) steps. This then would be the best that could be done, but there would have to be a parallelism of \( O(n^3) \) available in the hardware, a formidable amount indeed.
Earlier on, in connection with the discussion on the lower time bound to evaluation of linear recurrences, it was mentioned that there are certain conditions. We shall not go into all of them here, but we shall just list some of the relevant assumptions made in our foregoing discussion:

i) Each arithmetic operation takes the same time
ii) No access conflicts to registers or memory occur for operands
iii) There is no limit on the number of available processing units
iv) No time is taken to move operands to processors.

It is clearly difficult to satisfy all these conditions in any real machine, but it is relevant to remark that these assumptions are precisely those on which a new class of machine has been based, namely the data-flow computer\textsuperscript{42}. In data-flow systems, an operation can proceed as soon as operands are available, so that as many operations can proceed concurrently as there are available input operands. This is clearly a way automatically to exploit the maximum degree of concurrency available in the computational process to be executed. We will not discuss data-flow machines further, except to observe that, in the end, the degree of parallelism to be exploited for any given algorithm will be limited for these machines, as it is for any others, by the same lower bounds as we have already indicated.
5. **CLASSIFICATION AND PERFORMANCE**

We have discussed several different machine architectures and many forms of concurrency, and we have referred in a rather loose way to relative performances. The question thus arises of classification of different computers which exploit parallelism, and the characterization of their relative performances. There have been several attempts to do this, but it must be said that there is no unambiguous way at present to answer either of these basic questions.

Concerning taxonomy, there have been schemes to devise something quite simple, the best known of which is due to Flynn. He classified machines on the simple view that there could be more than one data stream and more than one instruction stream in any computer, with the possibility that a single instruction could process either a single stream of data or more. Clearly a machine with many concurrent instructions could process a multiple data stream, but the inverse category is not so evident.

In obvious acronyms, Flynn denoted three main types of machine by SISD, SIMD and MIMD. Clear candidates for these three are, respectively, the classical uniprocessing von Neumann machine, processor arrays of the Unger type, for example ILLIAC IV, and any ordinary multi-processor. The trouble is, however, that there is no agreement about any other example.

![Hockney plot for performance comparison](Fig. 18)
For example, Flynn puts pipeline machines into the second category because a pipeline is considered as a
time-multiplexed version of a processor array (with each PE operating in lock-step). In SIMD systems a single
instruction processes a multiple stream of data, and Flynn clearly regards the different elements of a single vector
operand as a "multiple data stream". But others regard pipelining as just an internal way to speed up certain
operations when it is possible to organize a continuing sequence of such operations. In this way pipelining is
considered to be in the SISD category since there is really a separate instruction for each operand pair, only the
instruction happens to be a repetitive one.

The Flynn classification is useful, but overly simplistic. Others have tried to refine it by extending the scheme
to a sub-level, in which one differentiates whether a single instruction unit does or does not control multiple execution
units. Yet others, notably Hockney\(^44\) have abandoned the Flynn idea completely, and introduced a taxonomy based
on a structural notation for machine systems. Unfortunately, the Hockney scheme is really quite cumbersome and
goes into much too fine detail to be useful.

I suspect that there never will be an absolutely satisfactory taxonomy for concurrently operating computer
systems, and that there will always be certain systems whose classification is ambiguous. Perhaps we are condemned
to be like biologists who are, even now, uncertain whether the unicellular organism \textit{euglena} should be classified as a
plant or as an animal! Because of this, I think it better to continue to describe computer systems, as we have done up
to now, by their principal attributes rather than any taxonomical niche.

On the question of characterizing performance, the situation is somewhat better. First of all, despite the
unfashionableness of quoting MIPS or MFLOPS, both these are measures which give an idea of comparative
performances, at least for certain combinations of computers and algorithms. Indeed, if it is actual performance of a
specific program execution, MFLOPS is not a bad measure at all. The difficulty arises only when one wants to gauge the
potential performance of some parallel machine, for here the answer depends upon the kind of problem to be solved.

To overcome the difficulty of variable performance on different programs, Hockney\(^44\) has proposed that two
parameters rather than one are used. The parameters are the vector length \(n_1\) required to achieve half the maximum
performance (in, say, MFLOPS) and the maximum rate of computation \(r_\infty\), which is usually that rate (in MFLOPS) that
might be achieved asymptotically for "infinitely long" vectors. Performance is then characterized in a
two-dimensional logarithmic scatter plot of \(n_1\) against \(r_\infty\). The Hockney scheme is certainly the best that has been
devised so far, in that it does allow us to compare the performance of machines with quite dissimilar architectural
properties. Figure 18 shows a typical plot of this type, on which some of the well-known parallel machines appear.
6. PROBLEMS WHICH HAVE PARALLELISM

In this section we shall discuss a number of classes of problems where parallelism of computers can be exploited in the numerical domain. Our aim here is not to discuss numerical analysis as such, since there is a vast literature dealing with the subject, but rather to touch upon just a few areas where there is a definite advantage in employing computational concurrency. Before proceeding, however, to specific questions, there is one point of rather general importance which needs to be mentioned, concerning the balance of choice in deciding which methods to adopt.

Even in sequential processing, any numerical analysis for a specific problem must take into account not only the effectiveness of the method, in its ability to find the necessary solution, but also the associated issues of convergence, stability and propagation of errors. Sometimes this is a global problem, for which it is difficult to find an optimal solution. With the possibility of parallelism, however, we introduce a further dimension, and this makes the problem even more difficult to solve. The general point to make here is simply that there is a balance to be found between parallelism and the amount of computation necessary to find some solution. It is no use selecting a numerical algorithm with a very high degree of parallelism if, at the same time, the amount of computation is increased to the same extent (or even greater!).

The problem of efficiency in utilizing computational resources, particularly processors, is of lesser importance. This question, which used to be central in discussions on parallelism some ten years ago, is now of far less significance because of the rapid evolution of monolithic technology and the associated reduction in costs.

Only a few references will be cited in the following discussion. For a fuller discussion, the reader is recommended in the first instance to consult a number of excellent review articles [Refs.: 45, 46, 47, for example]; these articles, in turn, give an extensive bibliography.

6.1 Partial differential equations

It is natural that we should begin with the numerical solution of partial differential equations (PDE), since this was the principal motivation for developing the earliest parallel-processing machines. The numerical prediction of weather is a well-known example here, where atmospheric flow is modelled by a system of partial differential equations in three space variables and time. The idea is simply, based on known conditions at some time (say now!), to predict what might be the flow some several days hence. Unlike in so many other domains, the algorithms used here for such solutions on sequential processors already involved parallelism, which was intrinsic in these algorithms. And so the main problem was to exploit the parallelism, rather than to introduce it in the first place.

A general form of the equation we need frequently to solve is

$$\nabla^2 \phi(x, y, z, t) = f(x, y, z, t)$$

subject to some boundary conditions. Such equations are of central importance in physics, as for example the Laplace, Poisson or Schrödinger equations.

To simplify the discussion somewhat, let us consider for the moment only the linear second-order PDE that does not depend upon time

$$c_1 \frac{\partial^2 \phi}{\partial x^2} + c_2 \frac{\partial^2 \phi}{\partial y^2} + c_3 \frac{\partial \phi}{\partial x} + c_4 \frac{\partial \phi}{\partial y} + c_5 \phi = f(x, y)$$

where the coefficients $c$ are functions of the Cartesian coordinates $(x, y)$ within some boundary on which the values of the potential $\phi$ are known. Solutions to this equation are derived by the very well-known technique of approximating the derivatives by differences, and then relating the potential at any given node $(ij)$ of a regular Cartesian mesh (as in Fig. 19) to that at the four neighbouring nodes by an equation of the form

$$\phi_{i,j} = \rho_{ij} + a_1 \phi_{i-1,j} + a_2 \phi_{i+1,j} + a_3 \phi_{i,j-1} + a_4 \phi_{i,j+1}$$

In this equation, all the coefficients $a$ and $\rho$ depend on the node $(ij)$ and on the values of $c$ and $f$ at the node. Clearly there are as many linear equations of this type as there are mesh points within (and on) the boundary $B$. 
There is a whole class of procedures for solving these equations by successive approximations. In the simplest of these, the left hand side of the above equation is taken to be the “new” value of $\phi_i$ at the n-th stage of iteration for the mesh point $(i,j)$, and is calculated from the four “old” values of $\phi^{n-1}$ at the neighbouring points. The starting values of $\phi$ are estimated, and the successive values at all the mesh points may be calculated in parallel, since they are independent. The process terminates when convergence of successive approximations has been obtained.

If all the new values of $\phi$ are computed concurrently at each mesh point (the point Jacobi method) then the parallelism of computation is maximized, being equal to the number of mesh points. Unfortunately, this is nearly always the worst procedure since the rate of convergence for many forms of PDE is impossibly slow. What is usually done instead on serial computers is to use a linear combination of old values $\phi^{n-1}$ and new values $\phi^n$ to derive a better “new” value. Depending upon whether this is done with successive points or points on successive lines, the methods are described correspondingly as “point successive over-relaxation” (SOR) or “successive line over-relaxation” (SLOR). There are also various ways of forming the necessary linear combination of “old” and “new” values, one of the best of which is to take alternate points in a raster scan of the mesh (odd-even or “red-black” ordering) and to adjust only the even values first and then the odd, each time taking a different form of linear combination; this is known as the point SOR method with Chebyshev acceleration.

We can see that there is a conflict here between rate of convergence and parallelism. The simplest method, with maximum parallelism, is actually by far the slowest converging. The better, point SOR, method has much faster convergence (as much as 200 times faster for a 128 X 128 mesh and reasonable final error), but it is unfortunately essentially sequential, since each new value of $\phi_{ij}$ is a combination of old and new values.

Fortunately the SLOR method with Chebyshev acceleration does introduce parallelism, since the ordering of mesh points in alternate lines allows each stage (each half-iteration) to be based on computations involving only “old” values. There are several other methods, either with different ordering of mesh points or with different acceleration procedures, and each is characterized by a different balance between convergence and parallelism. There are also other techniques, in which the difference equations used to approximate the PDE are taken along the x and y directions separately and solutions are obtained by iteratively solving a set of tridiagonal systems of equations. These methods are termed alternating direction implicit (ADI), and the important point here is that the separate tridiagonal systems (one for each horizontal or vertical line of the mesh) can be solved in parallel.
We see that there is a variety of methods involving parallelism for the solution of PDEs, each with its own balance of choice between parallelism and rate of convergence. Moreover, we need to take into account the data structure, when choosing the most appropriate algorithm for a given machine architecture. The ADI method may be not very suitable, for example, where arrays cannot be accessed except with a stride of unity.

In this short discussion of PDE solution, we have mentioned only iterative methods. For certain types of equation however, particularly the Poisson and other elliptic equations, there are available a number of powerful "direct" methods, mainly by transform techniques. These are mainly of a high degree of parallelism, and much work has been done to implement these algorithms on parallel computers.

6.2 Recurrences

A very important class of calculation arises in recurrence relations, where we need to solve a set of equations whose left-hand sides depend upon values of parameters calculated at the previous stage. Such recurrences occur frequently in matrix operations, and are the basis of many methods of function evaluation. Predictor-corrector methods in numerical integration also involve recurrence relations.

The most frequently met recurrence is linear and first order so let us consider these briefly from the point of view of parallelism.

6.2.1 First-order linear recurrences

The most general form of such recurrences is

\[ x_i = a_i x_{i-1} + b_i \quad i = 1, 2, ..., n \]

with some starting value for \( x_0 \), and all the coefficients \( a \) and \( b \) known. There are thus \( n \) such equations, and it is usual to take \( x_0 = 0 \). Typical of such a recurrence is the Horner expression for polynomial evaluation. In the form we have written it, the above expression is essentially sequential, since each \( x_-, \) cannot be evaluated until is known. We would thus require \( 2n \) operations to compute \( x_n \). As we have seen earlier in Section 4.4, however, there is something we can do to introduce parallelism into the calculation.

In the specific case of polynomial evaluation, one tactic that has been successfully employed is to divide the polynomial of degree \( n \)

\[ P_n(x) = a_0 + a_1 x + a_2 x^2 + ... + a_n x^n \]

into \( q \) segments, each consisting of a subpolynomial \( Q_i \) multiplied by an appropriate power of \( x \). Thus:

\[ P_n(x) = \sum_{i=1}^{q-1} Q_i(x) x^{m_i} + P_m(x) \]

each of the \( Q_i \) are of fixed degree and each successive \( m_i \) is increased by that degree for successive values of \( i \). \( P_m \) represents the remaining low-order terms, and can be evaluated in parallel with all the \( Q_i \).

The idea is to evaluate all the terms of the summation in parallel, in a computational tree, while \( P_m \) is also evaluated in parallel in a separate tree. At the last operation the \( \sum \) and the \( P_m \) are summed together.

To illustrate the method, Fig. 20 shows the computation tree for evaluation of a polynomial of degree 36, using decomposition into two \( Q \) segments and one \( P_m \); \( m_1 \) and \( m_2 \) are thus 21 and 29 respectively. The process is shown at the fifth step in an 8-level tree. \( P_m \) takes 7 steps to evaluate, while \( Q_1 \) and \( Q_2 \) each take 5; the powers of \( x \) are computed in parallel as well, prior to step 6. Clearly the trick here is to choose the degree of \( P_m \) such that it can be evaluated in \( s + 2 \) steps, while the polynomials of lower degree take \( s \) steps. This can be done in a variety of ways systematically, both for the case of a fixed number of processors and for an unlimited number. Some of these methods come very close to the theoretical lower bound of \( \lceil \log_2 n \rceil + 1 \) steps for evaluating an \( n \)-th degree polynomial.

The technique just outlined is of course valid mainly for polynomials. However a similar folding method does exist for the general linear recurrence equation, and the essence of the method is again to collect several terms of the recurrence sequence together. In one example of this method, successive terms are taken together to give a new recurrence relationship between alternate terms, but this time with a new set of coefficients, the new coefficients being related to the original ones.
Thus, if the original recurrence at step \( i \) is of the form (1), then the preceding step is \( x_{i-1} = a_{i-1} x_{i-2} + b_{i-1} \), and the two equations combine to give

\[
x_i = (a_i a_{i-1}) x_{i-2} + (a_i b_{i-1} + b_i)
\]

which is clearly a linear recurrence analogous to (1). Now, however, \( x_i \) can be computed from \( x_{i-2} \) and does not have to await evaluation of \( x_{i-1} \); all the coefficients are of course known beforehand, and the necessary combinations of them to form the new coefficients [within brackets in (2)] may be computed separately, and also in parallel. The parallelism is introduced not only in computing "new" coefficients, but also in evaluating \( x_i \) and \( x_{i-1} \) concurrently.

To illustrate the process, Fig. 21 shows the computational scheme for evaluation of the first six terms of the general linear recurrence. The necessary input coefficients to the multipliers at levels 3 and 5, which are products of the original coefficients \( a_i \), are all computed concurrently in a separate computational tree. Clearly, these products can all be computed in time. It is evident also that the tree, shown in Fig. 21 for only 6 values of \( x \), can readily be extended to higher degree.

The technique just illustrated is known as cyclic reduction, and it is surprising just how much parallelism it can introduce into algorithms hitherto considered essentially sequential. This technique is used in a more general form, termed recursive doubling, a method particularly appropriate in solving tridiagonal systems by Gaussian elimination.

6.2.2 Non-linear recurrences

Despite the great success in exploiting parallelism in evaluation of linear recurrences, the problem of extending this to the case of non-linear systems is much more difficult. Indeed it appears that algebraic methods of transforming the recurrence, such as we discussed in the linear case, can only speed up computation at most by some constant factor (and then only under certain conditions). This is regrettable, since there are many non-linear recurrences which occur frequently in numerical computation. For example, there is the well-known square-root extractor (based on Newton-Raphson):

\[
x_{n+1} = \frac{x_n^2 + S}{2x_n}
\]

which is a second-degree recurrence.

One suggestion that has been made is to use non-algebraic transforms instead, so as to obtain a linear recurrence, which can then be evaluated in parallel. For example, in the above, one might use the transcendental transform

\[
y_n = \text{arc coth} \left( \frac{x}{S} \right)
\]

But this is not particularly useful since we just shift the problem to that of computing the transform! There is also the important question of convergence and of errors introduced by the transform. It needs to be said, however, that the approach is an interesting one which might, for certain particular cases of non-linear recurrences, lead to speed-up in computation.

One very important non-linear recurrence is that arising in the forward elimination phase in the Gaussian algorithm for solving a set of linear algebraic equations. Fortunately, this recurrence, which involves division, can be transformed algebraically quite simply, resulting in a linear (but second-order) recurrence, which can then be evaluated using methods such as cyclic reduction.

6.3 Linear Numerical Systems

No discussion of parallelism can proceed without including a mention of matrix operations and algorithms for solving linear systems of the type \( Ax = b \), where \( A \) is some non-singular matrix. There is no place here for any extensive discussion, and we will not differentiate between different matrix structures. Nevertheless, it is most important to recognize that both the algorithms and the data structures may need to be chosen with much greater care when exploiting parallelism, than when computing on a serial machine. And the techniques that are best for dense matrices may by no means be optimal for sparse ones.

Concerning the elementary matrix and vector operations, addition and multiplication, it is very evident that we can profit by parallelism. Two matrices can be added in a single cycle with \( n^2 \) parallelism, for example, provided the elements of the full matrices are accessible as a contiguous structure. We have also seen earlier the very effective exploitation of parallelism in the case of scalar products and of matrix multiplication, and we recall that the best parallel
Fig. 20 Computation tree for evaluating polynomial of degree 36

Fig. 21 Six-level computational tree for evaluation of linear recurrence
methods to use may differ depending upon whether the product is performed with operands used only once or repeatedly. In the case of sparse matrices in particular, if they are to be used frequently, then it may be very useful to index the arrays (as can be done easily on STAR-100 and its successor the Cyber 205) or to gather the arrays into a more compact structure, with an associated control bit-vector. Structured sparse matrices are particularly easy to store in more compact form, as are large banded matrices.

On the question of linear systems of equations, there are two basic approaches: direct methods and iterative. Among the direct methods, the most usual are first to factorize the matrix \( A \) so as to obtain a triangular system of equations, and then to solve this triangular system. Two well-known factorizations are \( PA = LU \) and \( QA = R \), where \( L \) is a lower triangular matrix and \( U \) and \( R \) are upper triangular; \( P \) is a permutation matrix and \( Q \) an orthogonal matrix.

Gaussian elimination is frequently used in the case of dense systems, with or without pivoting. There are two phases. In the first, the transformation of the augmented matrix proceeds, until an upper triangular form is obtained, while the second phase—the substitution phase—derives the solution by uncoupling the variables \( x \) one by one. Parallelism can be used in all the steps of Gaussian elimination.

Pivoting is frequently an intermediate step in Gaussian elimination. It is interesting to note that parallelism can be used here in two ways: first to find the largest element, or pivot, and then to interchange rows (or columns). In the first aspect, nearly all vector-processing machines can find the largest element of an array quickly and with a single instruction. For the vector interchange, gather and subsequent scatter operations are very powerful ways of implementing the vector moves in the matrix.

The decomposition or factorization phase turns out to be expressable as two recurrences. We have already mentioned earlier that the first of these is a non-linear recurrence, but one which can be transformed into a linear recurrence of second order. The second recurrence is linear and of first-order. Thus, these recurrences may both be evaluated by parallel techniques of the kind mentioned in Section 6.2.1, particularly cyclic reduction.

The substitution phase, using the matrix in upper semi-triangular form, is recursive and so there is no problem in employing parallelism. This can be done, however, in a number of ways, and the best choice will depend upon the problem. In general, the choice will be dictated by whether one or many sets of equations are to be solved using the same \( A \), and whether the matrix is stored in row-like or column-like storage.

To illustrate the point, let us look at the final equations to be solved, after decomposition, for the simple case of \( n = 5 \). We have

\[
\begin{align*}
x_5 &= 1/a_{55} (b_5) \\
x_4 &= 1/a_{44} (b_4 - a_{45} x_5) \\
x_3 &= 1/a_{33} (b_3 - a_{35} x_5 - a_{34} x_4) \\
x_2 &= 1/a_{23} (b_2 - a_{25} x_5 - a_{24} x_4 - a_{23} x_3) \\
x_1 &= 1/a_{11} (b_1 - a_{15} x_5 - a_{14} x_4 - a_{13} x_3 - a_{12} x_2)
\end{align*}
\]

Here, for row-like storage, we can see the advantage there is to proceed by using the value of \( x \) at each stage (starting with \( x_5 \)) and substituting this value immediately in all succeeding equations with a single vector multiply and subtraction. We have here indeed a linked triad, precisely the form so suitable for some pipeline machines.

The significance of this very simple example should not be overlooked; it is that the data structure for stored arrays is not only important, but the associated algorithm must be chosen appropriately.

So far as iterative methods are concerned, there are many. But the basic techniques are based on decomposing \( A \) into \( M - N \), and then to find \( x \) from \( Mx = Nx + b \) in a recursive manner. Starting with approximations for \( x^{(0)} \), we repeat the solution of the recursive equation \( Mx^{(n+1)} = Nx^{(n)} + b \), until the solution converges within the appropriate norm. Of the two main ways of decomposing \( A \), the Jacobi scheme uses \( M = A_D \) and \( N = -(A_L + A_U) \), where \( A \) can be separated thus

\[
A = A_D + A_L + A_U;
\]

\( A_D \) is the diagonal part of \( A \), and \( A_L, A_U \), the lower and upper triangular portions. The other method, the Gauss-Seidel (or its near relative the SOR we have already mentioned), uses \( M = A_D + A_L \) and \( N = -A_U \).
We have no space here to discuss either of these algorithms, save to say that, just as in the discussion of PDE solutions, both the Jacobi and the SOR methods can be stated in parallel terms, with considerable saving in computing time. However, the Jacobi method converges only very slowly and is not to be recommended. The SOR scheme, since it replaces values of \( x \) as soon as they are computed, and hence bases further calculations on a mixture of old and new values, converges very much more rapidly.

We do not discuss here which methods, direct or iterative, are best for which kind of matrix, and the reader is referred to the now extensive literature on the subject in the context of parallelism. There is a useful discussion of this in Ref. 47. In general, one does not use SOR or Jacobi schemes on full systems of equations, but they are useful for structured sparse matrices.

### 6.4 Ordinary Differential Equations

Considering for the moment only initial value problems, it may be thought that there is little scope for parallelism in the solution of ordinary differential equations (ODE), since the usual integration methods are essentially sequential. Nevertheless, as early as 1964, Nievergelt showed\(^48\) that there was one way that this might be done, at least in principle. He considered the problem

\[
y' = f(x, y) \quad \text{for} \quad a \leq x \leq b, \quad \text{with} \quad y(a) = c
\]

and proposed that, if the interval \([a, b]\) were divided into \(N\) equal sub-intervals, then the \(N\) integrations for each of these could be solved in parallel, provided that there is a predicted value of the solution \(y(x)\) for each of these sub-intervals. The solutions obtained at the ends of each sub-interval will not, in general, correspond to the values at the beginnings of the successive sub-intervals, so what is done is to choose a close range of values of \(y\) about each predicted value (at each sub-interval boundary), and then to "connect" the actual values at the ends of the sub-interval integrations with the starting values at the beginning of the next sub-interval by linear interpolation; the process must of course start at the first sub-interval and then proceed in sequence.

The method is not particularly effective, even for linear equations, where the interpolation can be performed accurately, but it does give a considerable speed-up in computation time if one accepts the substantial amount of extra processing involved. For non-linear equations, the interpolation procedure involves a larger error and a much greater amount of computation.

Much more useful are parallel algorithm versions\(^49\) of serial predictor-corrector methods. Just as an illustration of this approach, let us consider the initial-value problem

\[
y' = f(x, y) \quad \text{for} \quad x > 0 \quad \text{with} \quad y(0) = y_0
\]

In the usual serial method, we lay down a mesh of increment \(h\) with \(x_n = (n-1)h\) for \(n = 1, 2, \ldots\) and \(y_n\) is an approximation to \(y\) at \(x_n\). Then we use predictor-corrector equations of the following form (there are others):

\[
\begin{align*}
y_n^{p+1} &= y_n^c + \frac{h}{2} [f_n^c - f_{n-1}^c] \\
y_n^{c+1} &= y_n^c + (h/2) [f_{n+1}^c + f_n^c]
\end{align*}
\]

where \(y_n^c\) and \(y_n^p\) denote predicted and corrected values of \(y_n\), while \(f_n^c\) and \(f_n^p\) represent \(f(x_n, y_n^c)\) and \(f(x_n, y_n^p)\) respectively.

The sequence of computations is shown in Fig. 22(a), where the upper line represents the process for \(y_n^c\) and \(f_n^c\), with the lower line for \(y_n^p\) and \(f_n^p\). The sequence is \(\rightarrow y_{n+1}^c \rightarrow f_{n+1}^c \rightarrow y_{n+1}^p \rightarrow f_{n+1}^p \rightarrow \ldots\), and the computational front is indicated by the dotted line. Since the computation of the \((n+1)\)th mesh point depends upon calculations at preceding points, it can be seen that the process is essentially sequential.

If however, we choose an alternative pair of predictor-corrector formulae,

\[
\begin{align*}
y_n^{p+1} &= y_n^c + 2hf_n^p \\
y_n^{c+1} &= y_n^c + (h/2) [f_n^c + f_{n-1}^c]
\end{align*}
\]

then we can divide the computational process into two concurrent parts,

\[
\cdots \rightarrow y_{n+1}^c \rightarrow f_{n+1}^p \rightarrow \cdots \\
\cdots \rightarrow y_n^p \rightarrow f_n^c \rightarrow \cdots
\]
which can be processed in parallel, since the computational front is now skewed. This is shown schematically in Fig. 22(b).

![Fig. 22 Schemes for (a) serial and (b) parallel predictor-corrector sequences](image)

This parallelization of the predictor-corrector method, illustrated here only for the simplest example, has been extended to the case of many processors and also to algorithms such as Runge-Kutta.

Boundary value problems have also been tackled to see if parallelism can be exploited, and there has been some degree of success.

6.5 Other numerical procedures

There are many other numerical problems which we could have included in this section, to show that parallelism can be exploited in nearly all cases by suitable choice of data structure and associated algorithm. For example, we could have mentioned the evaluation of arithmetic expressions, which, as we saw earlier, can be tackled by reduction of the height of the relevant computational tree. Another area is root finding, where a number of techniques have been developed to introduce parallelism. Principal among these is to find \( m \) roots concurrently using \( m \) processors, but there is an alternative method which multiplexes the search for a single root. In this method, an iterative sequence of vectors is generated, each component of which is supposed to converge to the root in question, the computation being divided among as many processors as there are components.

One area where parallelism has been used on a very large scale is that of transforms, particularly the Fourier transform. The discrete Fourier transform of a vector \( \mathbf{a} \) with \( n \) components \((a_1, a_2, ..., a_n)\) is another \( n \)-vector \( \mathbf{b} \), where

\[
b_i = \sum_{j=0}^{n-1} \omega^j a_j \quad \text{for} \ 0 \leq i \leq n-1
\]

and \( \omega \) is the principal \( n \)-th root of unity, and all arithmetic is with complex numbers.
Since the transform involves the matrix-vector multiplication $b = Fa$, we can see that parallelism of $n^2$ can be used to evaluate $b$ (in about $2 \log_2 n$ steps). There are several different versions of the FFT algorithm but the Cooley-Tukey algorithm for FFT can be implemented with fewer processors than for most other algorithms.

There is an excellent survey of the use of parallelism in transforms in Hockney’s book. A most important point that is brought out in that survey is the importance, once again, of data structures. The problems arising in data flow are just as great here as in the arithmetic, and to find the optimal solution is not easy for any given computer architecture.
So far we have discussed the parallel properties of certain machines, both at the macro and micro level, and the algorithmic aspects of some numerical processes which can be expressed so as to take advantage of machine parallelism. What we have not done is to say anything about how programs might be written so as to implement a parallel algorithm on a real machine. Moreover, we have stressed the absolutely central importance in this process of having the right data structure, and we have not mentioned any way that this can be achieved.

Of course, the problem might be solved in principle by just writing all the necessary commands in Assembly language for a particular machine, and this could be used to define the associated data structures. Indeed, this is precisely what is done for certain very demanding tasks of a fairly static nature. But what we seek is some way to use a high-level approach, and this is what we shall discuss now.

We have to acknowledge right away that there is, at the present time, no standard language in which to express parallelism at high level. Moreover, even for a specific parallel machine, there is really no adequate single language which will do what is required at high level. In practice, what is done is to use a mixture of some standard high-level language (usually FORTRAN), special sub-routine calls, and a whole range of conventional procedures which are known to work, and are usually embodied as a set of extensions to the standard language.

7.1 The FORTRAN problem

The first problem one encounters in parallel processing is how to express arrays, both as entities with which to operate and as objects to store or access in some particular way. The first does not depend upon the particular machine, although specific properties of the machine may have a profound effect on the efficiency of the processing. The second aspect is most certainly highly machine dependent, as we have seen in the discussion of memory segmentation and interleaving.

The basic problem which arises here with FORTRAN is that there is no choice of data structure, since all arrays are stored in a single prescribed order, namely column major order, and there is no way in the language to manipulate whole arrays or operate with them except on an element by element basis.

The second main problem is that there is no way explicitly to stipulate parallelism in FORTRAN, although there may be concurrency implicit in the code.

To solve the problems just indicated, but still not to abandon FORTRAN as the principal vehicle for program expression, there have been many approaches. They are, however, mainly of two categories: either one leaves it to a special compiler to try to extract what parallelism is implicit in the source code (this is called "vectorizing"), or one puts at the disposal of the programmer a whole range of auxiliary features in the language to enable parallelism to be expressed explicitly.

There is of course the third possibility, of a new language for parallel processing, which may even retain vestiges of FORTRAN, and we will come to this later. Let us consider, however, the first two main approaches, namely those which seek to preserve FORTRAN.

7.2 Vectorization

Let us start by considering the very simple example of arithmetic expressions. For example, the statements

\[
X = (A + B) \times (C - D) \\
Y = (E - F) / (G + H) \\
Z = X + Y
\]

evidently imply parallelism and an order of execution, and it is clear that a compiler could produce optimal object code to exploit parallelism in a machine. Obviously the first two statements here can be evaluated concurrently, while the third statement requires to wait. This is a typical example of an area that can be tackled quite systematically by tree-height reduction techniques of the kind we have already discussed.

Another area is that of mathematical functions, such as \( Y = \text{ATAN}(X) \). Since the actual code used for execution will have been written once and for all, taking into account all the parallel features of the machine, it is evident that the routine can be "vectorized" to any extent allowed by the algorithm and the machine architecture. For example, many trigonometric functions employ polynomial evaluation, a task certainly amenable to vectorization. And, indeed, there is no reason here why the algorithm chosen should not match the machine architecture in the optimal fashion.
Difficulties arise as soon as we have subscripted arrays, and particularly in iterative loops. Consider the simple DO loop

\[
\text{DO } I = 1,100 \\
A(I) = A(I + 1) + B(I) \\
\text{CONTINUE}
\]

Here, whether the loop is evaluated sequentially or whether all the 100 arithmetic statements are processed with 100-fold parallelism, the final values stored in the array \( A \) will be the same in both cases. This is simply because the left-hand side of the statement requires only vector elements that have been computed previous to entry into the loop.

On the other hand the following loop

\[
\text{DO } I = 1,100 \\
A(I) = A(I-1) + B(I) \\
\text{CONTINUE}
\]

would produce quite different results depending upon whether it were evaluated sequentially or in parallel [except for \( A(1) \)]. This is because the arithmetic statement is recursive and because there is a “data dependency”; the left-hand side depends upon a value computed in a previous iteration. Thus, the second loop will not “vectorize”, although the first one would. It is evident that, if the arithmetic statement were of the form

\[
A(I) = A(I + L) + B(I)
\]

then the loop could not be compiled in vector form, since the value of \( L \) is not known at compile time. We see here an interesting example of the inadequacy of FORTRAN as a language for parallelism, since the implied ordering of statement execution is sequential. This can be seen also in implied recursive sequences of the following kind

\[
\text{DO } I = 2,3 \\
X(I - 1) = \text{CONST} \\
Y(I) = X(I) \\
\text{CONTINUE}
\]

where the normal execution would form the sequence \( X(1), Y(2), X(2), Y(3) = \text{CONST}, X(2), \text{CONST}, X(3) \), whereas the vectorized sequence would be re-ordered to \( X(1), X(2), Y(2), Y(3) \) and the value of \( Y(2) \) would be \( \text{CONST} \) instead of \( X(2) \).

Another problem that arises is with irregular increments in indexing, or increments which are unknown at compile time. In the following loop

\[
\text{DO } I = 1,100 \\
J = \text{INDEX}(I) \\
A(I) = B(J) \\
\text{CONTINUE}
\]

there is nothing that the compiler can do to execute the assignment statement for \( A \), since it cannot predict which 100 elements of \( B \) to fetch as a vector. We should repeat here also a point made earlier, namely that if \( J \) has an irregular stride, then we would expect an inefficient implementation on machines (such as CRAY) where effective data streaming from memory can only be obtained for regular stride. Indeed, on STAR-100 (or CYBER 205), only unity stride (contiguous indexing) is used for arrays.

Of course something can be done to vectorize the above loop, by noticing that we have in effect a \textit{Gather} as well as an assignment operation, and the two can be separated thus

\[
\text{DO } K = 1,100 \\
J = \text{INDEX}(K) \\
\text{TEMP}(K) = B(J) \\
\text{CONTINUE} \\
\text{DO } I = 1,100 \\
A(I) = \text{TEMP}(I) \\
\text{CONTINUE}
\]
The I loop here vectorizes readily since all the vector elements are in contiguous locations. The K loop is either executed sequentially or, more usually, is replaced by an appropriate subroutine to perform the Gather (e.g. the routine Q8 VGATHR on Cyber 205, or the library routine GATHER on CRAY); whether or not any parallelism can be used to implement the gather will depend upon the actual machine.

In general, any variable element inside a loop will inhibit vectorization. Here, a computed branch is one example, and the presence of variable functions (such as SQRT) is another. There is little that can be done by a compiler to handle branches, although there is much that the programmer can do to avoid them! In the case of mathematical and other variable functions, what is usually done is to have a set of special "vectorizable" routines. We note here, incidentally, that there are broadly two different ways of vectorizing built-in routines: in the first, the routine is just coded for scalar input arguments but makes use of any machine parallelism available; the second possibility is to re-code the whole routine to accept an array of input arguments, returning a corresponding array of results. The latter approach is rather rare.

The preceding brief discussion shows that vectorization of source code is a possible process, but one where there are many dangers. It is clearly a process that benefits greatly from a re-structuring of code, so as to take into account any specific ambiguities implicit in the use of FORTRAN and to take note of particular machine features.

7.3 Special routines and procedures

We have already mentioned routines which enable users to perform certain parallel operations at high level, particularly those which manipulate arrays such as scatter and gather. It has to be said however that the principal way that the difficulties implicit in the use of FORTRAN are overcome is by the use of library routines, of which there are several hundred for any of the large parallel-processing machines. These routines perform a wide range of functions, from manipulation of arrays and data structures to complicated processing such as FFT operations. It is usual also to have vector routines for operations such as scalar product of two vectors, or finding the maximum element of a vector.

In addition to special routines, called either explicitly by the programmer or invoked implicitly by the vectorizing compiler, there is sometimes a need for certain procedures which take into account specific machine features. One such procedure has already been mentioned, namely to arrange vectors so that successive elements to be used take into account the stride that can be used without creating memory access difficulties. Another example of this is the ROWWISE statement, used on Cyber 205 rather like DIMENSION to define a non-standard FORTRAN ordering for contiguous elements of an array.

On some machines, we need to take into account that there are only a small number of parallel processors (or vector registers) available, so that the natural degree of parallelism is limited. For example, the CRAY computer handles 64-element vectors as a natural unit. It can be a considerable advantage, therefore, sometimes to break up long sequences of operations into a succession of shorter ones which reflect the parallelism of the machine to be used. For example, long DO loops can be rewritten as a succession of shorter ones.

7.4 Introducing parallelism into the programming language

From the preceding discussion we have seen that a number of properties we would like to have in order to exploit parallelism are just not contained in current programming languages in widespread use, although these properties can be added by techniques such as subroutine calls. The question arises whether we can, nonetheless, introduce such properties into the language itself, and it turns out that there have been many attempts to do this. The principal areas where efforts have been made are handling whole arrays, defining and modifying data structures, and introducing commands to define parallelism explicitly by language constructs. And, of course, much effort has gone into the introduction of macro vector operations into the language, such as inner product, FFT, and so forth.

One of the very first programming languages not just to allow parallelism but to have the possibility of handling whole arrays was APL, in 1962. In APL, arrays are not subscripted, although the language does allow the isolation of a single element as an operand. Instead, arrays of any dimension and rank are represented by a single variable which may serve as the operand in a wide range of functions; the functions are either monadic or dyadic. The basis of array representation in APL is that they are conformable (the same number of elements within each subscript range). It is a powerful alternative to the subscript representation, but it does give rise to difficulties if one wants to operate together in some way on non-conformable arrays.

There are very firm protagonists of APL (and also antagonists), but the principal difficulty of using APL as a language for the expression of parallelism is that it does not allow this expression to be extended to parallel code segments, but is restricted just to functions of array variables. But it is indeed a powerful language.
Other, more recent, languages, particularly those for specific machines, retain the normal FORTRAN constructs for variables, including the type and dimension declarations, but add specific new data declarations to indicate arrays. An example here is DAP FORTRAN, where vectors and two-dimensional arrays may be declared by omitting the first and first two dimensions respectively. BSP FORTRAN is even more comprehensive and allows not only selection of a whole array and elements of the array, but also specific patterns of elements (for example, all the even elements from a specified column) or even certain vector mappings. Gather and scatter operations are also features of the BSP FORTRAN language, and not just special subroutines.

Concerning parallelism within the language, we have already noted the difficulties with most conventional languages, namely that independent code segments may be difficult to identify as such by vectorization. One of the very first attempts to solve this problem was to allow the programmer himself to identify such segments, and to do this by a new construct of JOIN and FORK. This idea is described by Conway in 1963, but his paper mentions that the notion was already well-known even then (under other names). The use of this construct is shown in the following example, where an inner product is computed in two independent and parallel segments:

```plaintext
FORK FIRSTHALF, LASTHALF

FIRSTHALF begin SUM 1 := 0;
for I := 1 STEP 1 UNTIL N/2 DO
SUM 1 = SUM 1 + (X(I) * Y(I));
GO TO CONTINUE

LASTHALF begin SUM 2 := 0;
for K := N/2 + 1 STEP 1 UNTIL N DO
SUM 2 = SUM 2 + (X(K) * Y(K));
GO TO CONTINUE

CONTINUE JOIN FIRSTHALF, LASTHALF
SUM = SUM1 + SUM2
```

The FORK statement can be extended to the case of an \( n \)-way fork either by \( (n-1) \) forks in a hierarchy or by a single \( n \)-FORK statement (with \( n \) separate addresses). There are also many variants of JOIN with either multi-way attributes or conditional properties.

In general, the features added to languages to introduce parallelism have been machine oriented. DAP FORTRAN is a case in point here, where some functions depend upon the size and shape (64 X 64) of the processor array. But several attempts have been made to define a machine-independent language, the best known of which is VECTRAN. Something very much like VECTRAN is being considered now as an ANSI standard extension to the FORTRAN language.

In VECTRAN it is possible to define and manipulate arrays, sub-arrays and scalars, all by name, and to have associated with the arrays a range (or shape in the ANSI extensions). The arrays may or may not be conformable. Operators or functions can then be used with arrays as operands either on an element-by-element basis, in which case they are termed elemental, or on the whole arrays, by a transformational process. Matrix addition is an example of an elemental operation, matrix multiplication an example of a transformational operation. The equivalent of gather and scatter operations exist and are called PACK and UNPACK, and they are used in conjunction with a control vector of values.

A powerful construct

```plaintext
WHERE (lexpr)
   block 1
OTHERWISE
   block 2
END WHERE
```

allows conditional control to be exercised on the array operations in block 1 and block 2 in accordance with the logical vector lexpr.
One surprising omission from VECTRAN is anything like the FORK-JOIN construct, which appears to be the simplest way for a programmer to tell any compiler where he wants parallelism to be implemented. But the discussions within ANSI are by no means finished yet. At this stage, all that can be said is that at least the ANSI work offers us some degree of transportability, which all the other languages for vector processors do not.

One other language that should be mentioned, in conclusion, is ACTUS\textsuperscript{53)} which was defined and partially implemented initially for the ILLIAC IV. The principal features of the language appropriate to parallel processing are a set of data-declaration statements, allowing whole arrays to be manipulated, and a form of syntax well suited to expression of problems incorporating parallelism. Important attributes of ACTUS are that it is machine independent and Pascal-like, and also that there are explicit statements available for data alignment.
8. THE CRAY AND CYBER 200 MACHINES

Two important machines have not been described in the preceding sections, namely the CRAY-1 and the Cyber 205. They are members of two separate and small series of related machines, manufactured respectively by CRAY Research and by CDC. The reason they have been singled out for particular mention is that they are the only large-scale vector machines that are commercially available at the present time for general purposes. Of the other production examples, the ICL DAP is not stand-alone and requires a 2900 as host, while the FP Systems processor also requires a separate host, but is in any case too specialized in its function.

The first CRAY-1 was delivered in 1976 to Los Alamos, and there are at least 35 others somewhere. The first Cyber 205 went to the UK Meteorological Office, and there are perhaps 3 or 4 others in the course of delivery (or in the course of up-grade from the Cyber 203). The CRAY-1 is a completely new machine, although architectural features can be traced to the 7600. The Cyber 205 derives very directly from the STAR-100, with the Cyber 203 as an intermediate step (of which there were perhaps only 3).

8.1 The CRAY-1

The first thing to remark about the CRAY-1 is that it is, by any standards, a very fast processing engine. The first models had up to 1 Mword of memory with 16-way interleaving and 64-bit words, using a 50 nanosecond cycle time. Neglecting for the moment the vector capability, the CP architecture appears very reminiscent of the 7600, but much faster. It is shown in Fig. 23, and it operates with a cycle time of 12.5 nanoseconds. As can be seen, there is an instruction buffer of four groups of 64 parcels, with an instruction occupying one or two parcels. There are 9 independent functional units to handle scalars (or addresses), there are two sets of registers each, for addresses and for scalar operands (the A, B and S, T register groups respectively), and all operations deal only with operands in the A and S registers, not with memory directly (the B and T registers are temporary storage). Since the operational units are segmented, they can handle successive operands in pipeline fashion, producing results one every clock period, and since the operational units operate independently, they can in principle collectively generate several results every clock period. In practice, there are two limitations on the attainable rate of scalar operations; only a single instruction can be issued per clock period, and the maximum memory bandwidth to registers is only one word per clock period, irrespective of memory interleaving. Thus, the very maximum scalar processing rate is 80 Mflops, but this is a highly impressive figure indeed. It is unlikely to be attained, even with the most carefully written assembler code, since the actual time for single operations usually takes several clock periods and there are unlikely to be very long sequences of the same operation in scalar code. Since floating add or multiply take 6 and 7 cycles respectively, with other units requiring about 3, we might expect a more usual performance on scalar code to be about 25 Mflops, and this is indeed what we observe in practice. It is when we come to array processing that this figure can be increased by an impressive amount.

Vector processing is accomplished in the CRAY-1 by means of the 3 floating-point units and 3 additional vector units, all 6 of which are segmented and can function concurrently. Because they can all produce results every clock period, they need to be supported by a very high operand rate, and this is achieved by 8 sets of 64-word V registers. The limitations on processing speed is not now the rate of instruction issue, since one vector instruction can define a whole series of identical operations on successive operands in the V registers, but the rate of operand flow. In practice, the vector add and multiply units may operate concurrently, but each require two input and one output operand per clock period. Such a rate cannot be sustained by the memory bandwidth to the V registers, which is only 80 Mwords per second. However, results from one vector operation, stored in the V registers, can be used as input operands to another operation. In this way, the memory limitation can be overcome, and much higher processing rates achieved. This process, called “chaining” can also proceed concurrently with loading or storing operands between memory and registers. So if results from, say, the floating multiplier are used as input operands to theadder, we might expect a processing rate as high as two operations in one clock period, namely 160 Mflops. And, indeed, in a hand-written code for inner-product evaluation, exploiting chaining, a rate in excess of 150 Mflops has been attained.

A further feature of importance in the CRAY-1 CP is that all the vector operational units are segmented, so as to enable pipelining irrespective of function. We have already seen how this can be done with addition, multiplication and shifting, but it is something difficult to achieve in a divider, which is generally iterative. What is done in the CRAY-1 machine is to separate the division of two operands S1 and S2 into four segments, and to use a reciprocal approximation (which is a machine instruction). The process of division is then to find the starting approximation A, for 1/S2, to generate an improvement factor $F = (2 - S2 \cdot S3)$ using the other floating-point units, and to perform two multiplications $S1 \cdot A$ and $F \cdot (S1 \cdot A)$. The improvement correction is based simply on a single iteration of the well-known Newton method. Actually only three segments are in sequence, since two of the multiplications can overlap.
Fig. 23 The CRAY-1 CP Architecture
It is these three features of the CRAY-1, the fast vector registers, segmented functional units with parallel operation, and chaining, that make the machine so powerful in array processing. It is impossible here not to quote the lines from Horatius which so aptly describe the position:

\[
\text{And straight against that great array} \\
\text{Forth went the dauntless Three} \\
\text{(Macaulay)}
\]

One further attribute of the CRAY-1 needs to be mentioned as a major factor contributing to fast performance, namely the short start-up time for pipelined units. We saw earlier how a long time can be a detrimental factor in the case of the long pipes in STAR-100. In CRAY-1, results begin to appear in pipelined operations only a few clock periods after instruction issue, only the division process taking much longer than the others. Thus for a series of similar operations, vector performance is already better than successive scalar operations already for vectors of length about 5 elements.

On the negative side, the restricted memory bandwidth of CRAY is a factor that could certainly be improved, and the absence of scatter and gather operations a significant disadvantage. However successive vector elements need not be in contiguous locations, as in the STAR-100, but may be separated by any constant stride. Recently, a newer model has become available, the CRAY-1S with up to 4 Mwords memory and the possibility of several I/O processors. And only very recently indeed, CRAY Research has announced a dual-processor version, the XMP, with certain other important improvements.

The CRAY-XMP is very similar to the 1S architecturally, but has a number of very significant differences. First of all it is a dual processor, each of which may share memory either for data or code, or both. Secondly, the processor cycle time is now 9.5 nanoseconds compared with 12.5 on previous models. By far the most significant improvement is, however, that there are now four memory-access ports per processor compared with only one previously. This was a major bottleneck before, and the increased number of ports now allows concurrency in fetching or storing operands as well as concurrent instruction fetching and I/O. Also the memory bandwidth is increased as well as the degree of interleaving.

All these new features in the XMP, together with a mass solid-state store between memory and disk, and other hardware enhancements, undoubtedly make the machine an extraordinarily powerful parallel-processing computer, possibly the most powerful in existence at the present time.

8.2 The Cyber 205

The principal difference between the Cyber 205 and its predecessor, the STAR-100, is the technology of implementation, allowing not just a very much faster operation but also greater reliability. As we saw earlier, the processing pipes of STAR could operate at very high asymptotic rates, but the memory speed did not really match, and the long start-up time made the processing of short vectors a very ineffective process.

The cycle time for Cyber 205 has been reduced to 20 nanoseconds and there can be up to four independent processing pipes, all of which can operate concurrently and all of which can perform the operations of add, multiply, divide and square root. To support these pipes, the memory can be up to 4 Mwords of 80 nanosecond cycle, but interleaved 16 ways and with the possibility of fetching superwords (512 bits). In the largest configuration, the Cyber 205 memory bandwidth is 16 words in one clock cycle of 20 nanoseconds, or 800 Mwords per second. Thus, the memory bandwidth is not, in general, any obstacle to processing speed (except that vectors need to be with contiguous elements, i.e. with unity stride), and this factor together with the short-stopping of pipes allows indeed a peak processing speed of \(4 \times 50\) Mflops for 64-bit operands, with twice this rate attainable in certain instances of computations on so-called linked triadic groups such as \(A + B \cdot C\).

One of the greatest advantages of the Cyber 205 architecture is the scatter-gather and compress-merge operations, which are hardware supported and can proceed at a speed of almost one element per clock cycle. The greatest disadvantage is the long start-up time for arithmetic in pipes, which is due in part to the absence of intermediate high-speed registers. Unlike the CRAY-1, the Cyber 205 performs arithmetic memory to memory, except for short-stopping, when it can be used. The Cyber 205 start-up time is still about 50 cycles (i.e. about 1 microsecond, compared with about 60 nanoseconds on average in the CRAY-1), so we would expect the Cyber 205 to perform faster than the CRAY only for very long vectors (with length \(> 200\) elements).
9. PARALLELISM IN DATA PROCESSING

When discussing applications of computational parallelism, we have concentrated almost exclusively on numerical algorithms. But there are other aspects which should not be neglected, since they too may benefit from parallelism. One such area is, of course, pattern recognition, particularly geometrical patterns. And indeed, arrays of processors, such as PEPE, STARAN and DAP, are particularly valuable here, and are being exploited. Another area is associative searching of data files, and here also there are a number of machines available with both content addressability and the capability of parallel processing on the basis of concurrent keys.

An important area of computation that has not been mentioned so far in the context of parallelism is data processing, and yet this is the largest component of computing that exists globally. It is an area of particular importance in high-energy physics, where it also represents the major part of the computing done. We may ask, therefore, what can parallelism offer to improve the way we perform data processing. It turns out that something can indeed be done, at least in the case of processing large numbers of discrete events.

One of the obvious things that can be done is to use $n$ separate processors to process $n$ separate events concurrently. There is little virtue, however, in this approach since each event requires (in general) a different computation, and so the different processors need to be under independent control of different code sequences. The only benefit to be obtained here is that the time taken to process events is reduced approximately by the factor $n$, but usually at a cost much increased compared with the effective exploitation of a single, much larger computer.

Another attack on the problem is to process events sequentially, but to try to 'vectorize' the processing code. This has been tried with a number of well-known codes in daily use for high-energy physics data processing, but with little success. The reason is simply that the programs have many variable elements in them (branches, library functions, iterations) and also linear recurrences involving data dependencies; there is just insufficient information in these FORTRAN codes, in general, to extract sufficient concurrency.

A third possibility shows much more promise. This is to take data blocks (the whole or parts of events) which are almost similar, and then to impose regularity on the data structure. An example here might be a set of trajectories in which the number of space coordinates vary, but not greatly; this is typical of HEP experiments. The coordinate number can then be made the same by a variety of statistically valid techniques.

Having imposed a regular data structure (also one that is free from anomalies, "zeroes", "infinities", etc.), one can then form a "regular" code to process the events entirely in parallel and in lock-step. This code is termed invariant, and it can be formed by a systematic re-structuring. It has been shown that this can indeed be done, but only by human means (not by a compiler) at this stage. The key advantage that emerges here is that the inefficiencies introduced by re-structuring of data and code (typically factors of 2 to 3) are far outweighed by the gain from parallelism (typically 64 or more, depending upon the machine). In one example of this technique, net gains of more then 10 have been reported. One of the most important conclusions that emerges from this work is that much larger gains may be expected in processing events concurrently in this way, with the processing code written ab initio bearing in mind eventual parallel processing.

One final point needs to be made here concerning processing time. Of course the bulk processing of data events in parallel will proceed faster on an array processor, using the techniques just outlined. Also, it should be possible in this way to perform the processing cheaper, since it costs much less to replicate processors without control circuitry, program storage etc. than to replicate whole computers. However, in one area, parallelism is the only way to achieve the desired end, and that is in triggering or even full on-line analysis for HEP experiments. When there is just insufficient time between successive interactions in an experiment to make the necessary computations (and hence triggering decisions) the only thing to do is just to store the raw data for subsequent analysis. This is the typical situation in most HEP experiments at the present time. However, sequential computers, even if speeded up by a factor of ten, would just not solve the problem, and the only alternative to continuing to store data is to try to exploit parallelism, since we might gain time factors of two or even more orders of magnitude.
10. THE FUTURE

Despite the relatively long history of parallelism, which extends back over almost the same period as automatic computation itself, we have seen that its exploitation has been very much less than sequential processing. Only in very recent times, perhaps only in the last decade, has there been a significant change towards parallelism, but this has been overwhelmingly for special applications, for which special-purpose computing systems have emerged.

The reason for the present situation is twofold. First of all, the necessary hardware for concurrent processing has hitherto just been too expensive (and also difficult to build with the required speed and reliability). Secondly, the complexity associated with all aspects of parallelism, from algorithms through to programming languages, has been rather too great to be handled by the overwhelming number of computer users.

Despite this picture, we begin to see a change. For the first time in the history of electronic computing, the hardware costs are affordable by nearly all users, and indeed the cost of replication of similar units, precisely what is needed for many kinds of parallelism, has become a particularly easy thing to do, because of the advent of monolithic VLSI technology. At the same time, there is now a growing experience of algorithms and software on which to base concurrent computation; the complexity is beginning to be manageable.

Two important developments have taken place comparatively recently. First the advent of the CRAY-1, which more than any other machine, has become available for general-purpose computation at reasonable cost. Second, the add-on IAP which is offered by Hitachi as a feature of their normally sequentially-processing systems.

Looking to the future, it seems that we are in fact in a transition phase from purely sequential systems, and that parallelism will become a standard feature of most computing systems in the future at the processor level. To exploit this there is, of course, much to do. We need programming languages appropriate to algorithms expressing parallelism, and we need new algorithms and practices. I have no doubt whatsoever that all this, and much more besides, will be achieved over the next one to two decades. Computational parallelism by the end of this period will have become as commonplace as computational sequentialism is now.

Acknowledgement

Figure 9 was reproduced from a listing provided by Dr. H. Lipps at CERN, and Fig. 18 was provided by Roger Hockney. The block diagram of the CRAY processor is a slightly re-worked version of the well-known illustration in an early article on the CRAY machine which appeared in 1978 in Communications of the ACM.

I should like to acknowledge some very helpful conversations I have had with my colleagues at CERN, notably Tor Bloch and Mike Metcalf.
REFERENCES


54) CYBER 200/Model 205 Technical Description (Nov. 1980), publ. CDC Inc.