

A Survey of Some Theoretical Aspects of Multiprocessing



L. BAER

University of Washington, Seattle, Washington 98195

In this paper, several theoretical aspects of multiprocessing are surveyed. First, we look at the language features that help in exploiting parallelism. The additional instructions needed for a multiprocessor architecture; problems, such as mutual exclusion, raised by the concurrent processing of parts of a program; and the extensions to existing high-level languages are examined. The methods for automatic detection of parallelism in current high-level languages are then reviewed both at the inter and intra statement levels. The following part of the paper deals with more theoretical aspects of multiprocessing. Different models for parallel computation such as graph models, Petri nets, parallel flowcharts, and flow graph schemata are introduced. Finally, prediction of performance of multiprocessors either through analysis of models or by simulation is examined. In an appendix, an attempt is made toward the classification of existing multiprocessors.

Key Words and Phrases: multiprocessing, mutual exclusion, semaphores, automatic detection of parallelism, graph models, Petri nets, flow graph schemata, scheduling, array processors, pipe-line computers.

CR Categories: 6.0, 8.1, 4.32, 5.24

INTRODUCTION

In this paper we plan to survey some theoretical aspects of multiprocessing. By definition, *multiprocessing* is the simultaneous processing of two (or more) portions of the same program by two (or more) processing units. Among the latter, the I/O processors are excluded; that is, the overlapping of I/O operations with arithmetic or logical instructions is not considered as multiprocessing in this paper. Multiprocessing must not be confused with *multiprogramming*, which is the time and resource sharing of a computer system by two (or more) programs resident simultaneously in primary memory. Sometimes we shall use the term *parallel processing* for either of the above or a combination of both.

If one considers a von Neumann machine as the model for computer systems, there is no chance of parallel processing; when one module in the machine takes over control, all others stand idle. However, the concept

of the von Neumann machine has evolved, and the first breaches were an overlap of I/O operations with CPU computations (which led to multiprogramming) and look-ahead features. Then, keen interest in systems with multiple CPU's developed.

An increasing number of applications calls for more computing power. A classical example is the weather prediction problem with its stringent deadline requirement. In fact, some computations that should be thought of as parallel (e.g., addition of two vectors) have been transformed into sequential algorithms only because of the restrictions imposed by the architecture of the machines. Thus, multiprocessors seem to be a natural expansion of uniprocessors. Some multiprocessors can be viewed simply as extensions of uniprocessors; that is, different jobs are assigned to different CPU's, but no job may use more than one CPU. Obvious advantages of this design are: speed-up of global throughput; improved reliability with the possibility of graceful degradation; and

CONTENTS

I	Introduction	31-33
II.	Language Features to Exploit Multiprocessing	33-38
	A	Explicit Indication of Parallelism
	B	The Lock-Out Problem
	C	New Programming Language Concepts
III	Automatic Detection of Parallelism	38-46
	A	Detection of Parallelism Within an Arithmetic Expression
	B	Detection of Parallelism Between Statements
	Conditions for Statements Independence	
	Automatic Program Analysis	
	Discovering the Hidden Parallelism	
	Loop Replication	
	Experimental Results	
IV	Models of Parallel Computations	46-57
	A	Generalities Relative to the Modeling of Parallel Computations
	B	A First Look at Directed Graph Models
	Directed Acyclic Bilogic Graphs	
	Petri Nets and Extensions	
	C	Graph Models With Extensive Control
	D	Models Introducing Data Queueing on the Arcs
	E	Parallel Program and Flow Graph Schemata
	F	Other Models and Current Research
V	Prediction of Performance	57-64
	A	Analysis
	Queueing Theory	
	Graph Model Analysis	
	B	Gross Level Simulation
	C	Fine Level Simulation
VI	Conclusion	64-65
	Acknowledgment	
	Bibliography	65-72
	Appendix: Hardware Systems	72-80
	A	Homogeneous Multiprocessors
	Crossbar Switch System	
	Multiple-Bus Connected Systems	
	Time-Shared Bus System	
	B	Nonhomogeneous Multiprocessing Systems
	C	Array Processors
	D	Pipe-Line Computers

Copyright © 1973, Association for Computing Machinery, Inc. General permission to republish, but not for profit, all or part of this material is granted, provided that ACM's copyright notice is given and that reference is made to this publication, to its date of issue, and to the fact that reprinting privileges were granted by permission of the Association for Computing Machinery.

decreased overhead in switching among jobs in a multiprogramming environment.

Also, one has to note that, while in the early days of computing a large percentage of the dollar value of a system resided in the circuits of the arithmetic and logical units, this is no longer true with the advent of integrated circuits and LSI; furthermore, replication of CPU's does not increase the price of a system as significantly as before. These technological advances allow more easily and economically the building of special-purpose computers—or at least units—which lend themselves to (particular) cases of multiprocessing.

We shall start our survey (Section II) by looking first at the necessary changes in the instruction set of multiprocessor machines, and then at the modifications or extensions induced in the high-level languages for programming multiprocessors. Problems of concurrent control and new language developments will be examined. Section III will deal with automatic recognition of parallelism in a program written for a conventional machine. Both the instruction level and the statement level of languages such as FORTRAN and ALGOL-60 will be examined. The next section will be devoted to modeling of computations for parallel processing. Graph models and program schemata have been used widely, and they will be classified according to their structural properties, such as logic, interpretation, queueing possibilities, etc. Section V will survey the uses of models for prediction of performance; we shall distinguish between analytical prediction through queueing or graph theory, a priori scheduling, and full-fledged simulations. Finally, we shall conclude with some remarks on the future of multiprocessing.

In the appendix we have classified multiprocessors into four categories:

- 1) homogeneous multiprocessors; i.e., systems composed of a set of identical CPU's.
- 2) non-homogeneous multiprocessors; i.e., systems for which the arithmetic-logical unit is a set of special-purpose functional units.
- 3) array processors; i.e., systems composed

of a set of identical CPU's (called processing elements), acting synchronously under the control of a common broadcasting unit.

- 4) pipe-line processors; i.e., systems in which the CPU has the capability of performing operations on streams (vectors) of operands.

These terms are used throughout the paper. Our survey covers the literature up to the first half of 1971, with a few references to later papers. An attempt has been made to keep the bibliography up-to-date.

II. LANGUAGE FEATURES TO EXPLOIT MULTIPROCESSING

Since multiprocessing adds a new dimension in the power of a computer system, it must be reflected in the instruction set of the machine. Furthermore, assembly language macros and extensions to high-level languages must exist to allow programmers to ask explicitly for parallel paths if they choose to do so. It is also reasonable to investigate whether totally new programming language concepts ought not be defined to take advantage of the multiprocessing design. In this section we first review proposals and existing features of assembly and high-level languages that permit explicit indication of parallelism. This leads necessarily to the "lock-out" problem; that is, how parallel paths avoid destroying their shared data. We end this section by looking at proposals for new programming languages geared to parallel processing.

A. Explicit indication of parallelism

In some array and pipe-line systems, such as ILLIAC IV, VAMPS, STAR-100, ASC*, the instruction repertoire possesses vector instructions which allow the treatment, in parallel, of ordered sets of data. Here, we are mainly interested in path parallelism, that is, the concurrent execution of two different parts, or of the same part with different data, of the program; this has been called "the problem of flowcharting and programming the actual links" [GILL 58]

* See Appendix.

for a multiprocessor with asynchronous events. The basic concept at the assembly level is that of FORK-JOIN [CONW 63, DENN 66], which is also called SIMU [DREI 58A], SPLIT-ASSEMBLE [LEHM 66], and START-HALT [WIRT 69].

Conway [CONW 63] defines five possible uses of the FORK-JOIN combination. They are:

- | | | |
|------------|---|--|
| FORK A | : | 1. Initiate process at address A. |
| | | 2. Continue current process at next instruction. |
| FORK A,J | : | 1 and 2 above. |
| | | 3. Increment counter at address J. |
| FORK A,J,N | : | 1 and 2 as above. |
| | | 3. Set counter at address J to N. |
| JOIN J | : | 1. Decrement counter at address J. If zero, initiate processing at address J+1; else |
| | | 2. Release the processor executing the JOIN. |
| JOIN J,B | : | 1. As above. |
| | | 2. The original value of J is kept and a branch to B (i.e., a wait) is initiated. |

Application of these instructions for the control of three concurrent processes is shown in Figure 1. These instructions do not allow a path to terminate per se, that is, without coming to a junction point. QUIT in [DENN 66] or TERMINATE in [LEHM 66] play this role.

Equivalent extensions to ALGOL-like languages have been proposed:

DOTOGETHER-HOLD [OPLE 65];
FORK-JOIN-TERMINATE [ANDE 65];
PARBEGIN-PAREND [DIJK 68] AND
[WIRT 66] (for optional forking).

All of these work on a local block basis.

Gosden [GOSD 66] states that most parallelism is to be found in loops and, thus, defines five primitives, some of which will allow him to implement easily PARALLEL FOR statements. They are:

PREP : A Parallel Path Counter, PPC, is initialized ($PPC \leftarrow 1$). (A stack of PPC's is kept in case of nested loops.)
AND(L) : Two-way fork. $PPC \leftarrow PPC + 1$. Process at address L is initiated and the current process is continued at the next instruction.
ALSO(L) : As above but without incrementing the PPC.
JOIN : $PPC \leftarrow PPC - 1$. If $PPC = 0$, the PPC is "popped" and processing continues at the next instruction, else the processor executing the JOIN is released.
IDLE : Terminates a path and releases the processor that was executing it.

Figure 2 shows the realization of a PARALLEL FOR using these instructions.

It is to be noted that this scheme is independent of the number of processors available, and that there is no need to state explicitly the relationship between AND and JOIN.

Similar ideas can be found in PFORTAN [DRAU 67], an extension of FORTRAN IV with the additional possibility of limiting the number of processors executing a loop.

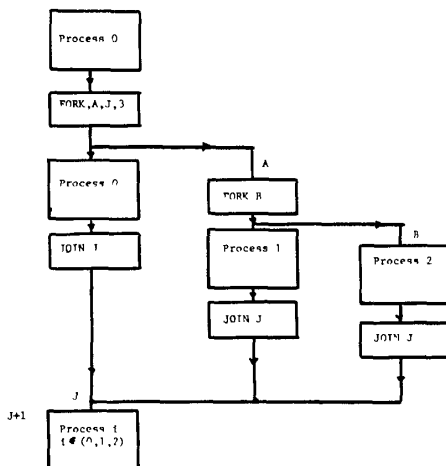


FIG 1 Conway's FORK-JOIN concept

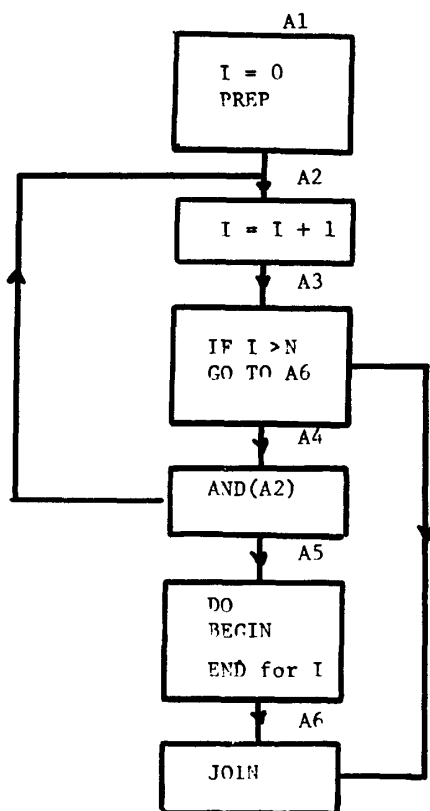


FIG 2. PARALLEL FOR I = 1, N, 1 DO BEGIN ... END

Two general-purpose high-level languages, PL/I [IBM 69] and ALGOL-68 [VANW 69], have incorporated these concepts in their definitions.

In PL/I, forking, called asynchronous processing, is done at the procedure level. A procedure called with a TASK option can be performed in parallel with its called path, and is said to be a subtask attached to the latter. Associated with the parallel subtask is an EVENT option. The joining operation can be programmed in different ways: first, by a RETURN or END statement which gives back control to the calling task; second, by an EXIT (or STOP in the case of the main procedure) which stops the current

task and all its attached subtasks. Finally, a WAIT statement allows suspension of operations in a calling task until all—or only a limited number of—events of attached subtasks (defined through the EVENT option) have terminated.

In ALGOL-68, parallelism can be present both at the procedure level by using a **par** symbol and at the statement/declaration level through the concept of collateral clauses. A simplifying explanation of the latter is to say that whenever a “;” is replaced by a “,” when separating syntactical units, the actions implied by these syntactical units can be done in parallel instead of sequentially. For example in:

```
real x := 1.1, int y := 2, z;
      (x := x + 4.3, z := y);
```

the declarations and initializations of *x*, *y*, and *z* can be performed in parallel; and, after their terminations, the two assignments can in turn be done in parallel.

This collateral clause concept was already present in APL's original definition [IVER 62]. Several assignment statements could be written on the same line, the respective left and right parts being separated by commas. Because of APL's orientation toward on-line computing, this idea has been abandoned. Parallelism is still present in APL in vector and matrix operations, which makes the language particularly attractive for pipe-line machines.

B. The Lock-Out Problem

When splitting a task in parallel paths, it may happen that two (or more) of these paths want to access the same data. To be consistent, there must be some means of preventing unordered changes in the shared data base, if it is to be updated by at least one of the concurrent processes. This has been referred to as the “lock-out” or “mutual exclusion” problem. The portion of code—in a path—that accesses the shared data is called the “critical section” of that path [DIJK 68].

One way to provide the necessary protection is by having instructions of the form TEST AND WAIT AND SET [LEHM 66]

or the equivalent pair LOCK *w*/UNLOCK *w* [DENN 66], where *w* is a one-bit switch. The interpretation is as follows: If one attempts to execute LOCK *w* when *w* is already set, the LOCK *w* instruction is repeated until *w* is reset; when *w* is (or becomes) reset, LOCK *w* sets *w* and the program path containing the instruction may continue. When UNLOCK *w* is reached, the switch *w* is again set to off.

For high-level languages, similar schemes have been proposed. OBTAIN and RELEASE statements [ANDE 65] provide the same functions as the LOCK/UNLOCK pair; a new procedure type, called SHARED, would surround all blocks representing the portions of parallel programs which should not be run concurrently [WIRT 66]. In PFORTRAN [DRAU 67] variables can be either PUBLIC or PRIVATE, therefore allowing the programmer to protect the integrity of his program.

The synchronization of concurrent processes cycling through their critical sections has been thoroughly studied by Dijkstra [DIJK 65, DIJK 68]. The requirements he sets forth are:

- 1) At any given time only one process is in its critical section.
- 2) Stopping one process outside its critical section has no effect on other processes.
- 3) The decision as to which process enters its critical section cannot be postponed indefinitely.

The general solution to the problem can be found in [DIJK 68] where a number of faulty examples are worked out until a satisfactory result is obtained. Only logical and integer variables are used; there is no extension to the usual statements of high-level languages. However, the solution for *n* processors has a flaw; namely, while requirement 3) above assumes that some processor will eventually enter its critical section, it may happen that a particular processor is locked out forever. Knuth [KNUT 66] re-states that requirement as:

- 3) Every process wanting to enter its critical section will be eventually allowed to do so.

Furthermore, he gives an algorithm to solve this modified problem.

Aside from the difficulty of the programmed solution (a more formal solution is now available [GILB 72]), the main drawback to the above approaches is that processes waiting to enter their critical sections are busy setting flags and accessing common variables, thereby slowing the whole system. Primitive operations were, therefore, defined to circumvent this waste of processor time. Elementary primitive operations similar in nature to the LOCK/UNLOCK pair, but without requirement of processor looping were defined by Saltzer [SALT 66] and Lamson [LAMP 68], namely:

wakeup (p): If processor p is logically blocked (i.e., dormant), change its state to active;
 Else set up a wake-up waiting switch (wws) to remember the wake-up call.

block (p): If p 's wws is set, reset it;
 Else change p 's state to dormant.

It is to be noted that many processes may be blocked waiting for the same condition; therefore, a wakeup list may have to be kept and updated, forcing processes to be cognizant of each other. To allow programs to be more independent, Dijkstra defines a new type of variables, called semaphores (which can take only non-negative integer values), and two operations on them, V and P , defined as follows:

$V(S) : S := S + 1;$

$P(S) : L : \text{If } S = 0$

then go to L else $S := S - 1;$

with the understanding that these ALGOL-like statements are indivisible and may not be executed simultaneously.

In fact, it can be shown [DIJK 68] that binary semaphores are sufficient at the expense of more complexity in the code. Figure 3 shows an example of the use of semaphores for a simple buffering routine.

This concept has been now widely accepted. For example, semaphores have been incorporated as a feature of ALGOL-68, and there have been proposals [WIRT 69] for hardware P and V . However, it is to be

```

"begin integer number of queueing portions,
  number of queueing portions = 0,
  parbegin
    producer
      begin
        again 1 produce the next portion,
          add portion to buffer,
            V(number of queueing portions),
              goto again 1
          end;
        consumer
          again 2 P(number of queueing portions),
            take portion from buffer,
              process portion taken,
                goto again 2
            end
          end
        end"
  end

```

FIG. 3. Example of semaphores usage.

noted that the indivisibility of these operations has been criticized. Using a model similar to that used in sequential circuits theory, Bredt [BRED 70A] gives procedures to design a hardware lock-out mechanism free of any assumption about this indivisibility.

C. New Programming Language Concepts

As early as 1962, foreseeing the advent and generalization of multiprocessors, Brown [BROW 62] proposed a method "free, at least above a certain minimal level, from the kind of overspecification of sequence constraints that characterize present programming." This idea has recently been reconsidered. We here report two different approaches to it. In the first, [TESL 68], a class of "single assignment" languages is presented and depicted with a particular example, COMPEL (for Compute Parallel). The main characteristics of this class of languages are that no variables are assigned values by more than one statement, and that there is no possibility of side effects. This allows the compiler to detect easily which variables are dependent on previously assigned ones and, thus, to find the parallelism inherent in the program. (We shall return to this problem in the next section.) COMPEL processes only three types of quantities: numbers, lists, and functions. COMPEL also has only one type of statement: the assignment statement. Because of this latter requirement, there must exist a sizable number of built-in functions. For example, an ALGOL **for**-loop such as:

```

s:=0; r:=r0;
for i:=1 step 1 until n do
begin s:= s + a[i] * r;
      r:= r + 2 end;
r1:=r;
will be replaced in COMPEL by
r: each[r0 by 2 for n];
s: + list of (each a) * r;
r1: last(list of r);

```

The replacement may look rather artificial and clumsy to a programmer; but, as the authors of the paper point out, it may look as natural as FORTRAN to a novice. Another advantage of such an approach is that incremental compilation is readily feasible since the ordering of the statements is immaterial. Unhappily, this aspect of the language makes it unreadable and undocumentable. Further development and extensions of the single assignment concept can be found in [CHAM 71].

A less revolutionary approach has been taken by Volansky [VOLA 70]. His proposal for a new language is an outgrowth of a graph model of computations used at UCLA, that will be discussed in detail in Section IV. In ordinary high-level languages, data is defined separately from computation and sequencing. Volansky's claim is that the latter should also be separated since languages for parallel processing involve much more control than those for sequential machines. This would simplify programming for parallel processing, as well as enhance modular compiler design. Postulating a data sublanguage and a computation sublanguage, as in FORTRAN or ALGOL-60, Volansky [VOLA 70] defines a control sublanguage with the following capabilities:

- 1) Upon completion of a computation, one or more computations can be initiated (FORK).
- 2) More than one computation may precede another's initiation (JOIN).
- 3) A set of computations can be looped sequentially (DO-loop).
- 4) A set of computations can be looped with

each set executing in parallel (replication).

- 5) Back and forth communication must exist between control and computation sublanguages (e.g., for tests and looping).

To satisfy these requirements, each statement has lists of immediate predecessors and successors statements, as well as a set of computations to be activated either in parallel, sequentially looped, or replicated, depending on the type of statement. Branching statements (IF-type) are also present.

As with COMPEL, this language is hard to read and, further, hard to write. The amount of work involved in writing a program in such a language makes its practicality dubious. Nevertheless, the idea of separating control and computations can be pursued. By defining a few default options, the read and write abilities will be greatly enhanced.

Figure 4a gives an example of a FORTRAN program; Figure 4b, a control sublanguage for its parallel execution [VOLA 70]; and Figure 4c, a middle of the road approach with the description of computational blocks and their parallel control.

Before closing this section on new languages, we should note a now defunct language, TRANQUIL, which was specifically designed for array processors. TRANQUIL [ABEL 69] was an ALGOL-like language with capabilities reflecting the specific hardware. For example, the arrays could be skewed or straight (see Appendix); there were two types of **for** statements, one for sequential execution and one for replication; and new tests were devised for conditional branching, such as:

If any $x < y$ then ... else ... , or

If all $x < y$ then ... else

Loops could be performed either sequentially or in parallel, with a wide range of possibilities for the indexing. TRANQUIL was very much hardware dependent, and could not be considered a general-purpose language.

How easily existing languages, with adequate extensions, can be used for some classes of multiprocessors is an open question. In fact, the approach considered in the next section cannot be applied to special-

```

      DIMENSION A(10),B(10),Y(10),X(10)
1  READ (N,A,B)
2  SA = 0
3  SB = 0
   DO 5 I = 1,N
4  SA = SA + A(I)
5  SB = SB + B(I)
   DO 7 I = 1,N
6  X(I) = A(I) + B(I)
7  Y(I) = X(I)**2
   DO 10 I = 1,N
8  IF(X(I) LT SA) GO TO 11
9  A(I) = X(I) - SA
10 B(I) = Y(I) - SB
11 WRITE (N,A,B)
   END

```

Fig 4 (a) Sample FORTRAN program

```

      @. V1(N) : 'A1,A5';
      A1. V2,V3 : A2;
      A2. <I=1,N : A3;
      A3. V4,V5 : A4,
      A4. END > : A9;
      A5. <I(1/N) : A6;
      A6. V6 : A7,
      A7. V7 : A8;
      A8. END > : A10,
'A9,A10': <I=1,N : A11,
      A11. V8(LV) : ?LV (A14,A12);
      A12. V9,V10 : A13;
      A13. END > : A14,
      A14. V11 : @

```

FIG 4 (b) Control program for sample FORTRAN program.

```

DEC 1  REAL A(10),B(10),X(10),Y(10)
      B 1  READ (N,A,B),
           SA = 0,
           SB = 0,
      B 2  SA = SA + A(I),
           SB = SB + B(I),
      B 3  X(I) = A(I) + B(I),
           Y(I) = X(I)**2,
      B 4  IF (X(I) LT SA) END,
           A(I) = X(I) - SA,
           B(I) = Y(I) - SB,
      B 5  WRITE (N,A,B),
END DEC 1,
      (i) Computational blocks
          SEQ B1, L1, L2, B5, END,
          L1 FORK L3, L4,
          L3 DO I = 1,N B2, JOIN L5,
          L4 DO PAR I = 1,N B3, JOIN L5,
          L5 END L1 (2),
          L2 DO I = 1,N B4,
      (ii) Flow of control

```

FIG 4 (c) Two-dimensional programs.

purpose computers or to array and pipe-line processors.

III. AUTOMATIC DETECTION OF PARALLELISM

In the previous section we have seen how parallelism could be explicitly stated by the programmer. This section is devoted to the problem of detecting parallelism automatically at the compiler level. We shall restrict the discussion to procedure-oriented languages such as FORTRAN and ALGOL-60. Studies in this area fall into two groups: detection of parallelism within an expression; and detection of parallelism between consecutive statements. The former may find its best use in generating code for non-homogeneous multiprocessors with distinct functional units (e.g., CDC 6600), while the latter is geared toward homogeneous multiprocessors.

At this point it is worth noting that hardware mechanisms exist to detect possible concurrent execution/decoding of instructions at run time, given a sequential stream of instructions. Tjaden and Flynn [TJAD 70] delineate the three types of dependency that can prevent two instructions from being executed/decoded in parallel; that is:

- 1) procedural: depending on the flow of control;
- 2) operational: depending on the allocation of resources; and
- 3) data: depending on the utilization of the result of some operation for operands of others.

They propose a pre-decoding stack, an extension of a look-ahead unit, which can be used to decode n instructions (with extended opcodes) in such a way that all independent instructions can be initiated in parallel. In a more conventional look-ahead machine, such as the IBM 360/91, a hardware mechanism is included for dispatching floating-point instructions to their respective functional units. This scheme allows maximal concurrency by a tagging process which does not significantly increase the execution time of the instructions [TOMA 67].

We now turn our attention to the software aspects of the problem.

A. Detection of Parallelism Within an Arithmetic Expression

Transforming an arithmetic expression into a binary tree representation is a well-known process. Here we are interested in a transformation that not only yields a binary tree but also displays the operations which can be performed concurrently, that is, which make use of the associative law of some operators. These operations will be said to be at the same level, where the level of a variable is 0 and the level of the result of an operation involving operands of levels i and j is $\max(i, j) + 1$. The published algorithms (in chronological order [SQUI 63A, HELL 66, STON 67, BAER 68, and RAMA 69]) that perform this transformation can be classified according to their characteristics, namely:

- The type of machine for which they are intended to generate code. All but one of the algorithms cited above are directed toward multi-register, multi-functional units CPU's. The exception is Stone's [STON 67], which is designed for a pipeline push-down stack computer (see Appendix). In [RAMA 69] one can also deduce the maximum number of registers required to evaluate the expression without extraneous STORE and FETCH operations.
- The possibility of reordering of the operands, making use of the commutative law of some operators. Two algorithms [BAER 68, SQUI 63A] take this into account when transforming expressions such as: $(a + b \times c + d)$ into $(a + d) + (b \times c)$.
- The reduction of the number of levels. This depends on the possible reordering of operands and the treatment of unary operators. It seems that the Baer and Bovet's algorithm [BAER 68] performs best when applied to a single arithmetic expression. In [SQUI 63A] "blocks" of assignment statements, without procedural dependency, are treated, possibly leading to a better overall tree structure.
- The number of passes over the input string. Stone's algorithm requires only one pass for the parsing; however, comparing it with the other algorithms is difficult since the code generated does not have

the same structure. Squire's algorithm is the most complex, and requires scanning the input string back and forth many times. The other methods require either as many passes as there are levels [BAER 68], or this number plus an initial pass to transform the input string into Polish notation.

- The simplicity of the code generated. Hellerman's, and Baer and Bovet's algorithms produce a 3-address code, or quadruple, of the form:

temporary result operand 1 operator operand 2

This seems to be an appropriate form of intermediate language.

Summaries of these algorithms have already been presented [BAER 68, REIG 69, RAMA 69]. Therefore, we limit ourselves to the illustration of one of them [BAER 68] for the simplified case where the only operators allowed are "+" and "×".

As stated earlier, the algorithm uses multiple passes, one per level of the resulting binary tree. During each pass, the input string is scanned once from left to right, the result of a call to the scanning procedure being the triplet LSCOP (operator to the left of the operand), ITEM (the operand), and SCOP (operator to the right of the operand). A double-word stack is used (STACK(1, IS) storing operands, STACK(2, IS) storing operators) for entities which cannot yet be part of a temporary result either because of precedence constraints or because a complete quadruple cannot yet be generated. Table 1 shows the parsing of the expression $A + B * C * D + E + F$, and Figure 5 the resulting binary tree.

It is to be noted that in the above algorithms there is no mention of operational dependency; i.e., it is implied that as many functional units and as many registers as necessary are available. (Procedural dependency is taken care of by forbidding side effects and allowing reordering of the operands. Extra parentheses could be added to the expression if one wanted to prevent some of this reordering). Resource allocation has been studied in connection with compilation of FORTRAN expressions for the CDC 6600. In a first paper [ALLA 64],

TABLE 1. COMPILATION OF $A + B * C * D + E + F$ FOR PARALLEL COMPUTATIONS

(Pass 1				Input String $A+B*C*D+E+F$;			
LSCOP	ITEM	SCOP	IS	STACK(1,IS)	STACK(2,IS)	TEMP. RESULTS	OUTPUT STRING
+	A	+	1	A	+		$\emptyset$
+	B	*	2	A	+		$\emptyset$
				B	*		$\emptyset$
*	C	*	1	A	+	$T_1=B*C$	T_1*
*	D	+	1	A	+		T_1*D+
+	E	+	0	$\emptyset$	$\emptyset$	$T_2=A+E$	T_1*D+T_2+F ;
+	F	;	0	$\emptyset$	$\emptyset$		T_1*D+T_2F ;
(Pass 2				Input String $\leftarrow$ Output String = T_1*D+T_2+F ;			
+	T_1	*	1	T_1	*		$\emptyset$
*	D	+	0	$\emptyset$	$\emptyset$	$T_3=T_1*D$	T_3+
+	T_2	+	1	T_2	+		T_3+
+	F	;	0	$\emptyset$	$\emptyset$	$T_4=T_2+F$	T_3+T_4 ;
(Pass 3				Input string $\leftarrow$ Output String = T_3+T_4 ;			
+	T_3	+	1	T_3	+		$\emptyset$
+	T_4	;	0	$\emptyset$	$\emptyset$	$T_5=T_3+T_4$	T_5 ;

81

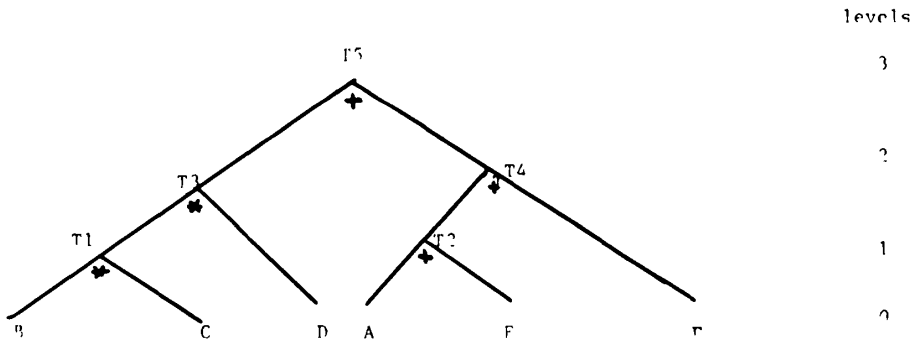


FIG. 5. Binary tree for the parallel computation of $A + B * C * D + E + F$.

Allard, et al., created an intermediate language which displays not only the temporary results generated, but also registers and units usage. Subsequent passes on the intermediate string remove unnecessary FETCH and STORE instructions, and perform register assignment with the goal of reducing delays when issuing instructions to functional units. The starting point of a second study [THOR 67] is an acyclic directed graph representation of a series of assignment statements; resources are subsequently

allocated according to a PERT time scheduling. Until now, we have looked only at the impact of the associative and commutative laws in the construction of the binary tree. In his PhD thesis, Muraoka [MURA 71] investigates the possibility of decreasing the number of levels in the tree by making use of the distributive law. This is far from being a trivial problem, since in some cases application of the distributive law lowers the height of the tree while in others it may have the

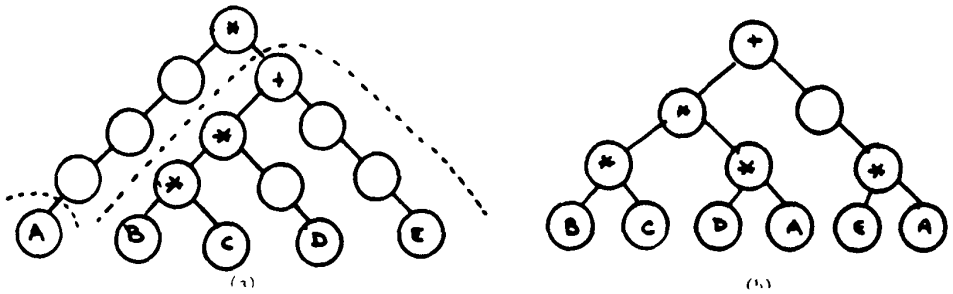


FIG. 6 Applying distributive law to fill up "holes."

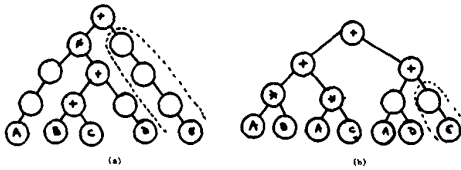


FIG. 7. Applying distributive law to create "spaces."

opposite effect. For example, consider the expression $A * (B * C * D + E)$. Applying the distributive law will reduce the number of levels in the tree from 4 to 3 (see Figure 6). On the other hand, the expression $A * B * (C + D)$, left as is, will be transformed in a tree of level 2, while the expanded expression $A * B * C + A * B * D$ requires a tree of level 3. Muraoka's algorithm detects the cases where the distributive law will help (or rather will not hinder) in lowering the height of the tree. The process being quite complex, we shall illustrate it with two simple examples.

Consider the binary tree for $A * (B * C * D + E)$ obtained by application of one of the previous algorithms (see Figure 6a). At some levels there are free nodes (noted as circles in the figure). Some of these "holes" in subtree T_1 can be filled by the entire subtree T_2 . In general, the possibility of such a "filling" depends on the size of the holes in T_1 compared to the height of T_2 . The replacement is illustrated in Figure 6b.

The second possibility for obtaining a lower tree is by creating "spaces" to be filled later. This is shown in Figure 7 for the expression $A * (B + C + D) + E$.

The main disadvantage of the detection

of parallelism at this level is in the extra time necessitated by the compiling process, which might not be compensated for by the speed-up at execution time. Muraoka's process increases this ratio by at least one order of magnitude.

B. Detection of Parallelism Between Statements

Conditions for Statements Independency

Data dependency is the main factor for the inter-statement detection of parallelism. Bernstein [BERN 66] was the first to attack the problem; his results are summarized below.

Variables which are referenced during the execution of a segment of program P_i can be classified into four categories:

- 1) those which are only fetched, i.e., the set W_i ;
- 2) those which are only stored, i.e., the set X_i ;
- 3) those which are first fetched and then stored, i.e., the set Y_i ; and
- 4) those which are first stored and then fetched, i.e., the set Z_i .

Suppose now that we have two processors sharing a main memory. We want to know the conditions under which two segments P_1 and P_2 , originally scheduled to be executed sequentially before a third segment P_3 , can now be initiated in parallel. The three requirements described below are mandatory.

- 1) Locations which are fetched in P_1 , i.e., $W_1 \cup Y_1 \cup Z_1$, must not be destroyed by storing operations from P_2 , i.e., $X_2 \cup Y_2 \cup Z_2$, that is:

$$(W_1 \cup Y_1 \cup Z_1) \cap (X_2 \cup Y_2 \cup Z_2) \\ = \Phi \text{ (}\Phi \text{ is the empty set).}$$

- 2) By symmetry, exchanging the roles of P_1 and P_2

$$(X_1 \cup Y_1 \cup Z_1) \cap (W_2 \cup Y_2 \cup Z_2) = \Phi$$

- 3) When entering P_3 , the state of the machine (the contents of the total memory locations) must be the same independently of the mode (parallel or sequential) of execution of P_1 and P_2 ; that is, the variables first fetched in P_3 , i.e., $W_3 \cup Y_3$, must be independent of the storing operations in P_1 and P_2 yielding:

$$(W_3 \cup Y_3) \cap (X_1 \cup Y_1 \cup Z_1) \\ \cap (X_2 \cup Y_2 \cup Z_2) = \Phi$$

This can be reduced, by taking into account the first and second requirements above, to:

$$X_1 \cap X_2 \cup (W_3 \cup Y_3) = \Phi$$

If one looks at P_i as a statement of a high-level language, and if I_i and O_i represent, respectively, the input (those variables which appear only at the right of an assignment statement) and output data sets of P_i , then the requirements above can be rewritten as:

$$I_1 \cap O_2 = \Phi$$

$$I_2 \cap O_1 = \Phi$$

$$O_1 \cap O_2 = \Phi$$

It is based on these conditions that systems have been written for automatic detection of parallelism in source programs written in high-level languages. The most widely publicized are for FORTRAN IV (namely, the UCLA implementation [RUSS 69, VOLA 70] and the parallel task recognizer of Ramamoorthy et al. [RAMA 69, RAMA 69A, GONZ 70]) and for a subset of ALGOL-60 (Burrough's group [BING 67, FISH 67, REIG 69, REIG 70]).

Automatic Program Analysis

The main frames of these systems are similar. In a first phase a directed graph

representation of the original program is constructed along with Boolean connectivity and precedence matrices (see, e.g., [RAMA 66]). Each statement is a node of the graph, and arcs represent flow of control. Loops are detected and their feedback arcs deleted at the same time, the result being an acyclic directed graph. Procedural dependency (also called essential ordering) is recorded, so that there will be no attempt to find independency between statements which are mutually exclusive, such as, for example, two branches of an IF statement. Finally, examination of input and output data sets is performed, yielding a "parallel" graph and associated Boolean matrices. We illustrate this process with an example taken from Russell's PhD thesis [RUSS 69].

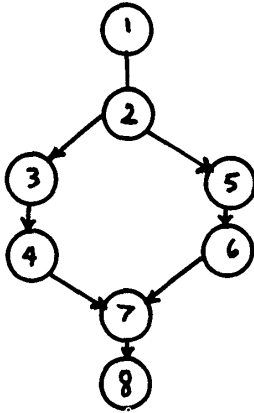
The initial program can be represented by the directed graph of Figure 8a, or equivalently by the connectivity matrix C of Figure 8b. Logical connections, i.e., those implying explicit transfer of control, are preserved in the logical matrix L (Figure 8c). Input and output data sets for each statement are recorded in $n \times m$ Boolean matrices, where the n rows represent statements and the m columns names of variables. Thus, $I_i' = 1$ ($O_i' = 1$) if variable j is an input (output) to statement i . For the example program, they are shown in Figure 8d,e. Finally, the precedence matrix P is shown in Figure 8f. (It could have been computed through Warshall's algorithm [WARS 62].)

The goal of the subsequent step of the analysis is to generate a connectivity matrix C_p showing possible parallelism while preserving flow of control. In order to do so two Boolean matrices are built: P_p , an incomplete parallel precedence matrix; and L_p , a parallel logic matrix. Initially $P_p = P$ and $L_p = L$. Then the following test is performed on all pairs of statements (i, j) .

For each $P_{pi}' = 1$ examine the set

$$I0 = (I_i \cap O_j) \cup (I_j \cap O_i) \cup (O_i \cap O_j)$$

If $I0 = \Phi$, then statements i and j are independent and, therefore, their precedence relationship should not be preserved, i.e., P_{pi}' is set to 0. Also, to preserve logical connections, the assignment $L_{pi} = L_{pi} \cup L_j$ is done (A_j is the j th row of matrix A); that



(a) Sequential Graph

	1	2	3	4	5	6	7	8
1	0	0	0	0	0	0	0	0
2	1	0	0	0	0	0	0	0
3	0	1	0	0	0	0	0	0
4	0	0	1	0	0	0	0	0
5	0	1	0	0	0	0	0	0
6	0	0	0	0	1	0	0	0
7	0	0	0	1	0	1	0	0
8	0	0	0	0	0	0	1	0

 (b) Connectivity
Matrix C

	1	2	3	4	5	6	7	8
1	0	0	0	0	0	0	0	0
2	0	0	0	0	0	0	0	0
3	0	1	0	0	0	0	0	0
4	0	0	0	0	0	0	0	0
5	0	1	0	0	0	0	0	0
6	0	0	0	0	1	0	0	0
7	0	0	0	1	0	0	0	0
8	0	0	0	0	0	0	0	0

 (c) Logical
Matrix L

	1	2	3	4	5
1	0	0	0	0	0
2	0	1	0	0	0
3	0	0	0	0	0
4	0	0	0	0	0
5	0	0	0	0	0
6	0	0	0	0	0
7	0	0	1	1	0
8	0	0	0	0	1

 (d) Input
Matrix I

	1	2	3	4	5
1	0	1	0	0	0
2	0	0	0	0	0
3	0	0	1	0	0
4	0	0	0	1	0
5	0	0	1	0	0
6	0	0	0	1	0
7	0	0	0	0	1
8	0	0	0	0	0

 (e) Output
Matrix O

	1	2	3	4	5	6	7	8
1	0	0	0	0	0	0	0	0
2	1	0	0	0	0	0	0	0
3	1	1	0	0	0	0	0	0
4	1	1	1	0	0	0	0	0
5	1	1	0	0	0	0	0	0
6	1	1	0	0	1	0	0	0
7	1	1	1	1	1	1	0	0
8	1	1	1	1	1	1	1	0

 (f) Precedence
Matrix P

Fig. 8. Initial graph representation of a program for detection of parallelism between statements.

is, all logical connections leading to j must also now lead to i .

In the example of Figure 8, one can see that for $i = 4$ and $j = 3$, I_0 is empty; thus, P_{p4}^3 is set to 0, and L_{p4} initially empty is set to L_3 (i.e., $L_{p4}^2 = 1$).

The result of this algorithm is shown in Figure 9a,b. Since the connecting matrix C_p is supposed to show only direct connections of length 1, one needs to generate from the precedence matrix P_p , the minimal connectivity matrix C_p' . A possible way to get C_p' is to use

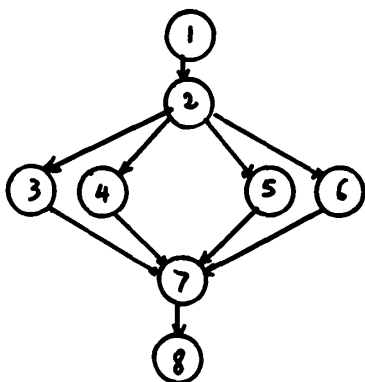
$$C_p' = P_p \cap \left(\bigcup_{i=2}^n P_p^i \right)$$

[SIMO 65]. In the example $C_p' = P_p$. Finally, the ultimate C_p is the union of C_p' and L_p , i.e., $C_p = L_p \cup C_p'$. It is shown in Figure 9c along with the graph it represents (Figure 9d). In order to separate branching and parallel control, dummy vertices (corresponding to FORK and JOIN) are added, resulting in the graph of Figure 9e.

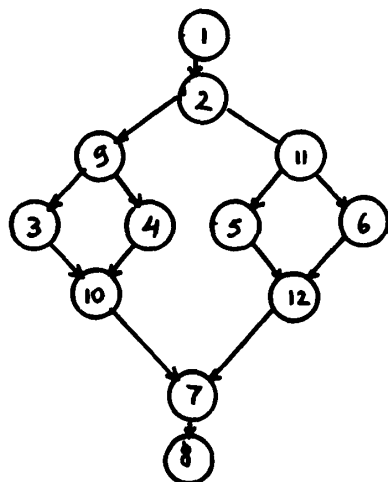
The method outlined above yields only limited success in detecting parallelism for three reasons:

- 1) In the input and output matrices, arrays have to be considered as a single entry: thus, subscript analysis is not performed. Considering each element of an array as

	1	2	3	4	5	6	7	8		1	2	3	4	5	6	7	8		1	2	3	4	5	6	7	8
1	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0
2	1	0	0	0	0	0	0	0	2	0	0	0	0	0	0	0	0	2	1	0	0	0	0	0	0	0
3	0	0	0	0	0	0	0	0	3	0	1	0	0	0	0	0	0	3	0	1	0	0	0	0	0	0
4	0	0	0	0	0	0	0	0	4	0	1	0	0	0	0	0	0	4	0	1	0	0	0	0	0	0
5	0	0	0	0	0	0	0	0	5	0	1	0	0	0	0	0	0	5	0	1	0	0	0	0	0	0
6	0	0	0	0	0	0	0	0	6	0	1	0	0	0	0	0	0	6	0	1	0	0	0	0	0	0
7	0	0	1	1	1	1	0	0	7	0	0	0	1	0	0	0	0	7	0	0	1	1	1	1	0	0
8	0	0	0	0	0	0	1	0	8	0	0	0	0	0	0	0	0	8	0	0	0	0	0	0	1	0
(a) Parallel Precedence Matrix P_p									(b) Parallel Logic Matrix L_p									(c) Parallel Connectivity Matrix C_p								



(d) Equivalent Parallel Graph



(e) Final Representation of Parallel Graph

FIG. 9. Transformation from "sequential" to "parallel" graph.

- an individual entry in I and O would lead to unmanageable storage problems.
- 2) A common practice in programming (and a very justified one) is to minimize the number of temporary locations one wants to use. In other words, the same name can be used in various statements of a program for completely different purposes. This can hide potential parallelism as shown below.

- 3) As stated in Section II, most parallelism is found in replication of loops. The above method does not treat this aspect of the problem. Some attempts at solving it will be reviewed shortly.

Discovering the Hidden Parallelism

Following Volansky [VOLA 70], we now look at the "memory deallocation" problem.

1	A = B + C	A1 = B + C
2	D = A + E	D = A1 + E
3	A = X + Y	A = X + Y
4	F = A * Z	F = A * Z
(a) Original segment		(b) Modified segment
	1 2 3 4	
1	0 0 0 0	$W_{k1} = (1, 0, 0, 0)$
2	1 0 0 0	$W_{k2} = (0, 0, 1, 0)$
3	0 0 0 0	Add variable A1 output of statement 1 input
4	0 0 1 0	to statement 2
(c) Utilization matrix U_A ($U_A^* = U_A$)		

FIG 10. Memory deallocation example

Consider the (trivial) example of Figure 10a. If one follows Bernstein's conditions, statements 1 to 4 have to be performed sequentially. If now, a new variable A1 is introduced as in Figure 10b, then statements 1-2 on one hand and 3-4 on the other can be performed concurrently, and the three original variables A, D, F keep the same final values as in Figure 10a. To automate this transformation, new $n \times n$ (n being the number of statements) Boolean matrices U_k are introduced, one for each variable k of the program. $U_{ki} = 1$ if and only if the value, associated with variable k that is input to the computation of statement i , is output by the computation of statement j . (See Figure 10c for U_A in the program of Figure 10a.)

For each variable k , U_k and its transitive closure U_k^* are formed. Then if U_k^* has $(m+1)$ unique non-zero rows $W_{k_0}, \dots, W_{k_m}$ new variables are introduced, say $k_1, k_2, \dots, k_m$. Each new variable k_i is an output of all statements p such that $W_{k_i}(p) = 1$ and an input to all statements i for which $U_k^*(i, j) = 1$ and $W_{k_i}(j) = 1$.

The main drawbacks of this method are:

- time loss: in order to form the different U_k 's, all possible execution paths have to be considered. Since there may be millions of them and since the computations have to be repeated for each variable, this cannot be practically considered; also for each U_k one has to form U_k^* .
- storage loss: each U_k^* has to be stored. For example, in a program of 100 statements and 20 variables, the number of bits required in $100^2 \times 20 = 2.10^5$, which can be prohibitive (the number of operations to obtain the U_k^* 's from the U_k would be of order greater than $O(20 \times 100^2)$).

Some of these disadvantages could be reduced if, instead of looking at all possible paths, only segments with no procedural dependency were considered. The amount of work needed would be orders of magnitude less with—in this author's opinion—almost as much success at discovering the "obscured" parallelism.

Loop Replication

Given a FORTRAN DO-loop or an ALGOL-60 **for** statement, how can one detect if all executions of the loop must be performed sequentially, or if all of them can be executed simultaneously, or if some of them can be replicated, with the number possibly varying dynamically?

The total replication test can be approached at different levels of sophistication. Let $L = \{S_1, \dots, S_i, \dots, S_n\}$ be the statements composing the DO-loop. Then one can form the following input and output sets: $I_L = UI, O_L = UO$, where I_j and O_j are the input and output sets formed with variables referenced within L , with each referenced subscripted array being an individual entry. If $I_L \cap O_L = \Phi$, then all loop iterations can be replicated, for example:

```
DO 10 I = 1, N
```

```
10 A(I) = B(I)
```

But if $I_L \cap O_L \neq \Phi$, then one can look at the variables for which conflicts arise. If those are set before they are used then the conflict is artificial and replication is permissible as, for example, in:

```
DO 10 I = 1, N
```

```
A(I) = f(A(I))
```

```
T = g(A(I))
```

```
10 B(I) = h(T)
```

where a different T could be set aside for each replication. Russell [RUSS 69] gives an algorithm to test these conditions.

The partial replication problem is far more complex. Suppose that the first $(k-1)$, $(k > 4)$, values of an array A have already been computed. One wants now to compute the remaining values $A(k+1)$,

..., $A(N)$ by:

```
DO      10      I = K,N
10      A(I) = A(I - 2) + A(I - 3)
```

It is readily observed that $A(I)$ and $A(I + 1)$ could be computed in parallel, but automatic analysis of it is not evident. Muraoka [MURA 71] presents some theoretical considerations on the partial replication problem.

Experimental Results

Both Russell and Volansky have tested their algorithms on sample programs. As could be expected, most of the discovered parallelism was found in loop replication. For some subroutines, potential savings of up to 50% were displayed but they require a somewhat excessive number of processors to exploit full parallelism. The memory deallocation algorithm was tested only on a limited number of execution paths, and it showed that some improvement could be expected. Kuck and his group at Illinois [KUCK 72] found that up to 16 processors could be used efficiently for small to medium size programs. To attain this amount of parallelism, special hardware for the treatment of conditional statements, and Muraoka's sophisticated algorithms for loop replication and height reduction of arithmetic trees, had to be used. Finally, it is obvious that not all programs submitted to a multiprocessor facility should be analyzed for parallel processing since the amount of time spent in the detection could be far more expensive than the savings at execution time. Phase I of the recognition process, i.e., the generation of the sequential graph, can partially answer the question by looking at the distribution of statement types [GONZ 71] as well as the expected weight, timewise, of individual subroutines and DO-loops. Then the analysis could be restricted to those subtasks that are important enough to be optimized and, at the same time, amenable to parallel processing.

If some of these comments appear to be on the pessimistic side, the fault does not reside necessarily on the detection programs. No automatic analysis can detect the paral-

lelism inherent in the following loop:

```
DO      10      I = 1,N
IF(A(I).NE.0)      GO TO 10
J = I
GO TO 20
10      CONTINUE
```

This is the representation of a sequential process looking for the first null element in an array, which is also the only way that a FORTRAN programmer can test if at least one element of the array is null. Although this last action is purely parallel, there is no way to say so in classical high-level languages and the parallelism cannot be uncovered. Thus, automatic detection of parallelism could be a more powerful tool if it were combined with the language extensions described in Section II.

IV. MODELS OF PARALLEL COMPUTATIONS

A. Generalities Relative to the Modeling of Parallel Computations

In the previous sections we have surveyed software techniques that extend the domain of computing from uniprocessing to multiprocessing. Concurrent to this development a large amount of research devoted to more theoretical studies of parallel computations and structures was, and is still, in progress. By means of models—directed graphs, networks, program schemata—numerous authors have investigated properties of parallel systems. Depending on their approach and background (engineers, logicians, mathematicians), their goals have been somewhat different. Included among the objectives toward which their efforts have been oriented are:

- The design of modular parallel systems and their interconnection,
- The control techniques that allow sharing of procedures and data among processors;
- The relations between sequential programs and their transformed representation for parallel processing; and
- The prediction in increase of performance

and cost when using a multiprocessor instead of a conventional single processor.

Modeling of parallel computations is a field in which there is still need for new concepts and ideas. In this part of the survey we shall not try to cover all existing models, but shall look at those which, in our opinion, have the most distinctive features.

We assume the reader is familiar with elementary notions of graph theory. The terms graph and net, node and vertex, arc, branch and link will be considered as synonymous. In general, nodes will be the images of operators (or computations), while arcs may represent either variables or flow of control, or both.

Adapting our notation from that of a recent survey of some of the models presented below [BRED 70], we define a parallel program model as the triple $\mathcal{P} = (W, U, C)$ where:

$W = \{w_1, w_2, \dots, w_n\}$	is a set of operators (or computations);
$U = \{u_1, u_2, \dots, u_m\}$	is a set of variables; and
C (to be defined)	is a control.

With each operator is associated an input set $I_i = \{u_{i1}, \dots, u_{ik}\}$ and an output set $O_i = \{u_{o1}, \dots, u_{op}\}$

This triple will serve as a frame of reference for describing the various models.

A simulation on $\mathcal{P}$ is defined by:

- For each variable u , a domain D , of values which the variable may assume.
- For each operator w , a mapping $O_i = f(I_i)$. This mapping can be both computational (values associated with members of O_i are defined in terms of values associated with members of I_i) and logical (control conditions can be dictated by operators).
- A set of initial values for members of U .

A variable history h_i is the sequence of values associated with u_i during a simulation on $\mathcal{P}$. A program history is the m -tuple $\{h_1, \dots, h_m\}$ consisting of the variable histories of the elements of U .

An important property of a parallel program model is its determinacy. A program is said to be determinate if the sequence of values associated with each variable is unique. More precisely, $\mathcal{P}$ is said to be determinate if and only if each simulation results in a unique program history. As we shall see, less stringent determinacy conditions can also be considered. Depending on the author, determinacy will be approached from one of two different viewpoints. In the first, only certain types of operators and associated control will be allowed in $\mathcal{P}$, and determinacy will be proved by construction. In the second, arbitrary control or interconnections will be permissible, and determinacy becomes a decision problem.

Finally, we shall say that two program models, $\mathcal{P}$ and $\mathcal{P}'$ defined by the triples (W, U, C) and (W', U, C') , are equivalent if their program histories are the same.

B. A First Look at Directed Graph Models

The models we are reviewing have existing programs or descriptive algorithms intended for single processors as their main point of departure. Their goals are prediction of performance for hypothetical multiprocessors, design of new special purpose systems, and provision of "a workable form of organization with complete specification of the information flow" [PETR 62].

Directed Acyclic Bilogic Graphs

Historically the intent of the model by [ESTR 63A, TURN 63] (also known as the UCLA graph model) was to give a representation of programs to be run on a variable structure computer (see Appendix). Several investigators under the direction of Gerald Estrin have refined the original model. Starting from a FORTRAN source program, the model can be automatically derived (see Section III. B [RUSS 69, VOLA 70]) and tested for its well-formedness. We shall see in Section V how it can be used for the evaluation and design of multiprocessors; here we give only its formal structure.

The two first elements of $\mathcal{P}$, that is W and U , are, respectively, the sets of vertices and

arcs of a directed graph $G(W, U)$. The vertices are images of single FORTRAN statements (see [RUSS 69] for exceptions), and the arcs represent flow of control. The function f mapping I , onto O , is not completely expressed (the model is said to be partially interpreted), but some of its characteristics, called vertex attributes, such as time, frequency, etc., can be detailed. We define now the control C of the model.

With each vertex w , is associated one of the ordered pairs $(*, *)$, $(*, +)$, $(+, *)$, $(+, +)$. If the first member of the pair is $*$, w , is said to be of AND-input logic (otherwise, it is of Exclusive-OR (EOR) input logic). Similarly, w , is of AND-output logic (EOR-output logic, also called branching vertex) depending on the value of the second element of the pair. A simulation on $\mathcal{P}$ proceeds as follows:

- The initial vertex is initiated (it is supposed unique).
- At termination of an EOR-output logic vertex, one and only one of the arcs incident out is enabled and all others disabled (see IF statement in a procedural language).
- At termination of an AND-output logic vertex, all arcs incident out are enabled (see FORK statement in Section II.A).
- To be initiated an EOR-input logic vertex must have one and only one of the arcs incident in enabled.
- To be initiated an AND-input logic vertex must have all of the arcs incident in enabled (see JOIN statement of Section II.A).

In the original model inclusive-OR input logic was also allowed. It was discarded for ease of analysis. Martin [MART 66] has shown how it could be eliminated, leaving only bilogic conditions, by introduction of dummy vertices.

Most studies of this model [MART 66, BAER 68A] imply an acyclic directed graph. The rationale behind the cyclic/acyclic transformation as well as its actual implementation will be given later. Here we consider only directed acyclic bilogic (d.a.b.) graphs, of which Figure 11a is an example.

In their studies of d.a.b. graphs, the UCLA group does not consider determinacy

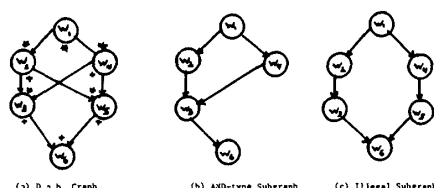


FIG 11. UCLA graph model.

requirements (note that there are no variable histories defined), and tests only the “legality” of the graphs [BAER 70] so that simulations on the model can be meaningful. A legal graph must have a unique initial vertex w_1 (i.e., one without predecessors), a unique terminal vertex w_n (i.e., one without successors); and all subgraphs obtained in a simulation on $\mathcal{P}$ must be AND-type subgraphs [MART 67] $A(W', U')$ of $G(W, U)$.

Starting at step 1) below, the construction of an $A(W', U')$ proceeds as follows [BAER 70]:

- 1) $w_1 \in W'$, $w_n \in W'$, i.e., the initial and terminal vertices of the graph belong to all AND-type subgraphs.
- 2) If $w \in W'$ and is of AND-output logic, U' contains all arcs of U incident out of w .
- 3) If $w \in W'$ and is a branching vertex, one and only one of the arcs incident out of w , is in U' .
- 4) A vertex w , of AND-input logic belongs to W' if and only if all the arcs incident in w , belong to U' .
- 5) A vertex w , of EOR-input logic belongs to W' if and only if one and only one of the arcs incident in w , belongs to U' .
- 6) If W' and U' satisfy 1) to 5) above and if there exist arcs in U' such that only one of their extremities is in W' , then such arcs are suppressed from U' .

Figure 11b shows an AND-type subgraph for the d.a.b. graph of Figure 11a, while the subgraph of Figure 11c is not an AND-type subgraph.

A test for the legality of a d.a.b. graph could be done by tracing, starting at the initial vertex w_1 , applying rules 1) to 6) above for all possible choices when encountering a branching vertex; the subgraphs obtained should be AND-type subgraphs, that is, terminating at w_n . Such a

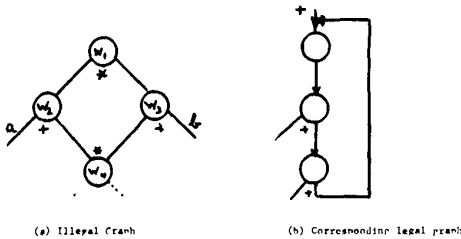


FIG. 12. Example of an illegal d.a.b. graph

combinatorial process has to be rejected because of the magnitude of the number of AND-type subgraphs which can be present in a d.a.b. graph. Rules to test the legality of the graph by looking only at the logics of the vertices in a one-pass scan over the graph have been defined [BAER 70, BOVE 69]*. To get the flavor of these rules, consider the example of Figure 12a. The graph is illegal; since w_n is unique, there must exist a common successor, say w_k to w_2 and w_3 . This successor can be reached if arc a and not arc b is traversed, or if arc b and not arc a is traversed, or if both arcs a and b are traversed. Therefore, the input of w_k is by inclusive-OR logic, and such a structure is not allowed. As we can see, we encounter the same problem as the one already pointed out for the automatic detection of parallelism when trying to "parallelize" statements of the form "Test and see if at least one element of an array is 0 and if so go to statement w_k ." The legal graph has to be cyclic and of the form given in Figure 12b.

This model has been recently extended [GOST 71, CERF 72] with the introduction of multi-head, multi-tail arcs, and a more formal definition of operator actions. This yields a model closer to the Petri nets and the flow graph schema described below. A reduction procedure, reminiscent of Bovet's approach for legality testing [BOVE 69], can be applied to cyclic bilogic graphs to test their proper termination. The latter can be viewed as a simulation on the graph, starting with the initiation of w_1 , ending with the operation associated with w_n , and

* An error in [BAER 70] has been reported by Mrs. Lepeuve; however, the reduction procedures [BOVE 69, GOST 71] are correct

leaving no other vertex either operating or ready for operation. Prevention of deadlock and use of semaphores become feasible with this model. Determinacy has also been studied [REGI 72], using a constructive proof method, for trilogic acyclic graphs where an inclusive-OR input logic replaces the exclusive-OR input logic.

In the d.a.b. graph model, there is no mention of the structure of the data. Some thoughts along this line have been expressed by Dennis [DENN 68]. His model is an acyclic directed graph where the nodes without successors are elementary items of information, either data values or instructions. The whole graph represents the environment, and subgraphs of it are called information structures. The latter can be referred to by pointers, i.e., arcs which have names. Thus, these subgraphs can represent files, records, arrays, procedures, etc. Each computational node is the image of a specific operation, that is, the nodes have more interpretation attached to them than in the UCLA model. Control is monitored through arc status (either start or done) in a manner similar to some of the models we describe below. In his PhD thesis, Gertz [GERT 70] blends the d.a.b. graphs and the information structure concepts. His intent is to model associative memories and algorithms for their efficient use. The computational part—which he calls a precedence graph—is a d.a.b. graph with arc names, such that no two arcs incident out from a node can have the same name. These arcs are pointers to parameter graphs, that is, graphs modeling the data structures operands of a computation. A step in the simulation on the model consists in taking an operand from the set of parameter graphs, in performing the given operation, and in returning the operand of which the internal structure may have been modified. Figure 13 gives an example of a parameter graph.

Petri Nets and Extensions

The main objective of Petri nets (named after their "inventor" whose thesis [PETR 62] describes nets more general than those which bear his name) and of some of their

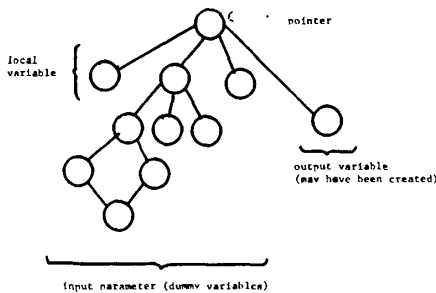


FIG. 13. Example of a parameter graph associated with a procedure call.

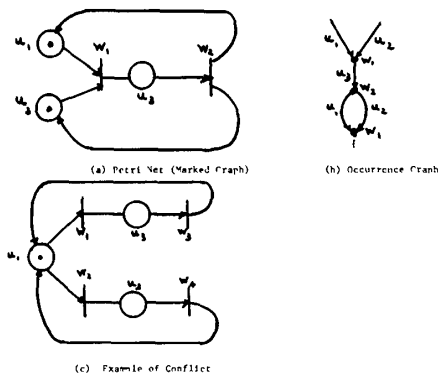


FIG. 14. Petri nets.

extensions is to represent concurrent conditions, both static and dynamic, which can exist in a system. Further extensions deal with a better representation of the coordination of asynchronous events. We begin by giving a formal description of the nets.

A Petri net is a triple $\Phi = (W, U, C)$ where: $W = \{w_1, w_2, \dots, w_n\}$ is a finite set of transitions (corresponding to events); $U = \{u_1, u_2, \dots, u_m\}$ is a finite set of places (corresponding to conditions); and C , the control, has for its basic component a set of directed arcs $A = \{a_1, \dots, a_p\}$ which connect elements of W to elements of U and vice-versa. If $a_i = (u, w_k)$, u is an input place to transition w_k and the set of input places to w_k is defined as $I_k = \{u, \mid (u, w_k) \in A\}$. Similarly, output places to a transition as well as the set of output places

are defined. For example, $O_k = \{u, \mid (w_k, u) \in A\}$.

Places are able to receive tokens. They can be either full, i.e., they have one or more token, or empty, i.e., they have no token. Since places represent conditions, a full place displays the holding of a condition. Holdings of conditions direct the firing of transitions—i.e., the occurrence of events—and transition firings in turn monitor condition holdings. In order to fire, a transition must have all its input places full. The firing of w_k removes one token from each member of I_k and places one token in each member of O_k . An instance of the history of the net, called a state, is represented by the distribution of tokens into places. Firing of a transition changes the state of the net. Figure 14a gives an example of a Petri net in a given state. Circles represent conditions, vertical bars show transitions and the dots are the tokens.

A simulation of the net proceeds as follows: 1) Define the initial conditions, i.e., the set of places $U_0 \subseteq U$ which initially have tokens. This is the initial state. 2) Given a current state, choose any of the transitions which are able to fire and fire them. This defines a new system state. 3) Stop if no more firing can occur, or after some finite number of firings.

The simulation itself can be represented by an acyclic directed graph $G(W, U)$ with the proviso that vertices can be linked by more than one arc. Such graphs, called occurrence graphs, have been studied by Holt [HOLT 68]. An example is shown in Figure 14b corresponding to the initial state of Figure 14a. Pragmatic modelling of queues, stacks, shift registers, etc., by occurrence systems, and compositions of occurrence graphs may be found in the report cited above, as well as in [SHAP 70].

A Petri net is said to be safe if a place cannot have more than one token in it at any time. A transition w is live if given the initial state there exists a sequence of states such that w will fire. A net is live if all its transitions are live. The concepts of safety and liveness replace that of determinacy in the studies of Petri nets. The latter is not possible here since conflicts in transition

may arise. More precisely, two transitions are in conflict if they share a common input place and the two transitions are enabled. For example, in Figure 14c w_1 and w_2 share the input place u_1 . With a token placed in u_1 an infinite number of different simulations can occur depending on which of w_1 and w_2 is fired since both are eligible.

Conservation of tokens in paths, liveness, and safety have been formally studied for a subset of Petri nets called marked graphs [HOLT 69, COMM 71]. A marked graph is a Petri net with the restriction that neither input nor output places can be shared by transitions. This allows concurrent conditions (e.g., u_1 and u_2 in the Petri net of Figure 14a which is also a marked graph) and avoids conflicting transitions. Another subset (transitions have only one input and one output place which can be shared) corresponds to state machine graphs (see Figure 14c).

In his PhD thesis, Patil [PATI 70] extends the control part of $\mathcal{O}$ by defining constraint sets, that is, the sets of places which should not have tokens at the same time (the constraint sets $C_i \subset \mathcal{Z}^U$, the power set of U). Input and output places of a same transition are not allowed to be part of a same constraint set, a mild restriction since it may simply imply the inclusion of supplementary transitions. Now during the simulation of the net, a transition can be fired only if the set of active places (that is, those which are already full and those which will become so after the firing) does not include any constraint set. Setting up the constraint sets leads to the definition of homogeneous coordination nets which in turn allow the implementation of operators such as those defined by Dennis [DENN 68] and their linkage.

In terms of the logic defined previously, one can see that transitions in a Petri net are of the AND-input AND-output form. Petri nets with EOR-input, EOR-output and Inclusive-OR input logics have been introduced by Noc for the description of operating systems [NOEJ 71] at different levels of detail. This notation could be useful as a "parallel flowchart" for simulation languages in the same way as GERT

[PRIS 66] and GASP are related. Further development in this direction can be found in Nutt's PhD dissertation [NUTT 72].

C. Graph Models With Extensive Control

Rodriguez [RODR 67] has extended the ideas behind the bilogic graph model and the Petri nets in terms of more specific, and thus more interpretative, operators and more elaborate control. The first two members of $\mathcal{O}$ again form the basis of a directed graph $G(W, U)$, with W being the set of operators and U the set of variables. With each variable is associated a status, either: disabled (-1), idle (0); enabled (1), or blocked (2). These variables also carry values and, hence, there exists two kinds of arcs: data arcs which have both a value and a status, and control arcs which only have a status. The computation represented by a vertex can only be initiated if the input and output arcs have the correct status, each operator type having its own set of enabling conditions. Termination of the computation involves not only transfer of values but also changes in status of incoming and outgoing arcs. The mapping $0_i = f(I_i)$ is both computational and logical.

The operator types which are defined are: input and output terminals, data operator functions, identity (i.e., duplication of data), AND and OR control operators, selector (i.e., branching vertex), junction, loop junction, and loop output. As an example consider the selector operator (see Figure 15a). β is a predicate on the data variables residing on arcs 1 and 2. When both these arcs are enabled (status=1) and the output arcs are in idle status (0) the predicate can be computed. At termination either control arc 3 or control arc 4 will be enabled, while the other will be disabled. Also, the incoming arcs 1 and 2 will become idle. Notice that if either arc 1 or arc 2 is disabled, both outgoing arcs become disabled. The most complex node of the model is the loop junction (see Figure 15b). Arc 1 represents the initial entry in the loop, arc 2 the feedback arc, arc 3 the control arc for termination, and arc 4 will carry the value of the computation performed by the node on the input value assigned to either arc 1 or 2. An initial entry

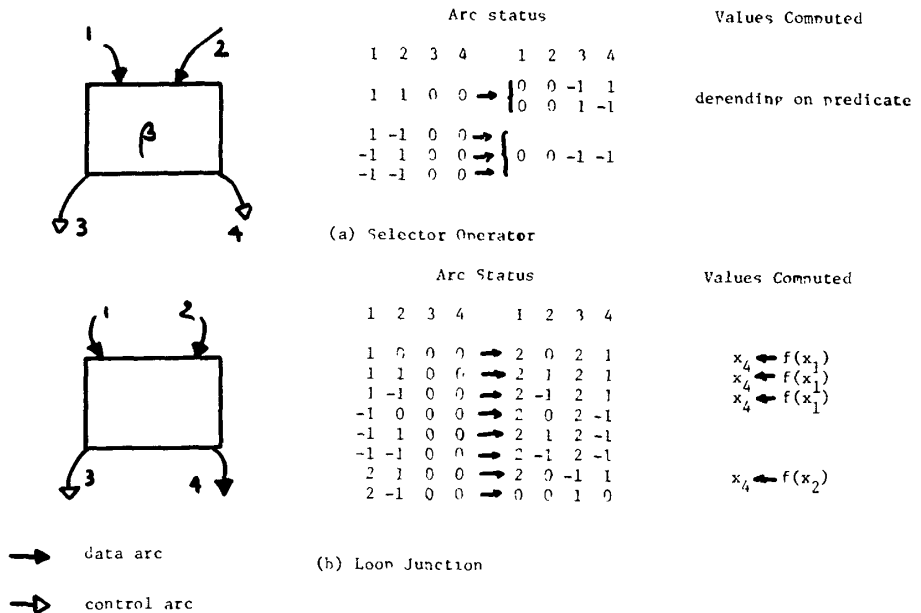


FIG 15 Examples of Rodríguez's operators.

in the loop can be made when arc 1 is enabled and all the others are idle. (Note that there are two other possibilities since the status of arc 2 is irrelevant as long as it is not blocked.) After this initiation, arcs 1 and 3 become blocked and the value of arc 1 is transmitted to arc 4 which becomes enabled. A feedback iteration is enabled when arc 2 is enabled (note that arc 1 is now blocked), arc 3 is idle (it will be set that way by the loop output operator), and arc 4 is idle. Arc 3 will become enabled when arc 1 is blocked and arc 2 disabled.

The only constraints in building a graph is that data arcs should be attached to data connectors of operators, and likewise for control arcs and control connectors. Also, every input connector of a node must be attached to some arc.

The initial state of a simulation is to have all arcs idle except for those attached (generally all) to enabled input terminals. The behavior or state of the graph is determined by the arcs status and the data contents of

the arcs. It determines which nodes can perform their operations. Then the simulation, here called execution sequence, is a sequence of states. A terminal state is one in which no node can be initiated. A hang-up state is a terminal state with at least one arc that is not in idle status. Necessary and sufficient conditions for a simulation not to terminate in a hang-up state are given, and models for which a simulation terminates normally are proven to be determinate. The main problem to be solved arises in proper connections of loop junctions and loop output operators. Rodríguez also defines the transformations which can be performed on graphs to obtain equivalent graphs.

In Luconi's computational schemata, [LUCO 67, LUCO 68], the variables, i.e., the set of arcs U , only have values associated with them. An operator w_k has a set of transformations $I_{k'} \rightarrow O_{k'}$ which maps values associated with the incoming arcs to values associated with the outgoing arcs. At a given time of the simulation, an operator

w_k is applicable if there is at least one transformation made possible by the values associated with members of I_k . A simulation starts by setting some initial values on members of U , a program history is the time-ordered sequence of values appearing on all the arcs.

In this model there exists some arcs $I \subseteq U$ which connect the entity represented by the model (e.g., a special-purpose computer, a systems program, etc.) to its environment. This set of arcs is called the interface. Determinacy in Luconi's models may either be as defined earlier—he calls it complete determinacy—or only on the arcs belonging to the interface, i.e., output determinacy. Local rules for (total) determinacy of computational schemata are reminiscent of Bernstein's conditions ([BERN 66] and Section III.B) for automatic detection of parallelism, that is:

- The computational schema must be conflict-free, i.e., if any two operators w_i and w_j become simultaneously applicable, the values associated with any connecting arc are altered by at most one operator (cf. $O_i \cap O_j = \Phi$).
- The computational schema must be transformation lossless, i.e., if any two operators w_i and w_j become simultaneously applicable, the execution of w_i does not affect the results to be produced by w_j , and vice-versa (cf. $I_i \cap O_j = \Phi$, $I_j \cap O_i = \Phi$).

There is no procedure to determine if any arbitrary partially interpreted schema is determinate, but there exists an effective procedure to determine the equivalence of two determinate schemata.

In the computational schema $\mathcal{P}$ above, the control C was determined by the values associated with the members of U . A special class of computational schemata, that Luconi calls structural schemata [LUCO 68, LUCO 68A], have their control defined by an acyclic directed graph $C(N, V)$ and an enabling function associating control nodes n_i with functional nodes w_i by a mapping $I_i \rightarrow O_i$. By means of connections of structural schemata through their interfaces, "complex operators" similar to Rodriguez's may be constructed. There exists an effective

procedure to test the equivalence of two complex operators.

In a recent paper [BRUN 71], this study of control networks is continued. The computational operations on the data are coordinated by a control flow network which itself is partitioned into functional building blocks. Five basic control modules, i.e., nodes in the network, are defined: WYE (two-way fork); JUNCTION (two-way join); SEQUENCE; ITERATION; and SELECT (predicate). The interaction between data and control flow structures is through those arcs of the control network which were part of the interface as defined in the previous model. The environment of the control flow is able to change the status of an input arc of the interface only from idle to active, and the status of an output arc from active to idle (idle and active being the only possible arc states). The thrust of the paper is to determine the necessary and sufficient conditions for well-formedness (i.e., absence of deadlocks) of the network. To do so the behavior diagrams of each basic module type are defined. They consist of infinite acyclic periodic directed graphs which represent the states of the modules, i.e., the status of their incoming and outgoing arcs. In fact, the "period" or section graph of a behavior diagram is sufficient to characterize a module. Well-formedness is proved, in general, by construction. Equivalence of networks (e.g., equivalence between an n -way WYE and $(n-1)$ basic WYE) is also considered.

These models, particularly Luconi's and Bruno's, are important for the study and the design of macromodular computer systems such as those proposed by Clark, et al. (see Appendix and [CLARK 67]).

D. Models Introducing Data Queueing on the Arcs

In the previous models when the arcs were representative of data values, a single data value was associated with an arc at a given state of the simulation. Queues of data could be modeled, for example by Petri nets, but subgraphs had to be created. A similar statement applies for the modeling of structured data. The first instance of the introduction of first-in-first-out (FIFO) queues instead

of single variables is found in Karp and Miller's first model [KARP 66].

The basic structure of this program model $\Phi = (W, U, C)$ is a directed graph $G(W, U)$ where the vertices w_i represent computations and the arcs $u_p = (w_i, w_j)$ represent FIFO queues of data. Each arc u_p has 4 attributes: A_p which gives the number of words initially in the FIFO queue; U_p' and W_p' which give, respectively, the number of data words added (removed) to (from) the queue when $0, (0,)$ —the operation represented by $w_i(w_j)$ —terminates (is initiated); and T_p which is the threshold value ($T_p \geq W_p'$) of the minimum queue length to initiate w_j (note that by making $U_p' = W_p' = 1$, we have a Petri net).

The control C corresponds to an *AND*-input-*AND*-output logic as defined in part B of this section. Thus, a weakness of this model is that data-dependent conditional transfers cannot be taken into account. The operation associated with w_j can be initiated if and only if for each arc u_p incident into w_j , the number of words in the queue associated with u_p is not less than T_p . After initiation of w_j , W_p' words are removed from each arc u_p leading into w_j . When w_j terminates, U_p' words are placed on each branch u_q incident out of w_j . These constraints can be formalized with the aid of the following definitions:

Let E be a sequence of non-empty sets $S_1, S_2, \dots, S_N, \dots$ where each $S_N \in \{1, 2, \dots, |W|\}$. Let $x(j, 0) = 0$ and for $N > 0$ let $x(j, N)$ be the number of sets S_m , $l \leq m \leq N$ of which j is an element.

E is said to be an execution of (i.e., a simulation on) G if for all N :

- 1) if $j \in S_{N+1}$ and $u_p = (w_i, w_j) \in U$, then $A_p + U_p'x(i, N) - W_p'x(j, N) \geq T_p$, i.e., for w_j to initiate for the $x(j, N+1)$ th time, the queue lengths on its input arcs must not be less than the respective threshold values.
- 2) if E is finite and of length P , then for each j there exists a node w_i and an arc $u_p = (w_i, w_j)$ such that: $A_p + U_p'x(i, P) - W_p'x(j, P) < T_p$, i.e., the execution terminates since there exists no node which can be initiated. E is a proper execution

- 3) if $A_p + U_p'x(i, N) - W_p'x(j, N) \geq T_p$ for every arc $u_p = (w_i, w_j)$, then $j \in S_p$ for $P > N$, i.e., if a node is able to initiate it will do so after a finite number of initiations of other nodes.

Graphs which have proper executions are proven to be determinate. This requires that the operation represented by a given node is performed an equal number of times independently of the proper execution. An algorithm is given to compute this number of performances. Necessary and sufficient conditions for queue lengths to remain bounded are also defined.

This model has been further studied by Reiter [REIT 68, REIT 68A]. His principal result is the determination of the maximum number of locations which must be reserved for a given computation; it is shown that, in general, a parallel system is more economical storage-wise than a sequential one. By simplifying the model in supposing $T_p = W_p' = 1$ and $U_p' \in \{0, 1\}$, and by adding a time attribute for each arc $u_p = (w_i, w_j)$ defined as the time required by vertex w_i to place its U_p' words on the arc u_p , Reiter demonstrates the existence of a schedule of the computation process which is optimal with respect to both computation speed and ease of control. An algorithm for the construction of such a schedule is also given.

Adams' model [ADAM 68, ADAM 70] is completely interpreted and is closer to a real programming language than any of the models surveyed in this section. It allows not only queues of data but also the possibility of their being structured, as well as the definition of recursive procedures. The basis of this model is again a directed graph $G(W, U)$ called a graph program. Its nodes are either primitive or procedure nodes; the differences between their roles will be explained below. To begin with, let us consider a graph program composed uniquely of primitive nodes. The arcs joining these nodes represent values of FIFO queues of data, each element of the queue being either a simple entity or an ordered set. The possible data types encountered in the graph are being defined separately by a context-free grammar. Primitive nodes may be either computational (*r*-nodes), that is, they map

values on the incoming arcs into values on the outgoing arcs as in some of the models defined so far, or both computational and logical (s-nodes) by also mapping arc status (locked or unlocked), as in Rodriguez's model. Now procedure nodes consist of a subgraph program itself composed of primitive/procedure nodes and a set of interface arcs whose data types must be compatible with their respective counterparts in the calling graph program. It is to be noted that this definition of a procedure graph is similar to that of an ALGOL-60 procedure, the interface arcs playing the role of parameters. Recursive procedures are thus allowed.

Another definition of a graph program is $H(G, X)$ where $G(W, U)$ is the "main" graph program as described in the above paragraph and X is the recursively defined set of all procedure graphs encountered in G or in procedure nodes called from G .

The simulation on Adams' model is activated in the following manner. With each node in the graph is associated an initiation FIFO queue, say IQ_i , for w_i . IQ_i is the representation of the activations of w_i , which have not yet been terminated. The FIFO ranking assures that execution of w_i will be terminated in the same order in which they have been initiated, a necessary condition for determinacy. A primitive node is ready for initiation if the queues attached to its unlocked incoming arcs are not empty. Further, in the case of an s-node, no other execution of w_i must be in progress; that is, IQ_i must also be empty. The execution of w_i consists of performing the computational/logical mappings particular to that node, removing one element from each input queue, assigning the computation to some uniquely determined processor, and adding an element to IQ_i . The termination of w_i places one element on each output queue, resets the arcs' status, and removes the first element of IQ_i . In the case of a procedure node, initiation requires that a copy of the graph procedure be added to the graph program. Termination implies that, in the procedure graph, no nodes are ready for initiation and no nodes have been initiated but not terminated. Termination of a procedure node may imply ordering of data (bracket-

ing) and removal of the copy of the graph procedure from the system. As can be seen, the graph can dynamically expand and contract in the same way as an ALGOL-60 program. More than one copy of a particular r -node may be in execution simultaneously, thus allowing greater parallelism.

Adams' principal result is that his model is determinate and has the universal computing capability; that is, any computation which can be performed on a Turing machine can be modeled by a graph program.

The models in this section differ from those already reviewed mostly by the fact that they are completely interpreted. In fact, program graphs may be used as a programming language, although only as a representation language at the present time. The introduction of queueing on arcs is also important since it allows multiple simultaneous executions of a given node.

E. Parallel Program and Flow Graph Schemata

The models we survey now are completely uninterpreted. As in Luconi's computational schema the data flow and the control flow structures are separated. Queues of data are associated with operators. We start by describing Karp and Miller's second model which forms the basis for the models described below [KARP 67, KARP 69]. Their parallel program schema $\mathcal{P} = (W, U, C)$ is defined by:

- U , a set of memory locations;
- W , a set of operations w_i such that;
 - K_i is the number of outcomes of w_i ;
 - $D_i \subseteq U$ is the set whose elements are the domain locations for w_i ;
 - $R_i \subseteq U$ is the set whose elements are the range locations for w_i ;
- $C = (Q, q_0, \Sigma, \tau)$ is a control with
 - Q a set of states;
 - $q_0 \in Q$ an initial state;
 - $\Sigma = \bigcup_{w_i \in W} \{\bar{w}_i, w_i^1, \dots, w_i^{K_i}\}$ the alphabet, $\bar{w}_i$ being an initiation symbol and the w_i^j being termination symbols;
 - $\tau: Q \times \Sigma \rightarrow Q$ a transition function such that $\tau(q, \bar{w}_i)$ may be undefined but all $\tau(q, w_i^j)$ are necessarily defined.

C can also be visualized as a directed graph $C(Q, \Sigma)$ where the states are nodes

and members of the alphabet are labelled arcs.

An operation represented by w_i can be initiated only when a state q for which $\tau(q, \bar{w}_i)$ is defined has been reached. When w_i is initiated it obtains input elements from locations from D_i , and when w_i is terminated it places output elements in locations from R_i and chooses an outcome $w_{i'}$ where $1 \leq j \leq K_i$.

More formally, in order to define a simulation on $\mathcal{P}$, one has to define an interpretation of $\mathcal{P}$. Let v be a function associating with each element u_i of U , a set $V(u_i)$. Let V_0 be the initial contents of the memory U . For each operation w_i , two functions are defined, namely:

- F_i , which is a mapping of members of D_i into members of R_i ; and
- G_i , which selects a $w_{i'}$ as an outcome of w_i .

Now a $\mathcal{P}$ -state (or instantaneous description of $\mathcal{P}$) is a tuple $\alpha = (V, q, \mu)$ where V is the set of values associated with each element of U , q is the present control state and μ is the set of queues of elements of D_i attached to each operation w_i . The initial $\mathcal{P}$ -state is $\alpha_0 = (V_0, q_0, \mu_0)$ where μ_0 is a set of empty queues. The rules for the sequencing of operations are defined through next $\mathcal{P}$ -state functions, that is, functions of the present $\mathcal{P}$ -state and of one member of Σ such that the transition function is defined for that element. One distinguishes between:

- An initiation of operation w_i , i.e., $\alpha \cdot \bar{w}_i$, defined only if $\tau(q, \bar{w}_i)$ is defined. Then $\alpha \cdot \bar{w}_i = (V', q', \mu')$ where $V' = V$, $q' = \tau(q, \bar{w}_i)$ and the only queue changed is $\mu_{i'}$ which is μ_i appended, in a FIFO manner, with the tuple containing $V(u_j)$ for each $u_j \in D_i$.
- A termination of w_i , i.e., $\alpha \cdot w_{i'}$ where for each $u_j \notin R_i$, $V'(u_j) = V(u_j)$, for $u_j \in R_i$, $V'(u_j)$ is the component of F_i corresponding to u_j , $q = \tau(q, w_{i'})$ and all queues are left unchanged except for $\mu_{i'}$ which is μ_i deleted of its first element.

As can be seen from these rules, operations have to be terminated in the order in which they were initiated (see Adams' model in the previous section).

In the above model there is no restriction

on the number of states in the control. Karp and Miller, and later Slutz [SLUT 68], have studied subsets of the parallel program schemata that they call, respectively, (parallel) flowcharts and flow graph schemata. Now the state of the control is given by the contents of p counters, that is, $Q = \{q \mid q \in N^p\}$ where N denotes the non-negative integers. Associated with each element $w_{i'}$ of the alphabet Σ , i.e., with each labelled arc of the control graph, is a vector that specifies the values of counter increments (from -1 to N). This mapping is such that all operations once initiated will terminate [KARP 69], or equivalently all terminations of an operation have the same set of input counters [SLUT 68]. Further, for each operation there exists some counter which will be incremented during initiation and decremented during a corresponding termination, and will be unaffected by all other operations. Such a counter in fact stores the number of elements in the queue associated with each operator w_i . If V is the function $\Sigma: (\{-1\} \cup N)^p$ specifying the value of counter increments and decrements for each operation w_i , then the transition

$$(q, w_{i'}) = \begin{cases} q + V(w_{i'}) & \text{if } q + V(w_{i'}) \geq 0 \\ \text{undefined otherwise.} \end{cases}$$

In order to have determinate flowcharts, four restrictions are imposed:

- Persistence, i.e., once an operator is ready to be initiated it must remain in this state. More formally, if σ and π belong to Σ , then if $\tau(q, \sigma)$ and $\tau(q, \pi)$ are defined, then $\tau(q, \sigma\pi)$ and $\tau(q, \pi\sigma)$ must also be defined.
- Commutativity. If $\tau(q, \sigma\pi)$ and $\tau(q, \pi\sigma)$ are defined then $\tau(q, \sigma\pi) = (q, \pi\sigma)$. It is to be noted that $\tau(q, \sigma\pi)$ or $\tau(q, \pi\sigma)$ may be undefined only if σ or π is of the form $\bar{w}$, that is an initiation symbol.
- Losslessness. The set R_i attached to each operator must be non-empty.
- Commutativity on next state function. If $\alpha \cdot \sigma\pi$ and $\alpha \cdot \pi\sigma$ are defined, then $\alpha \cdot \sigma\pi = \alpha \cdot \pi\sigma$.

Another important subclass is that of repetition-free schemata, which are such that if an operator w_i is executed twice, each variable in D_i must appear in the R_i of an

operator w , which is executed between the two executions of w . Then it can be proved that it is decidable whether:

- A finite-state schema (i.e., one with a finite Q set) is repetition-free.
- A repetition-free flowchart is a finite state schema.
- A given operator w , is terminating in a repetition-free finite state schema. Also, if w , is terminating one can find the maximum number of initiations of w , in a computation.
- A repetition-free lossless persistent commutative finite state schema is determinate.

A schema is said to be bounded if at every step in the simulation of the schema the length of each of the queues μ_i is bounded by some integer K . If $K = 1$ the schema is said to be serial.

It is decidable whether a given repetition-free finite state schema is bounded. However it is undecidable whether:

- Two persistent finite state schemata are equivalent.
- Two serial finite state schemata are equivalent.

Additional work on this model is continued, in particular in the area of finding those schemata which display maximum parallelism [KELL 70].

F. Other Models and Current Research

As can be seen by the above review, modeling for parallel processing may take different directions. Other approaches to the representation of parallel programs or systems exist. For example, flow tables similar to those used in sequential circuit theory form the basis of Bredt's model [BRED 70] for the description and analysis of the control of parallel systems.

Investigations in this field are still very active. Their importance for the design of multiprocessors does not have to be stressed, but it is also worthwhile to note that they have added a great deal of comprehension to the logical design and correctness of multiprogrammed operating systems, which in fact involve the synchronization of parallel processes and the scheduling of independent resources. Current and future research in

modeling should introduce more explicitly realistic constraints such as the means to represent and allocate resources efficiently. Work in this direction is already in progress, mostly in extending the properties of the models described in part B above. At the same time work on program schemata—both interpreted and uninterpreted—should continue in order to get a better understanding of the equivalence of parallel and sequential processes, as well as for the definition of programming languages for parallel processing.

V. PREDICTION OF PERFORMANCE

Most of the multiprocessor systems may not have yet been built or marketed. However, during the design stage some prediction of the increase in performance when passing from uniprocessing to multiprocessing has to be made to justify the extra expense. In the Appendix we give a trivial example—the multiplication of two matrices—of an evaluative comparison of a uniprocessor, an array processor and a “maximally parallel” multiprocessor. In Section III we reviewed some experiments measuring the efficiency of the automatic detection of parallelism. In this section we show how the use of models can help in assessing the projected performance of some systems. We shall consider three areas of investigation, namely:

- The use of analytic models where the prediction of performance is derived through queueing theory or graph model analysis.
- The gross level simulation where experiments are performed on models such as those reviewed in the previous section. It may imply a rather large departure from reality.
- The low-level simulation where a set of particular programs is run on a model very close to the actual design of the system.

A. Analysis

Queueing Theory

Coffman has investigated homogeneous multiprocessors from the queueing theory

viewpoint [COFF 66, COFF 67]. His main assumptions are:

- Poisson arrivals with an average arrival rate of λ .
- Exponential service time with an average execution time of $1/\mu$.
- c processors in parallel thus yielding an utilization factor of $\rho = \lambda/c\mu$.
- The jobs are composed of k subjobs. There is no assumption about the distribution $h(k)$ of these jobs other than the mean m_k and variance σ_k^2 of the number of subjobs are finite.
- The jobs wait in queue in a FIFO basis.

Based on these assumptions, the mean number $\bar{n}$ of subjobs in the system and the mean time w_q spent waiting in the queue for the first subjob of a job-arrival are derived for three disciplines:

- 1) Multiprocessing and multiprogramming. When a processor becomes free it takes the first subjob waiting in the queue. There can be a mix in the system of subjobs belonging to different jobs.
- 2) Multiprogramming. When a processor becomes free, it takes the first full job in the queue, and, therefore, we have a degradation of the above policy, i.e., $k = 1$.
- 3) Multiprocessing. There is no possibility of mix of subjobs from different jobs in the system.

The main results, relevant to our multiprocessing studies are:

- In case 1, $\bar{n}$ is greater for c processors than for $c/2$ processors with average execution times twice as great; that is, a better throughput is realized by distributing the computing power. The difference decreases when m_k increases.
- Comparing cases 1 and 2, the waiting time w_{q_2} is longer than w_{q_1} . The difference $w_{q_2} - w_{q_1}$ increases with loading while the ratio $(w_{q_2} - w_{q_1})/w_{q_1}$ decreases at the same time.

The throughput rate which is the maximum departure rate of jobs is $c\mu/m_k$ for discipline 1). For discipline 3), it is $1/m_o$ where m_o is the mean operation time of jobs and this value is smaller than the one obtained with discipline 1). A comparison between their respective values shows that the

second tends asymptotically to the first for large values of m_k but that for small values of m_k discipline 3) is comparatively very disadvantageous.

The first analytical study on pipe-line systems was performed by Aoki, et al. [AOKI 63] with a pipe-line of two computers separated by a buffer of size N . Two values of N were considered, namely:

- $N = 1$, i.e., the first machine cannot begin its $(i + 1)$ th computation until the second one has finished its i th.
- $N = \infty$, i.e., the first machine can proceed with its $(i + 1)$ th computation as soon as it has finished its i th.

For $N = 1$, the following uniform distribution functions of the computational times on the two processors were examined.

- Identical.
- The time interval of distribution 2 is included in the time interval of distribution 1.
- The lower bound of distribution 2 is equal to the mean of distribution 1 and the time intervals are equal.

It was found that:

$$R = \frac{\text{Time to compute } n \text{ computations on two machines}}{\text{Time to compute } n \text{ computations on one machine}}$$

increases with each above case when n becomes large. When $N \rightarrow \infty$, the analysis becomes rapidly intractable.

Kleinrock [KLEI 66] considers the same problem, but with N arbitrary. However, the distribution of computing time is taken as exponential. By modeling the system as a cyclic queue, R is derived first, in the case of two processors; then, in the general case of an arbitrary number of processors, by splitting the processors in successive groups of equal importance. As could be expected, the best system is a balanced one in which each processor has the same average processing time.

In [LEVY 65] the congestion of modular systems is studied. These are composed of n processors and $(k + 1)$ memories and are called (n, k) systems. One of the memories is occupied exclusively by an executive func-

tion; the other k memories hold jobs. The executive assigns memories, i.e., jobs, to processors. If the memory is already busy at assignment time, the processor waits in a FIFO queue until the memory gets free. When a processor finishes a job, it returns to the executive, stands in queue if necessary, enters the executive program, and is assigned a new job (if any available).

Memories are assigned to processors independently of the state of the system and with equal probability. The distribution functions of both the running times of the problems and of the executive (i.e., arrivals) are exponential. The steady-state flow equations are solved in (2, 3) and (3, 3) cases assuming that the time spent in the executive is wasted. Very little increase in absolute performance is gained by using three processors instead of two. Only a slight improvement is noted when the executive time is considered as useful.

By using more banks in the memory, that is increasing k , one can assume no congestion and therefore an exact analysis, as a single service closed queue, becomes possible. For a given n , processor utilization vs. executive load factor can be plotted.

We now turn our attention to the prediction of performance using graph models of the PERT-type or d.a.b. graphs (see Section III.B).

Graph Model Analysis

The primary assumptions here are that we model the programs with an acyclic directed graph $G(W, U)$ where attributes such as time and frequency of execution, deadlines, penalties etc. are associated with the computations, i.e., the vertices $\{w_i\}$. These computations have to be scheduled on a set of N (generally) identical processors with some specific function to be optimized. The structure of the computational tasks, the nature of precedence constraints and, the optimizing function provide the main criteria for differentiation. Few analytical results are known and we report some of them here, leaving heuristic methods for the next subsection.

The first solutions can generally be found in the field of operations research, since the

problem on hand is similar to the one of assigning partially ordered tasks on a limited number of processors. Thus, linear and dynamic programming solutions such as those used to solve job-shop and assembly-line assignment and sequencing problems could be used here; but the amount of computation required becomes rapidly un-surmountable for problems that are even rather small. However, if one (or more) of the three criteria mentioned above is severely restricted, some interesting results can be obtained. We adopt the following notation for the attributes of w_i , namely:

- t_i , time of execution of w_i on any machine;
- τ_i , deadline;
- p_i , penalty if the deadline is exceeded;
- $b_i = p_i/t_i$, the loss rate; and
- $l_i = \max(0, (T + t_i - \tau_i)p_i)$ the (linear) loss of task w_i when started at time T .

We shall note also T_G the time of execution of the total graph and L_G the total loss.

McNaughton [MCNA 59] was the first to obtain some results for special cases, namely that all m tasks were independent. Then:

- If $N = 1$ and all τ_i are equal, an optimal schedule for minimizing L_G is found by sequencing the task in order of decreasing b_i .
- If $N \geq 1$ and all $\tau_i = 0$, there is no need to split tasks on several machines in order to achieve an optimal schedule for L_G ; T_G is given by:

$$T_G \geq \max(t_1, t_2, \dots, t_m, \sum_{i=1}^m t_i/N)$$

These results have been generalized [ROTH 66] by considering losses of the form $b_i e^{-r t_i}$. Then on a single processor the sequencing is optimal if done in decreasing order of $b_i e^{-r t_i} / (1 - e^{-r t_i})$. Also, in the multiprocessor case there is no need for splitting the tasks. A dynamic programming solution to the case of m independent tasks to be assigned and scheduled on N processors is also given.

The next level in complexity is to go from a set of independent tasks to a set of tasks for which the graph of precedence relationship is a tree. The basic algorithm due to Hu [HUTC 61] gives an optimal solution for the minimization of T_G when all t_i are the

same. It has been extended for multicom-mensurable t_i by Muntz [MUNT 70]. However, in this latter paper tasks can be preempted before completion, or, in other words, a processor can be shared by several tasks during a given time period. For both algorithms the basic idea is to schedule first those tasks which are farthest (timewise) from the root of the tree and carry on recursively the "cutting of the longest branches" until the root has been reached. With the same restrictions on precedence relationships, Bowdon [BOWD 69] has generalized McNaughton's results by deriving an algorithm minimizing L_G when the losses are linear. The sequencing priorities are based on the set of successor tasks and their respective losses. All the algorithms discussed in this paragraph are nonoptimal when the graph of precedence relationship is not a tree, as can be shown easily through counter-examples.

Very little is known about optimal schedules for tasks linked by general precedence relationships, and this area is open for further investigation. A first breakthrough in this direction has been accomplished by Muntz [MUNT 69] for the minimization of T_G on two processors with a preemptive allocation strategy. Unhappily, this algorithm is not generalizable to a larger number of processors. Similarly, Hu's algorithm for minimizing T_G on a tree has been extended for an acyclic directed graph when the tasks, of equal time, are processed on two processors [FUJI 69, COFF 72]. Again, the algorithm is not optimal when the number of processors is greater than two or when the tasks are of unequal time. For the latter problems, one has to recourse to dynamic programming solutions [RAMA 72].

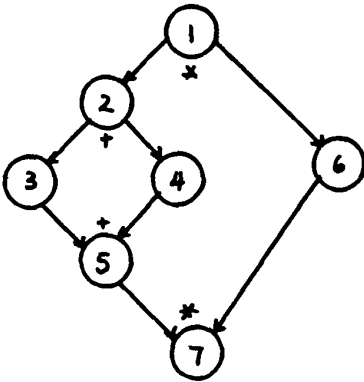
But, if optimal schedules cannot be found, at least some bounds can be obtained. For example, a lower bound on T_G is given by the critical path method. Some more interesting results due to Graham [GRAH 66, GRAH 72] are as follows. Let S be a schedule, i.e., a fixed assignment and sequencing of tasks on N processors. Now T_G and $T_{G'}$ are the times of processing graphs $G(W, U)$ and $G'(W', U')$, where $W = W'$ and $U' \subseteq U$ (i.e., some precedence constraints may have

been eliminated) on N and N' processors, respectively, where $t_i' \leq t_i$ and where the schedules S and S' may be different; then:

$$\frac{T_{G'}}{T_G} \leq 1 + \frac{N-1}{N'}$$

A first corollary is that given any schedule for N processors, the best improvement is by a factor of $2 - 1/N$. Second, if $N = N'$, $S = S'$ and some $t_i' \leq t_i$, one can get a larger $T_{G'}$; and also for all $t_i = t_i'$, $N' > N$ and $S' = S$, one can have $T_{G'} > T_G$, that is, an increase in the number of processors may result in an increase in the total running time! Thus, one can see that care must be taken when scheduling tasks heuristically. Rules for stabilizing schedules to prevent such anomalies have been established by Manacher [MANA 67].

The above discussion was restricted to acyclic graph models. However, two features found in almost all programs were not represented, namely cyclic behavior and branching conditions. Let us look first at the latter problem. When branching is taken into account, there is no possibility of having one optimal schedule, since different data sets may generate different paths. According to the terminology of Section IV, our model is a d.a.b. graph, and optimal schedules should be found for each AND-type subgraph. Because even simple computations such as that of the critical paths of each subgraph are not feasible in terms of time, the concept of mean path length had to be introduced [MART 69]. For this, a probability is assigned to each path; hence, each arc $u_k = (w_i, w_j)$ is assigned a probability attribute q_i , and the probability of an AND-type subgraph is the product of the probabilities of the arcs belonging to it. Now if P_A is the probability of subgraph A , and T_A is its execution time according to a given schedule for the whole graph, then the mean path length is defined by $\bar{T}_G = \sum_{A \in G} P_A T_A$. Martin [MART 69] extended some previous work on graph reduction procedures [ELMA 64] to compute an optimistic, i.e., negatively biased, value of the mean path length when the number of processors is not restricted. Figure 16 shows an example d.a.b. graph, vertex and arc attributes, a (simplified)



Vertex	1	2	3	4	5	6	7
Time attribute	1	2	3	6	1	7	1

Arc probabilities: $q_2^3 = \frac{1}{3}$, $q_2^4 = \frac{2}{3}$,
all other $q_1^j = 1$.

Mean Path length Computations:

$$PL_1 = 1;$$

$$PL_2 = PL_1 + 2 = 3;$$

$$PL_3 = (PL_2 + 3) \times \frac{1}{3} = 2;$$

$$PL_4 = (PL_2 + 6) \times \frac{2}{3} = 6;$$

$$PL_5 = PL_3 + PL_4 + 1 = 9;$$

$$PL_6 = PL_1 + 7 = 8$$

$$PL_7 = \max(PL_5, PL_6) + 1 = 10.$$

True Mean Path Length:

$$\begin{aligned} TPL &= t_1 + t_7 + \frac{1}{3} \max(t_2 + t_3 + t_5, t_6) + \frac{2}{3} \max(t_2 + t_4 + t_5, t_6) \\ &= 10 + \frac{1}{3} \end{aligned}$$

Fig 16. Mean path length computation on d.a.b. graphs.

mean path length computation, as well as the real mean path length. On the same type of graph one can determine an upper bound on the maximum number of processors required so that no computational task will ever be delayed [BAER 69]. It is readily seen that in the example of Figure 16 two processors are sufficient.

For the previous analysis to approximate reality one has to give time attributes which are reflective of the program being modeled. Since the model is an acyclic graph representation of a cyclic process, one way to define these time attributes is to transform the graph in such a way that the mean path length of the acyclic structures, obtained by removing the feedback arcs, is equal to the mean path length of the original structures,

when the latter are considered by themselves and not embedded in more complex structures. This implies the determination of the frequency of execution of each vertex when one knows the probabilities, or mean number of traversals, of the feedback arcs. Analytical solutions to this problem can be found in [MART 67A, BAER 68A, BOVE 70].

B. Gross Level Simulation

Since analytical results are difficult to obtain from models such as those described above, some heuristic methods have been devised to estimate the performance of programs to be run on multiprocessing systems. The most common objective is to predict the throughput time T_o as a function of the number of processors N according to differ-

ent scheduling strategies. We limit ourselves to this particular measure, but it is worthwhile to note that some investigators have also looked at the problem of memory allocation [CODD 62, TURN 63, BOVE 70]. We shall use the following terminology:

- An urgency rule assigns to each waiting entity (job, task, vertex) a scalar value, the highest total among entities waiting at a facility (machine, processor) determines the entity to be selected over all others for scheduling.
- A scheduling rule determines which entity among those waiting for service will be scheduled as having priority over others.

Note that the set of urgency rules is a subset of the set of scheduling rules. We add the following definitions:

- A one-shot scheduling is a process which, following the scheduling rules, assigns and sequences tasks on processors in a feasible fashion, such that no task is reconsidered for scheduling once it has been assigned and sequenced.
- An iterative scheduling is a process which, starting on an initial feasible schedule, iteratively displaces tasks until a convergence condition is observed.

The basis of a one-shot scheduling process is a three-step algorithm, namely:

- 1) See if any processor is available. If not go to step 3.
- 2) Schedule the first waiting process, if any, on the first available processor. Go to step 1.
- 3) Advance the "clock" by the minimum amount of time which will make at least one processor available. Go to step 1.

The terminal value of the clock (initialized to 0) will be T_G .

When dealing with an acyclic directed graph with time attributes associated with vertices some of the urgency rules used by different authors are:

- 1) Smallest time of operation first.
- 2) Longest time of operation first.
- 3) Random selection.
- 4) First-In-First-Out.
- 5) Last-In-First-Out.
- 6) The smallest name (index) associated with the job first.

- 7) Greatest number of immediate successors first.
- 8) Greatest number of successors first.
- 9) The maximum of the sum of the processing times of all successors first.
- 10) The greatest level, i.e., maximum number of arcs leading from the vertex to the terminal vertex, first.
- 11) A combination of some of the preceding.

As can be seen, the first six urgency rules are local in the sense that they depend only on past history, or on information carried by the tasks which are candidates for immediate assignment. By contrast, the remaining five rules imply a pre-processing of the graph.

One of the first simulations of this type was performed by McKenney [MCKE 62] who explored the gain in segmenting a program into subtasks linked by precedence relationships, each subtask having specific functions. He ran extensive simulations of a system composed of one or two groups of one CPU and eight specialized I/O processors, using a one-shot scheduling with a First-In-First-Out urgency rule.

In one paper, Schwartz [SCHW 61] used a one-shot algorithm for sequencing a program (no branching vertices) for which all tasks had already been pre-assigned on machines of different characteristics. Within one clock period, i.e., at the availability of a machine, all possible tasks which could be sequenced on that machine were scheduled, with the urgency placing the task with the greatest number of successors first (ties were broken by a smallest operation first rule). Note that, possibly because of the heterogeneity of the system, the urgency rule which is chosen depends very little on the time of execution of a vertex, and depends mostly on the structure of the graph. In a second paper [SCHW 64], the case of two homogeneous processors with only some of the vertices being preassigned was investigated. The same scheduling rule was used with the same urgency rule as above for preassigned vertices; for ties and for those vertices which can be assigned on either machine, the urgency rule was longest time of operation first. In both cases, the results

of these heuristic algorithms were very close to optimal schedules.

In a series of studies that culminated in Martin's dissertation [MART 66, MART 67B] the UCLA group implemented an a priori scheduling with a directed acyclic bilogic graph as a starting point. The function to minimize was the mean path length of the distorted graph. An iterative scheduling was performed in the following fashion.

- 1) Perform an initial assignment by equalizing the number of vertices assigned to each processor.
- 2) Perform an initial sequencing on each processor, grouping the vertices by their number of predecessors, and within these groups applying an urgency rule defined below.
- 3) Reassign vertices singly and resequence with the same policy. If the mean path length is reduced, keep the new schedule, otherwise restore the previous one.

Step 3 above was performed successively for all vertices for a given iteration. The urgency rule chosen was "urgency number proportional to the length of the longest path from the vertex to the terminal vertex." The number of iterations had to be kept low because of the time-consuming procedure.

The main weaknesses of this approach lie in the fact that a) the criterion function, i.e., the mean path length, is not very reliable when a large distortion is present in the graph (small number of processors); b) the computing time of an iteration is proportional not only to the number of vertices but also to the number of processors; and c) the vertex exchange as described above is limited to the rescheduling of one vertex at a time (or possibly a cluster) from one processor to another although it could be more advantageous to swap several vertices.

To circumvent these defects, a one-shot scheduling algorithm was implemented [BAER 68A]. The principal difference with those scheduling processes presented earlier was that the structure of the d.a.b. graph, that is, the presence of branching vertices and of mutually exclusive tasks, was taken into account in the scheduling rule. When a task was assigned to a machine, all other

tasks which were candidates for assignment and mutually exclusive to it—and mutually exclusive between themselves—were scheduled at the same time. With global urgencies involving path length computations from the vertex on hand to the terminal vertex, it was found that the one-shot scheduling algorithm outperformed the iterative algorithm, and thus could give a better basis for the realization of dynamic schedules.

With the advent of computer networks the scheduling problem is becoming more important. Algorithms optimal in restricted cases cannot be applied without study to general cases. For example, we have performed some experiments on a well-known graph, Manacher's map, using Bowdon's algorithm [BOWD 69] and a straightforward one-shot algorithm, and found the latter to be superior. Both analytical tools and experiments on heuristic methods are needed to advance the state of the art in this particular field.

C. Fine Level Simulation

In the previous section, estimates of performance were crude because both the hardware systems and the modeling of processes were idealized. For example, running times were expressed as mean values, no overhead was taken into account for processor switching and scheduling, etc. If one desires to perform simulations at finer levels, then computational time, i.e., cost, is going to increase radically. This is why, in general, one is restricted to simulating those parts of the system in which bottlenecks are most likely to arise (they may have been detected by grosser level experiments), or to testing the efficiency of a particular algorithm on a specific machine configuration.

The simulation of a multiprocessor system differs only in detail from that of a general multiprogramming system [MACD 70]. Difficulties more akin to multiprocessing simulation are discussed by Hutchinson [HUTC 68]. They emerge mostly from the fact that one has to be concerned with an additional set of resources, the CPU's, and their scheduling. For example, since more than one CPU may require interaction with

a single resource such as a disk, scheduling anomalies may arise because of the inability of the simulation program clock to differentiate between completion times of routines competing for that resource. Thus, in the simulation a high priority process may appear to be holding that particular resource far more often than in reality. To alleviate the problems created by simultaneous event occurrences, simulation programming languages should contain statements dealing with possible logical relationships between processes. Experts in this field are still debating as to whether there exists a high-level programming language which can be truly labelled as a simulation language for parallel processors.

A good illustration of the help given by simulation in the design stage of a multiprocessor system is the description of the decision process involved in determining the configuration of a computing facility for air traffic control [MERI 68]. The system comprised multiple CPU's, I/O control elements, storage elements, as well as peripheral adapter modules and tape controller units. The various elements could be mixed to provide different levels of computing and storage capacity. The simulation had several objectives, but its main thrust was to determine system performance, under different designs, in terms of job throughput and resource utilization. According to the authors of the paper, simulation is the only tool which can determine the best configuration when there exists problems such as simultaneous competition among processes for access to a common data base or for storage element memory cycles.

In the same paper some gross assumptions still had to be made because the system had to be tested for a large variety of load conditions. Programs were not simulated instruction by instruction but on a block basis. If one wants to emulate a multiprocessor on a single processor, then either the emulated machine must be an extension of the host computer, or the number of algorithms to be analyzed has to be severely reduced.

An application of the former type can be found in [DRAU 67] where simulations of FORTRAN-like programs to be run on a

parallel machine are discussed. The host machine is a CDC 6600 and the emulated system is composed of up to 60 CDC's 6600's organized as an ATHENE machine (see Appendix and [SCHW 66]). The source language is a direct extension of FORTRAN called PFORTRAN (see Section II.B). The application programs simulated included matrix multiplication, internal sorts, two-point boundary value problems for the solution of ordinary differential equations, Monte-Carlo simulations, etc. It was found that parallel programs written in PFORTRAN are not significantly more complex than their sequential equivalents, and that substantial gains can be attained mostly by loop replication.

At the other end of the spectrum, there exist numerous studies of specific algorithms specially geared to machines with parallel configurations. For example, Rosenfeld [ROSE 69] studied a program for computing current flows through an electrical network containing only resistors. The assumed machine was a parallel processing system resembling a multiple CPU IBM System/360. The number of processors and the number of storage modules, as well as the number of variables in the problem, were allowed to vary independently. The factors studied were the optimal number of processors for a given number of variables, the effects of storage interference and the resources utilization. Similar studies are reported in Lehman [LEHM 66].

When testing a particular algorithm on a particular configuration, the algorithm itself cannot be adapted directly from its serial form. Modifications to existing algorithms have to be performed or new methods devised in order to take advantage of the parallelism of the machine's architecture. The study of such processes is outside the scope of this paper and the reader is referred to a recent survey [MIRA 71].

VI. CONCLUSION

In this paper we have surveyed several aspects of multiprocessing: language features

to exploit multiprocessing; automatic detection of parallelism; models for parallel computation; and prediction of performance of multiprocessors. The amount of research being carried out and the volume of literature published recently on these topics attest the continuing interest in this field; our bibliography is already slightly out-of-date.

The imminent deliveries of ILLIAC IV and STAR-100, as well as the pipe-line computers of Texas Instruments ASC and the British MU5 (see Appendix), show the feasibility of array and pipe-line processors. Techniques for writing compilers generating efficient code for these types of machine organization are in development. General and special-purpose multiprocessors, homogeneous and nonhomogeneous, are considered for a large number of applications. Operating systems for these machines require firm theoretical background for protection, cooperation, and scheduling of processes. Extension of Dijkstra's semaphores, their hardware implementation, and the lock-out problem are current areas of investigation. New methods for automatic detection of parallelism at the procedural and assembly language levels have appeared. Measurements of their effectiveness on a large sample of sequential programs are needed. On the theoretical front, the modeling of computations by graph models, and the numerous problems deriving from it, are still very much in progress. Current trends seem to be to model resources as well as computations on the one hand, and on the other hand to look at the general problem of uninterpreted graphs. Finally, optimal assignment and sequencing of computations on parallel processors are still open questions and only some restricted cases have been solved.

Multiprocessors are now part of the current generation of computer systems. As often happens in the computer field, hardware realizations have appeared sooner than expected, and progress in efficient software and intelligent applications of the raw computing power are lagging. It is not difficult to predict that intensive efforts will be made to remedy this situation.

Acknowledgment

I am deeply grateful to one of the referees who not only helped in the reorganization of this paper, but who also improved its style by an order of magnitude.

BIBLIOGRAPHY

- [ABEL 69] ABEL, N. E.; D. P. BUDNICK; D. J. KUCK; Y. MURAOKA; R. NORTHCOTE; AND R. B. WILHELMSON. "TRANQUIL: a language for an array processing computer." In *Proc. AFIPS 1969 Spring Joint Computer Conf.*, AFIPS Press, Montvale, N.J., 57-73.
- [ADAM 68] ADAMS, D. A. "A computation model with data flow sequencing." PhD Dissertation, Computer Science Dept., Stanford Univ., Stanford, Calif., 1968.
- [ADAM 70] ADAMS, D. A. "A model for parallel computations." In *Parallel processor systems, technologies and applications*, L. C. Hobbs, et al. (Eds.), Spartan Books, New York, 1970, 311-334.
- [ALLA 64] ALLARD, R. W.; K. A. WOLK; AND R. A. ZEMLIN. "Some effects of the 6600 computer on language structure." *Comm. ACM* 7, 2 (Feb. 1964), 112-119.
- [ANDE 62] ANDERSON, J. P.; S. A. HOFFMAN; J. SHIFMAN; AND R. J. WILLIAMS. "DS25—A multiple computer system for command and control." In *Proc. AFIPS 1962 Fall Joint Computer Conf.*, Spartan Books, New York, 86-96.
- [ANDE 65] ANDERSON, J. P. "Program structures for parallel processing." *Comm ACM* 8, 12 (Dec. 1965), 786-788.
- [ANDE 67] ANDERSON, D. W.; F. J. SPORACIO; AND R. M. TOMASULO. "Machine philosophy and instruction handling." *IBM J. of Research and Development* 11 (Jan. 1967), 8-24.
- [AOKI 63] AOKI, M.; G. ESTRIN; AND R. MANDELL. "Analysis of computing-load assignment in a multiprocessor computer." *AFIPS 1963 Fall Joint Computer Conf.*, Spartan Books, New York, 147-160.
- [ASCH 67] ASCHENBRENNER, R. A.; M. J. FLYNN; AND G. A. ROBINSON. "Intrinsic multiprocessing." In *Proc. AFIPS 1967 Spring Joint Computer Conf.*, AFIPS Press, Montvale, N.J., 81-86.
- [BAER 68] BAER, J. L.; AND D. P. BOVET. "Compilation of arithmetic expressions for parallel computations." In *Proc. IFIP Congress 1968*, North-Holland, Amsterdam, The Netherlands, 340-346.

- [BAER 68A] BAER, J. L. "Graph models of computations in computer systems." PhD Dissertation, Dept. of Electrical Engineering, Univ. California, Los Angeles, Calif, 1968.
- [BAER 69] BAER, J. L., AND G. ESTRIN. "Bounds for maximum parallelism in a biologic graph model of computations." *IEEE Trans on Computers* C-18, (Nov. 1969), 1012-1914.
- [BAER 70] BAER, J. L.; D. P. BOVET; AND G. ESTRIN. "Legality and other properties of graph models of computations." *J. ACM* 17, 3 (July 1970), 543-552.
- [BALL 62] BALL, J. R.; R. C. BOLLINGER; T. A. JEEVES, R. C. McREYNOLDS, AND D. H. SHAFTER. "On the use of the SOLOMON parallel processing computer." In *Proc AFIPS 1962 Fall Joint Computer Conf.*, Spartan Books, New York, 137-146.
- [BARN 68] BARNES, G.; R. BROWN, M. KATO; D. KUCK; D. SLOTNICK; AND R. STOKES. "The ILLIAC IV computer." *IEEE Trans on Computers* C-17 (Aug. 1968), 746-757.
- [BERN 66] BERNSTEIN, A. J. "Analysis of programs for parallel processing." *IEEE Trans. on Electronic Computers* EC-15 (Oct 1966), 757-762.
- [BING 67] BINGHAM, H. W.; D. A. FISHER; AND E. W. REIGEL. "Automatic detection of parallelism in computer programs." TR-ECOM-02463-4, Burroughs TR-67-4, AD 622 274, Nov. 1967.
- [BLAA 64] BLAAUW, G. A. "The structure of system/360 Part V Multisystem organization." *IBM Systems J* 3, (1964), 181-195.
- [BLAK 67] BLAKENEY, G. R., L. F. CUDNEY; AND C. R. EICKHORN. "Design characteristics of the 9020 system." *IBM Systems J.* 6 (1967), 90-94.
- [BONS 69] BONSEIGNEUR, P. "Description of the 7600 computer system." *Computer Group News* 2 (May 1969), 11-15.
- [BOVE 69] BOVET, D. P. "Legality of multi-logic graphs." *Istituto di Automatica*, University of Roma, Dec 1969.
- [BOVE 70] BOVET, D. P.; AND G. ESTRIN. "A dynamic memory allocation algorithm." *IEEE Trans. on Computers* C-19 (May 1970), 403-411.
- [BOWD 69] BOWDON, E. K. "Priority assignment in a network of computers." *IEEE Trans on Computers* C-18 (Nov 1969), 1021-1026.
- [BRED 70] BREDT, T. H. "A survey of models for parallel computing." Tech. Report #8, Stanford Electronic Laboratories, Stanford Univ, Stanford, Calif, Aug 1970.
- [BRED 70A] BREDT, T. H., AND E. J. McCLUSKEY. "Analysis and synthesis of control mechanisms for parallel processes." In *Parallel processor systems, technologies and applications*, L. C. Hobbs, et al. (Eds.), Spartan Books, New York, 1970, 287-296.
- [BRIG 64] BRIGHT, H. S. "A Philco multi-processing system." In *AFIPS 1964 Fall Joint Computer Conf*, Part II, Spartan Books, New York, 97-141.
- [BROW 62] BROWN, G. W. "A new concept in programming." In *Computers and the world of the future*, M. Greenberger (Ed), MIT Press, 1962, 250-287.
- [BRUN 71] BRUNO, J., AND S. M. ALTMAN. "A theory of asynchronous control networks." *IEEE Trans on Computers* C-20 (June 1971), 629-638.
- [BUCH 62] BUCHHOLZ, W. (Ed.) *Planning a computer system project Stretch*, McGraw-Hill, New York, 1962.
- [BURN 67] BURNETT, G. J.; L. J. KOZELA; AND R. A. HOKOM. "A distributed processing system for general purpose computing." In *Proc. AFIPS 1967 Fall Joint Computer Conf*, AFIPS Press, Montvale, N.J., 757-768.
- [BUSS 62] BUSSELL, B. "Properties of a variable structure computer system in the solution of parabolic partial differential equations." PhD Dissertation, Dept. of Electrical Engineering, Univ California, Los Angeles, Calif, 1962.
- [CAMP 69] CAMPEAU, J. O. "The block-oriented computer." *IEEE Trans on Computers* C-18 (Aug. 1969), 706-718.
- [CAMP 70] CAMPEAU, J. O. "Communication and sequential problems in the parallel processor." In *Parallel processor systems, technologies and applications*, L. C. Hobbs, et al (Eds), Spartan Books, New York, 1970, 215-234.
- [CERF 72] CERF, V. G. "Multiprocessors, semaphores and a graph model of computations." PhD Dissertation, Computer Science Dept., Univ California, Los Angeles, Calif, 1972.
- [CHAM 71] CHAMBERLIN, D. D. "The single-assignment approach to parallel processing." In *Proc AFIPS 1971 Fall Joint Computer Conf*, AFIPS, Press, Montvale, N.J., 263-269.
- [CHEN 71] CHEN, T. C. "Parallelism, pipelining and computer efficiency." *Computer Design* 10 (Jan 1971), 69-76.
- [CHEN 72] CHEN, Y. E.; AND D. L. EPLEY. "Memory requirements in a multi-processing environment." *J. ACM* 19, 1 (Jan 1972), 57-69.
- [CLAR 67] CLARK, W. A. "Macro-modular computer systems." In *Proc AFIPS 1967 Spring Joint Computer Conf.*, AFIPS Press, Montvale, N. J., 333-336.
- [Codd 62] Codd, E. F. "Multiprogram-

- ming" In *Advances in computers*, vol 3, F Alt and M. Rubinoff (Eds.), 1962, 78-153
- [COFF 66] COFFMAN, E G "Stochastic models of multiple and time-shared computer operations" PhD Dissertation, Dept of Electrical Engineering, Univ. California, Los Angeles, Calif., 1966
- [COFF 67] COFFMAN, E G. "Bounds on parallel processing of queues with multiple jobs." *Nav Res Log Q* 14 (Sept 1967), 345-366.
- [COFF 72] COFFMAN, E G, AND GRAHAM, R L "Optimal scheduling for two processor systems" *Acta Informatica* 1 (1972), 200-213
- [COMF 63] COMFORT, W T. "A modified Holland machine" In *Proc. AFIPS 1963 Fall Joint Computer Conf*, Spartan Books, New York, 481-488.
- [COMM 71] COMMONER, R, A W HOLT; S. EVEN, AND A PNUCLI "Marked directed graphs" *J. Computer and System Sciences* 5 (1971), 511-523
- [CONS 68] CONSTANTINE, L L "Control of sequence and parallelism in modular programs" In *Proc. AFIPS 1968 Spring Joint Computer Conf*, AFIPS Press, Montvale, N J, 409-414
- [CONT 71] CONTROL DATA STAR COMPUTER SYSTEM, Hardware manual 602 56 000, Revision 01, 1971
- [CONW 63] CONWAY, M. "A multiprocessor system design" In *Proc AFIPS 1963 Fall Joint Computer Conf*, Spartan Books, New York, 139-146
- [CORB 65] CORBATÓ, F J.; AND V. A VISOTSKY. "Introduction and overview of the MULTICS system" In *Proc AFIPS 1965 Fall Joint Computer Conf*, Part I, Spartan Books, New York, 185-196.
- [COTT 65] COTTEN, L W "Circuit implementation of fast pipeline systems" In *Proc AFIPS 1965 Fall Joint Computer Conf*, Part I, Spartan Books, New York, 489-504
- [CRIT 63] CRITCHLOW, A J "Generalized multiprocessing and multiprogramming systems." In *Proc. AFIPS 1963 Fall Joint Computer Conf.*, Spartan Books, New York, 107-126
- [CURT 63] CURTIN, W. A "Multiple computer systems" In *Advances in computer*, vol. 4, F Alt and M Rubinoff (Eds.), 1963, 245-303
- [DAVI 69] DAVIS, R L "The ILLIAC IV processing element" *IEEE Trans. on Computers* C-18 (Sept 1969), 800-816
- [DENN 66] DENNIS, J. B, AND E C VAN HORN "Programming semantics for multiprogrammed computations." *Comm. ACM* 9, 3 (March 1966), 143-155.
- [DENN 68] DENNIS, J. B "Programming generality, parallelism and computer architecture." In *Proc. IFIP Congress 1968*, North Holland, Amsterdam, The Netherlands, C1-C7.
- [DENN 70] DENNIS, J. B. "Modular, asynchronous control structures for a high-performance processor" *Record of the Project MAC Conference on Concurrent Systems and Parallel Computation*, 55-80, 1970
- [DIJK 65] DIJKSTRA, E. W. "Solution of a problem in concurrent programming control." *Comm ACM* 8, 9 (Sept. 1965), 569-570
- [DIJK 68] DIJKSTRA, E. W. "Co-operating sequential processes." In *Programming languages*, F. Genuys (Ed.), Academic Press, New York, 1968, 43-112.
- [DRAU 67] DRAUGHON, E.; R. GRISHMAN; J. SCHWARTZ; AND A. STEIN. "Programming considerations for parallel computers." *New York University IMM* (Nov 1967), 362.
- [DREI 58] DREIFUS, P. "System design of the GAMMA-60." In *Proc. 1958 Western Joint Computer Conf*, Spartan Books, New York, 130-133
- [DREI 58A] DREIFUS, P. "Programming design features for the GAMMA-60 computer." In *Proc. 1958 Eastern Joint Computer Conf*, Spartan Books, New York, 174-180.
- [ECKE 59] ECKERT, J. P.; J. C. CHU, A. B. TONIK; AND W. F. SCHMITT. "Design of UNIVAC-LARC system. I" In *Proc. 1959 Eastern Joint Computer Conf.*, Spartan Books, New York, 59-65.
- [ELLI 71] ELLIS, C. A. "Parallel compiling techniques." In *Proc. ACM 26th Nat. Conf.*, ACM, New York, 1971, 508-519.
- [ELMA 64] ELMAGHRABY, S. E. "An algebra for the analysis of generalized activity networks." *Management Science* 10 (April 1964), 494-514
- [ESTR 60] ESTRIN, G. "Organization of computer systems: the fixed-plus-variable structure computer." In *Proc. 1960 Western Joint Computer Conf*, Spartan Books, New York, 33-40.
- [ESTR 63] ESTRIN, G.; B. BUSSELL, R. TURN, AND J. BIBB "Parallel processing in a restructurable computer system." *IEEE Trans. on Electronic Computers* EC-12 (Dec. 1963), 747-754.
- [ESTR 63A] ESTRIN, G.; AND R. TURN. "Automatic assignment of computations in a variable structure computer system." *IEEE Trans. on Electronic Computers* EC-12 (Dec. 1963), 756-773.
- [FATE 69] FATEMAN, R. J. "Optimal code for serial and parallel computation." *Comm. ACM* 12 (1969), 694-695
- [FIRE 71] FIRESTONE, R. H. "Parallel pro-

- gramming: operational model and detection of parallelism." PhD Dissertation, Depart. of Mathematics, New York Univ., 1971.
- [FISH 67] FISHER, D. A. "Program analysis for multiprocessing." Master's Thesis, Univ. Pennsylvania, Philadelphia, Pa., May 1967.
- [FLYN 66] FLYNN, M. J. "Very high speed computing systems." *Proc IEEE* 54, (Dec. 1966), 1901-1909.
- [FLYN 70] FLYNN, M. J.; A. POVGIN; AND K. SHIMIZU. "A multiple instruction stream processor with shared resources." In *Parallel processor, technologies, and applications*, L. C. Hobbs, et al. (Eds.), Spartan Books, New York, 1970, 251-286.
- [FLYN 71] FLYNN, M. J. "Shared internal resources in a multiprocessor." In *Proc IFIP Congress 1971*, North Holland, Amsterdam, The Netherlands, 565-569.
- [FUJI 69] FUJII, M.; T. KASAMI; AND K. NINOMIYA. "Optimal sequencing of two equivalent processors." *SIAM J. Applied Math* 17 (July 1969), 784-789.
- [GERT 70] GERTZ, J. L. "Hierarchical associative memories for parallel computations." PhD Dissertation, Massachusetts Institute of Technology, 1970.
- [GIBS 66] GIBSON, C. T. "Time-sharing in the IBM system/360 model 67." In *Proc. AFIPS 1966 Spring Joint Computer Conf*, Spartan Books, New York, 61-78.
- [GILB 72] GILBERT, P.; AND W. J. CHANDLER. "Interference between communicating parallel processes." *Comm. ACM* 15, 6 (June 1972), 427-437.
- [GILL 58] GILL, S. "Parallel programming." *Computer J.* 1, (April 1958), 2-8.
- [GONZ 63] GONZALES, R. "A multilayer iterative circuit computer." *IEEE Trans on Electronic Computers* EC-12 (Dec. 1963), 781-790.
- [GONZ 70] GONZALEZ, M. J.; AND C. V. RAMAMOORTHY. "Recognition and representation of parallel processable streams in computer programs." In *Parallel processor systems, technologies and applications*, L. C. Hobbs, et al. (Eds.), Spartan Books, New York, 1970, 335-374.
- [GONZ 71] GONZALEZ, M. J., AND C. V. RAMAMOORTHY. "Program suitability for parallel processing." *IEEE Trans on Computers* C-20 (June 1971), 647-654.
- [GOSD 66] GOSDEN, J. A. "Explicit parallel processing description and control in programs for multi and uniprocessor computers." In *Proc. AFIPS 1966 Fall Joint Computer Conf*, Spartan Books, New York, 651-660.
- [GOST 71] GOSTELOW, K. P. "Flow of control, resource allocation and the proper termination of programs." PhD Dissertation, Computer Science Dept., Univ. California, Los Angeles, Calif., 1971.
- [GRAH 66] GRAHAM, R. L. "Bounds for certain multiprocessing anomalies." *Bell System Tech. J.* 45, (Nov. 1966), 1563-1581.
- [GRAH 69] GRAHAM, R. L. "Bounds on multiprocessing timing anomalies." *SIAM J. Applied Math.* 17 (1969), 416-429.
- [GRAH 72] GRAHAM, R. L. "Bounds on multiprocessing anomalies and related packing algorithms." In *Proc. AFIPS 1972 Spring Joint Computer Conf*, AFIPS Press, Montvale, N.J., 205-217.
- [GREG 63] GREGORY, J.; AND R. McREYNOLDS. "The SOLOMON computer." *IEEE Trans on Electronic Computers* EC-12 (Dec 1963), 774-781.
- [HABE 72] HABERMANN, N. A. "Synchronization of communicating processes." *Comm. ACM* 15, 3 (March 1972), 171-176.
- [HELL 61] HELLER, J. "Sequencing aspects of multiprogramming." *J. ACM* 8, 3 (July 1961), 426-439.
- [HELL 66] HELLERMAN, J. "Parallel processing of algebraic expressions." *IEEE Trans. on Electronic Computers* EC-15 (Jan 1966), 82-91.
- [HOBB 70] HOBBS, L. C.; AND D. J. THEIS. "Survey of parallel processor approaches and techniques." In *Parallel processor systems, technologies and applications*, L. C. Hobbs, et al. (Eds.), Spartan Books, New York, 1970, 3-20.
- [HOLL 59] HOLLAND, J. H. "A universal computer capable of executing an arbitrary number of subprograms simultaneously." In *Proc. 1959 Eastern Joint Computer Conf*, Spartan Books, New York, 108-113.
- [HOLT 68] HOLT, A. W., et al. *Information system theory project*. Applied Data Research Inc., 1968.
- [HOLT 69] HOLT, A. W., AND F. COMMONER. *Information system theory project*. Applied Data Research Inc., 1969.
- [HUTC 61] HU, T. C. "Parallel sequencing and assembly line problems." *Operations Research* 9 (Nov-Dec 1961), 841-848.
- [HUTC 68] HUTCHINSON, G. K. "Some problems in the simulation of multiprocessor computer systems." In *Simulation programming languages*, J. N. Buxton (ed.), North-Holland, Amsterdam, The Netherlands, 1968, 305-318.
- [IBBE 72] IBBETT, R. N. "The MU5 instruction pipeline." *Computer J.* 15 (Feb. 1972), 42-50.
- [IBMC 69] PL/I Language Specifications. IBM Corporation, 1969, Form Y33-6003-1.
- [IRWI 70] IRWIN, J. D., AND J. M. THORINGTON, JR. "Multiprocessing bibli-

- ography." Auburn Univ. *Project THEMIS Report*, Number AU-T-12, July 1970.
- [IVER 62] IVERSON, K. E. "A programming language." John Wiley & Sons, New York, 1962.
- [KARP 66] KARP, R. M.; AND R. E. MILLER. "Properties of a model for parallel computations. determinacy, termination, queuing." *SIAM J. Applied Math.* 14 (Nov. 1966), 1390-1411.
- [KARP 67] KARP, R. M.; AND R. E. MILLER. "Parallel program schemata: a mathematical model for parallel computation." In *IEEE Conference Record of the 8th Annual Symposium on Switching and Automata Theory*, Oct. 1967, 55-61.
- [KARP 69] KARP, R. M.; AND R. E. MILLER. "Parallel program schemata." *J. Computer and System Sciences* 3 (May 1969), 147-195.
- [KATZ 66] KATZ, J. H. "Simulation of a multiprocessor computer system." In *Proc. AFIPS 1966 Spring Joint Computer Conf*, Spartan Books, New York, 127-139.
- [KELL 70] KELLER, R. M. "On maximally parallel schemata." *IEEE Conference Record of the 11th Annual Symposium on Switching and Automata Theory*, Oct. 1970, 32-50.
- [KLEI 66] KLEINROCK, L. "Sequential processing machines (SPM) analyzed with a queueing theory model." *J. ACM* 13, 2 (April 1966), 179-193.
- [KNUT 66] KNUTH, D. "Additional comments on a problem in concurrent programming control." *Comm ACM* 9, 5 (May 1966), 321-322.
- [KOCZ 68] KOCZELA, L. J. "The distributed processor organization." In *Advances in computers*, vol. 9, F. Alt and M. Rubinoff (Eds.), 1968, 285-353.
- [KOTO 68] KOTOV, A. E.; AND A. S. MARINGANI. "On transformation of sequential programs into asynchronous parallel programs." In *Proc IFIP Congress 1968*, North-Holland, Amsterdam, The Netherlands, J37-J45.
- [KRAS 72] KRASKA, P. W. "Parallelism exploitation and scheduling." PhD Dissertation, Dept. of Computer Science, Univ. Illinois, Urbana, Ill., 1972.
- [KUCK 68] KUCK, D. J. "ILLIAC IV software and applications programming." *IEEE Trans on Computers* C-17 (Aug. 1968), 758-770.
- [KUCK 72] KUCK, D. J.; Y. MURAOKA; AND S. C. CHEN. "On the number of operations simultaneously executable in FORTRAN-like programs and their resulting speed up." *IEEE Trans on Computers* C-21 (Dec. 1972).
- [LAMP 68] LAMPSON, B. W. "A scheduling philosophy for multiprocessing systems." *Comm. ACM* 11, 5 (May 1968), 347-360.
- [LEHM 66] LEHMAN, M. "A survey of problems and preliminary results concerning parallel processing and parallel processors." *Proc. IEEE* 54 (Dec. 1966), 1889-1901.
- [LEIN 59] LEINER, A. L.; W. A. NOTZ; J. L. SMITH; AND A. WEINBERGER. "PILOT—A new multiple computer system." *J.ACM* 6, 3 (July 1959), 313-335.
- [LEVY 65] LEVY, W. A. "Congestion in a multi-modular computer system." Pennsylvania Research Associates, April 1965.
- [LINC 71] LINCOLN, N. R. "Languages for parallel thinkers." *SIGPLAN Symposium, Languages for System Implementation*, Purdue Univ., Lafayette, Ind., 1971.
- [LUCO 67] LUCONI, F. L. "Completely functional asynchronous computational structures." *IEEE Conference Record of the 8th Annual Symposium on Switching and Automata Theory*, Oct. 1967, 62-70.
- [LUCO 68] LUCONI, F. L. "Asynchronous computational structures." PhD Thesis, Massachusetts Institute of Technology, 1968.
- [LUCO 68A] LUCONI, F. L. "Output functional computational structures." *IEEE Conference Record of the 9th Annual Symposium on Switching and Automata Theory*, Oct. 1968, 76-84.
- [MACD 70] MACDOUGALL, M. H. "Computer system simulation: an introduction." *Computing Surveys* 2, 3 (Sept. 1970), 191-210.
- [MANA 67] MANACHER, G. K. "Production and stabilization of real-time task schedules." *J.ACM* 14, 3 (July 1967), 429-465.
- [MART 66] MARTIN, D. F. "The automatic assignment and sequencing of computations on parallel processor systems." PhD Dissertation, Dept. of Electrical Engineering, Univ. California, Los Angeles, Calif., 1966.
- [MART 67] MARTIN, D. F.; AND G. ESTRIN. "Models of computations and systems—evaluation of vertex probabilities in graph models of computations." *J.ACM* 14, 2 (April 1967), 281-299.
- [MART 67A] MARTIN, D. F.; AND G. ESTRIN. "Models of computational systems—cyclic to acyclic graph transformations." *IEEE Trans. on Electronic Computers* EC-16 (Feb. 1967), 70-79.
- [MART 67B] MARTIN, D. F.; AND G. ESTRIN. "Experiments on models of computations and systems." *IEEE Trans. on Electronic Computers* EC-16 (Feb. 1967), 59-69.
- [MART 69] MARTIN, D. F.; AND G. ESTRIN. "Path length computations on graph models of computations."

- IEEE Trans. on Computers* C-18 (June 1969), 530-536.
- [MAVA 71] MAVADDAT, F. "Using stacks to detect expression parallelism." In *Proc IFIP Congress 1971*, North Holland, Amsterdam, The Netherlands, 591-596.
- [McKE 62] McKENNEY, J. L. "Simultaneous processing of jobs on an electronic computer." *Management Science* 8 (April 1962), 344-354.
- [McNA 59] McNAUGHTON, R. "Scheduling with deadlines and loss functions." *Management Science* 6 (Jan. 1959), 1-12.
- [MERI 68] MERIKALLIO, R. A.; AND F C HOLLAND. "Simulation design of a multiprocessing system." In *Proc AFIPS 1968 Fall Joint Computer Conf.*, AFIPS Press, Montvale, N.J., 1399-1410.
- [MIRA 71] MIRANKER, W. L. "A survey of parallelism in numerical analysis." *SIAM Review* 13 (Oct 1971), 524-547.
- [MULL 59] MULLER, D. E.; AND W. S. BARTKY. "A theory of asynchronous circuits." *Proceedings of an International Symposium on the Theory of Switching*, Vol. 29 of the *Annals of the Computation Laboratory of Harvard Univ.*, 204-243, 1959.
- [MUNT 69] MUNTZ, R. R.; AND E. G. COFFMAN. "Optimal preemptive scheduling on two-processor systems." *IEEE Trans. on Computers* C-18 (Nov 1969), 1014-1020.
- [MUNT 70] MUNTZ, R. R.; AND E. G. COFFMAN. "Preemptive scheduling of real-time tasks on multiprocessor systems." *J.ACM* 17, 2 (April 1970), 324-338.
- [MURA 71] MURAOKA, Y. "Parallelism exposure and exploitation in programs." PhD Dissertation, Dept. of Computer Science, Univ Illinois, Urbana, Ill., Feb. 1971.
- [MURT 66] MURTHA, J. C. "Highly parallel information processing systems." In *Advances in Computers*, Vol. 7, F. Alt. and M. Rubinoff (Eds.), 1966, 2-116.
- [NOEJ 71] NOE, J. D. "A Petri net model of the CDC 6400." *Proceedings of the ACM/Sproops Workshop on Systems Performance Evaluation*, April 1971, 362-378.
- [NUTT 72] NUTT, G. J. "The formulation and application of evaluation nets." PhD Dissertation, Computer Science Group, Univ. Washington, Seattle, Wash., 1972.
- [OPLE 65] OPLER, A. "Procedure oriented language statements to facilitate parallel processing." *Comm ACM* 8, 5 (May 1965), 306-307.
- [ORNS 67] ORNSTEIN, S. M.; M. J. STUCKI; AND W. A. CLARK. "A functional description of macromodules." In *Proc. AFIPS 1967 Spring Joint Computer Conf.*, AFIPS Press, Montvale, N.J., 333-357.
- [PATI 70] PATIL, S. S. "Coordination of asynchronous events." PhD Dissertation, Massachusetts Institute of Technology, 1970.
- [PETR 62] PETRI, C. A. "Kommunikation mit automaten." Translated in *Project MAC-M-212 Report*, Originally published in 1962.
- [PORT 60] PORTER, R. E. "The RW-400—a new polymorphic data system." *Dalation* 6 (Jan. 1960), 8-14.
- [PRIS 66] PRISTKER, A. B.; AND W. W. HAPP. "GERT graphic evaluation and review techniques, part I. fundamentals." *J. of Industrial Engineering* 17 (May 1966), 267-274.
- [RAMA 66] RAMAMOORTHY, C. V. "Analysis of graphs by connectivity consideration." *J.ACM* 13, 2 (April 1966), 211-223.
- [RAMA 69] RAMAMOORTHY, C. V.; AND M. J. GONZALEZ. "A survey of techniques for recognizing parallel processable streams in computer programs." In *Proc. AFIPS 1969 Fall Joint Computer Conf.*, AFIPS Press, Montvale, N.J., 1-15.
- [RAMA 69A] RAMAMOORTHY, C. V.; AND M. J. GONZALEZ. "Recognition and representation of parallel processable streams in computer programs II (task/process parallelism)." In *Proc. ACM 24th Nat. Conf.*, ACM, New York, 1969, 387-397.
- [RAMA 71] RAMAMOORTHY, C. V.; AND M. J. GONZALEZ. "Subexpression ordering in the execution of arithmetic expressions." *Comm. ACM* 14, 7 (July 1971), 479-486.
- [RAMA 72] RAMAMOORTHY, C. V.; K. M. CHANDY, AND M. J. GONZALEZ. "Optimal scheduling strategies in a multiprocessor system." *IEEE Trans on Computers* C-21 (Feb. 1972, 137-146.
- [REGI 72] REGIS, R. C. "Systems of concurrent processes. structural properties and stochastic analysis," PhD Dissertation, Computer Science Dept Johns Hopkins Univ., Baltimore, Md., 1972.
- [REIG 69] REIGEL, E. W. "Parallelism exposure and exploitation in digital computing systems." PhD Dissertation, Univ Pennsylvania, Philadelphia, Pa., 1969.
- [REIG 70] REIGEL, E. W. "Parallelism exposure and exploitation." In *Parallel processor systems, technologies and applications*, L. C. Hobbs, et al (Eds.), Spartan Books, New York, 1970, 417-438.
- [REIT 68] REITER, R. "A study of a model for parallel computation." PhD Dissertation, Dept. of Computer Science, Univ. Michigan, Ann Arbor, Mich., 1968.
- [REIT 68A] REITER, R. "Scheduling parallel

- computations." *J.ACM* 15, 4 Ann Arbor, Mich., (Oct. 1968), 590-599
- [RODR 67] RODRIGUEZ, J. E. "A graph model for parallel computations." PhD Thesis, Massachusetts Institute of Technology, 1967.
- [ROSE 69] ROSENFELD, J. L. "A case study in programming for parallel processors." *Comm. ACM* 12, 12 (Dec. 1969), 645-655.
- [ROTH 66] ROTHKOPF, M. A. "Scheduling independent tasks on parallel processors." *Management Science* 12, (Jan. 1966), 437-447.
- [RUGG 69] RUGGIERO, J. F.; AND D. A. CORYELL "An auxiliary processing system for array calculations." *IBM Systems J.* 8 (1969), 118-135.
- [RUSS 69] RUSSELL, E. C. "Automatic program analysis." PhD Dissertation, Dept. of Electrical Engineering, Univ. California, Los Angeles, Calif., 1969.
- [SALT 66] SALTZER, J. H. "Traffic control in a multiplexed computer system." PhD Dissertation, Massachusetts Institute of Technology, 1966.
- [SCHW 61] SCHWARTZ, E. S. "An automatic sequencing procedure with applications to parallel programming." *J.ACM* 8, 4 (Oct. 1961), 512-537.
- [SCHW 64] SCHWARTZ, E. S. "A heuristic procedure for parallel processing." *Management Science* 10 (July 1964), 767-777.
- [SCHW 66] SCHWARTZ, J. "Large parallel computers." *J.ACM* 13, 1 (Jan. 1966), 15-32.
- [SENZ 65] SENZIG, D. N.; AND R. V. SMITH. "Computer organization for array processing." In *Proc. AFIPS 1965 Fall Joint Computer Conf.*, Part I, Spartan Books, New York, 117-128.
- [SHAP 70] SHAPIRO, R. M.; AND H. SAINT. "A new approach to optimization of sequencing decisions." In *Annual Review Automatic Programming*, Vol. 6, Part 5, 1970.
- [SLOT 62] SLOTTICK, D. L.; W. C. BORCK; AND R. McREYNOLDS "The Solomon computer." In *Proc. AFIPS 1962 Fall Joint Computer Conf.*, Spartan Books, New York, 97-107.
- [SLUT 68] SLUTZ, D. R. "The flow graph schemata model of parallel computation." PhD Dissertation, Massachusetts Institute of Technology, 1968.
- [SQUI 63] SQUIRE, J. S.; AND S. M. PALAIS. "Programming and design considerations of a highly parallel computer." In *Proc. AFIPS 1963 Spring Joint Computer Conf.*, Spartan Books, New York, 395-400.
- [SQUI 63A] SQUIRE, J. S. "A translation algorithm for multiple processor computer." In *Proc. ACM 18th Nat. Conf.*, ACM, New York, 1963.
- [STAN 67] STANGA, D. C. "UNIVAC 1108 multiprocessor system." In *Proc. AFIPS 1967 Spring Joint Computer Conf.*, AFIPS Press, Montvale, N.J., 67-74.
- [STON 67] STONE, H. S. "One-pass compilation of arithmetic expressions for a parallel processor." *Comm. ACM* 10, 4 (April 1967), 220-223.
- [STON 70] STONE, H. S. "A pipeline push down stack computer." In *Parallel processors systems, technologies and applications*, L. C. Hobbs, et al. (Eds.), Spartan Books, New York, 1970, 235-249.
- [TESL 68] TESLER, L. G.; AND H. G. ENEA. "A language design for concurrent processes." In *Proc. AFIPS 1968 Spring Joint Computer Conf.*, AFIPS Press, Montvale, N.J., 402-408.
- [THOR 64] THORNTON, J. E. "Parallel operation in the Control Data 6600." In *Proc. AFIPS 1964 Fall Joint Computer Conf.*, Part II, Spartan Books, New York, 33-40.
- [THOR 67] THORLIN, J. F. "Code generation for PIE (parallel instruction execution)." In *Proc. AFIPS 1967 Spring Joint Computer Conf.*, AFIPS Press, Montvale, N.J., 642-644.
- [TJAD 70] TJADEN, J. S.; AND M. J. FLYNN. "Detection and parallel execution of independent instructions." *IEEE Trans. on Computers* C-19 (Oct. 1970), 889-895.
- [TOMA 67] TOMASULO, R. M. "An efficient algorithm for exploiting multiple arithmetic units." *IBM J. of Research and Development* 11, (Jan. 1967), 25-33.
- [VANW 69] VAN WIJNGAARDEN, A.; B. J. MAILLOUX; J. E. L. PECK; AND C. H. A. KOSTER "Report on the algorithmic language ALGOL 68." *Mathematisch Centrum, M R 101*, Amsterdam, (Oct. 1969).
- [VANH 66] VAN HORN, E. C. "Computer design for asynchronously reproducing multiprocessing." PhD Thesis, Massachusetts Institute of Technology, 1966.
- [VOLA 70] VOLANSKY, S. A. "Graph model analysis and implementation of computational sequences." PhD Dissertation, Dept. of Electrical Engineering, Univ. California, Los Angeles, Calif., 1970.
- [WARS 62] WARSHALL, S. "A theorem on Boolean matrices." *J.ACM* 3, 1 (Jan. 1962), 11-12.
- [WATS 72] WATSON, W. J. "The Texas Instrument advanced scientific computer." *COMPCON 72 Digest* (1972), 291-294.
- [WIRT 66] WIRTH, N. "A note on program structures for parallel processing." *Comm. ACM* 9, 5 (May 1966), 320-321.
- [WIRT 69] WIRTH, N. "On multiprogramming, machine coding and com-

puter organization." *Comm. ACM*
12, 9 (Sept. 1969), 489-498.

APPENDIX: HARDWARE SYSTEMS

In this brief Appendix we classify multi-processors into four categories by answering two questions: Is the system homogeneous or nonhomogeneous? Is the system general or special-purpose?

This leads us to consider the main subdivisions: general-purpose homogeneous processors, general-purpose nonhomogeneous processors, array computers, and pipe-line computers. Other taxonomies have appeared such as Flynn's [FLYN 66] based on the unicity or multiplicity of instruction and data streams.

A. Homogeneous Multiprocessors

The ideal machine, called IT by Murtha [MURT 66], would consist of N identical processors (N being as large as one wishes), each having a full instruction set and capabilities of performing asynchronously. Furthermore, each processor could route its results to any other without any loss of time. There would be no storage problems, nor indexing, fetching, storing, etc., time penalties. Therefore, IT is only a means of representing the parallelism inherent in a program.

In comparing IT to a uniprocessor, with the same simplifying timing assumptions, we can consider two measures as relevant: the time speed-up T , which we define as:

$$T = \frac{\text{Time to complete the task on uniprocessor}}{\text{Time to complete the task on IT}} = \frac{T_{\text{UNI}}}{T_{\text{IT}}}$$

and the occupancy factor F , that is:

$$F = \frac{\Sigma \text{Time spent in each processor}}{N \times T_{\text{IT}}}$$

For example, consider the multiplication of two matrices, $A(p, n)$ by $B(n, q)$ giving $C(p, q)$. Evidently $T_{\text{UNI}} = png(t_m + t_a)$ where t_m and t_a are, respectively, the multi-

plication and addition times. Now if one builds an IT of pnq elements (for the pnq multiplications) and then applies a log-sum process for the additions (we suppose that each processor can add only two numbers at a time), then: $T_{\text{IT}} = t_m + Mt_a$ with $M = \lceil \log_2 n \rceil$, ($\lceil x \rceil$ is the smallest integer greater than or equal to x) and

$$T = \frac{png(t_m + t_a)}{t_m + Mt_a}$$

While each processor is busy during the multiplication and the first addition, half of them are idle in the next addition, then half of the remainder in the next addition, and so on, thus yielding:

$$F = \frac{Nt_m + Nt_a \sum_{i=0}^M 2^{-i}}{N(t_m + Mt_a)} = \frac{t_m + t_a(1 - 2^{-M})}{t_m + Mt_a}$$

For example, if $p = n = q = 100$, then with 10^6 modules we could get a 4.10^5 speed up (we suppose $t_m = 3t_a$) with an occupancy factor of 4.

Before continuing to realistic systems, let us examine a paper design, the Holland machine [HOLL 59, COMF 63, GONZ 63]. A Holland machine is a two-dimensional array of cells. Each cell contains a storage register, control registers and logic, and communication logic. The storage register may hold either an instruction or a data word. A cell can be either active, in an accumulator state, or passive, these states being recorded in the control logic. Communication logic allows connections between a cell and its four neighbors. At a given time for an independent subprogram, there exists a cell in accumulator state (A), the first operand, a cell in active state ($*$) holding the instruction to be executed, and a control line from the (A) to the ($*$) cells. Each cell on the line knows its predecessors and successors. On this line lies a cell which is in "the beginning of operand path state" (P). The operand path terminates in an "end of path" cell (T), the second operand.

There are two main phases in the execution of an instruction: operand path building and execution. Path building starts from the (P) cell, down to the former (T) cell and,

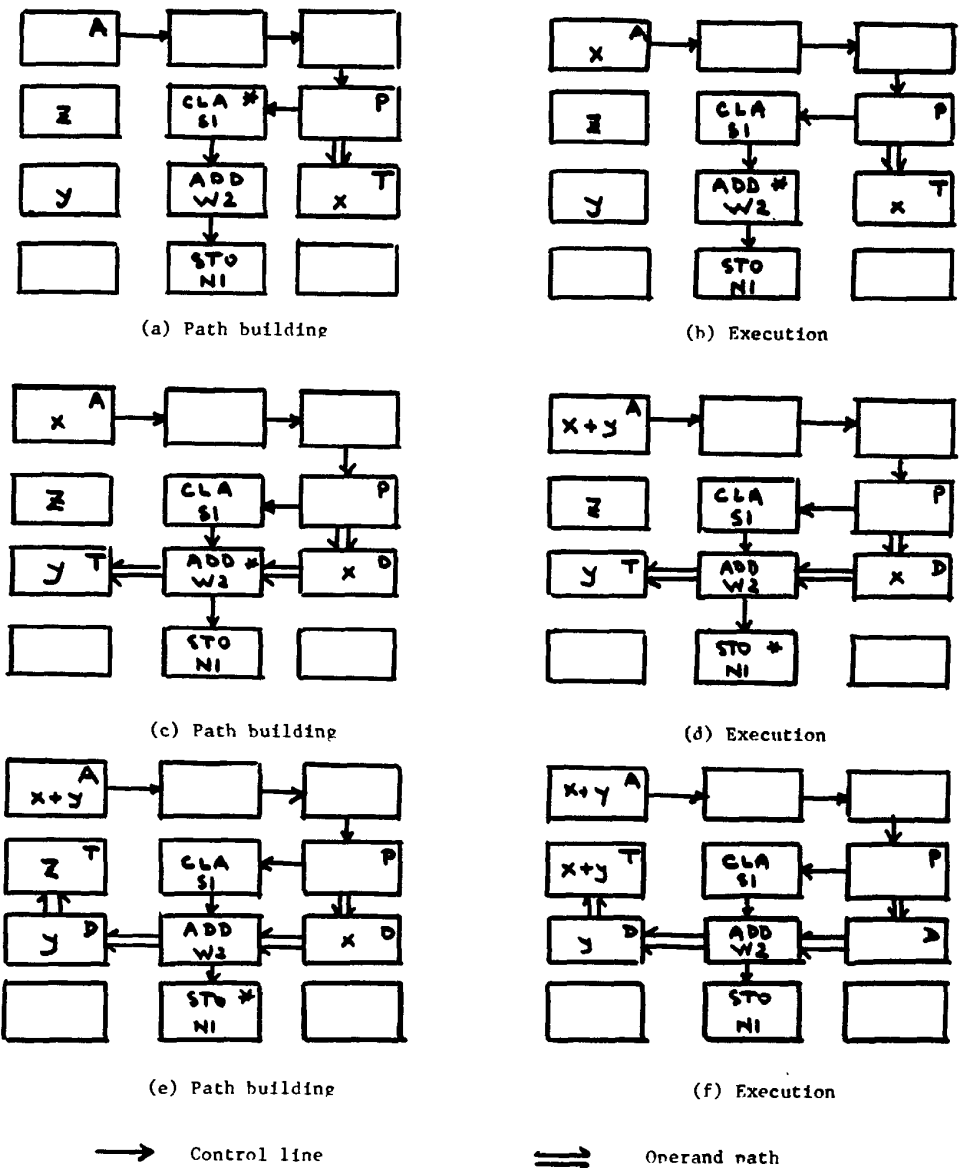


FIG. A1 Holland machine program example.

depending on the information contained in the (*) cell, a new (T) cell is created. This information could be of the form: Dn , meaning move n cells in the D direction. The operation (there is provision for eight) is then executed, the result, if any, stored

in the accumulator and the next cell in the control line becomes active. Figure A1 shows the execution of $x + y \rightarrow Z$ in such a machine.

The Holland machine approach, combined with some concepts of array processing, has

been applied to the design of the "distributed" processor [KOCZ 68, BURN 67]. Such processors also consist of cells, which can themselves be considered general-purpose computers with small memories. These cells are interconnected to form a group, and the processor itself is a net of groups. Parallelism can be present between groups and within a group. In the latter case, the cells can work independently or under the direction of a special controller cell which dictates the synchronous performances of the operations. The controller also directs intercommunication between cells, although neighborly communication exists for facility of routing.

This organization is primarily intended for spaceborne computers where a rather large number of tasks have to be performed simultaneously. Since some tasks are rather small, they can be performed by a single cell, while others which require more power will use cooperative cells; the fact that all cells are identical improves the reliability of the system. This is of the greatest importance for such applications.

Similarly, one should mention Litton's block-oriented computer [CAMP 69, CAMP 70] which also consists of individual cells which can be interconnected. The originality of this design resides in two possible modes of connection served by different busses. One is for sequential processing or filling data from memory cells. The other permits communications between any cells. The latter is time-shared by the cells which have definite positions on the bus. Therefore, communication can be performed simultaneously with processing.

Now, let us turn our attention to the linkage of two or more identical, general-purpose computers (called also multi or multiple computers) [CURT 63]. We will suppose that all processors share a common set of peripherals, and that they operate asynchronously under the direction of an executive program. To respect homogeneity (as well as reliability), any of the processors must be able to execute this supervisory program. The main problem is then the intercommunication scheme between processors and memories (we will speak of core memory

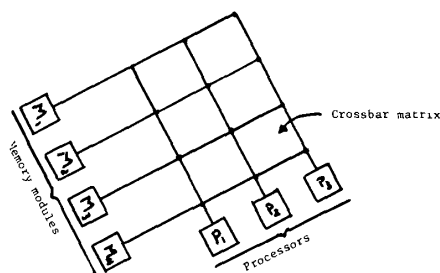


FIG. A2 Crossbar switch system

with interleaved features as "memories"). Three main implementations have been proposed and have been reviewed by Critchlow [CRIT 63]. It is to be noted that these interconnection schemes do not necessarily imply homogeneous multiprocessing systems.

Crossbar Switch System

With this system, (see Figure A2) implemented for example on the RW-400, a "polymorphic" computer [PORT 60], and on the Burroughs D825 [ANDE 62], N processors can have access via multiple wire paths to M memory modules. ($N = 1$ to 4, $M = 1$ to 16 in the D825). Because of the very large number of switches and the need for a large amount of circuitry for selection, control, and queueing of conflicting requests, this system can become too costly. Also, full reliability requires duplication of the crossbar matrix. However, it is still advocated in recent designs [LEHM 66, REIG 69], and for the Burroughs 5700 and 6700 series ($M = 1$ to 8, $N = 1$ to 8).

Schwartz [SCHW 66] proposed a modification of this scheme which he calls a "conveyor belt," for his Athene computer. To fetch an instruction or data word, the processors place their request in the section of the conveyor facing them. During the "rotation," each memory module is asked to take possible action upon the requests. In a second revolution, the conveyor picks up and distributes the fetched results to the processors.

Multiple-Bus Connected Systems

Each processor has access to any of the M memory modules by its own busses. (See

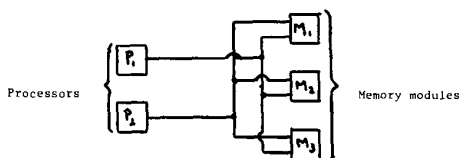


FIG. A3. Multiple-bus connected system.

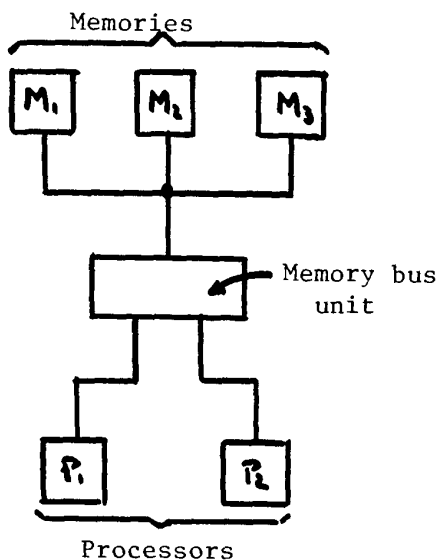


FIG. A4. Time-shared bus system

Figure A3.) This system is less costly than the crossbar system because of the reduction of the cross points. Also, modularity involves only hardware attachments at the memory ports, and is therefore easier to implement. Again, control must resolve queueing and conflicting requests. This system has been implemented on the CDC 3600 [CRIT 63], UNIVAC 1108 [STAU 67], Philco 213 [BRIG 64], MULTICS [CORB 65], the IBM 360 [BLAA 64, GIBS 66] and 9020 [BLAK 67].

Time-Shared Bus System

All memory circuitry is shared by all processors yielding a lower cost. (See Figure A4.) Connections between processors and memories need not be continuous, and time-shared techniques can be employed. Systems of this type were used in the UNIVAC LARC [ECHE 59] and are now implemented in the

communications between peripheral processors of the CDC 6600 [THOR 64] and its main memory.

This last implementation is less costly and more flexible for modularity, and control is achieved by using standard time-sharing techniques. Of course, delays may be greater than in the preceding systems.

B. Nonhomogeneous Multiprocessing Systems

In nonhomogeneous systems, some processing units have special-purpose roles. As a reduction of homogeneous systems, some multiprocessors can be the linkage of general-purpose computers, some having special functions in the system and operating concurrently. One of the first systems of this kind was PILOT [LEIN 59] which was composed of three processors, the first one handling mainly arithmetic computations, the second housekeeping, and the third I/O operations.

Another approach extending single computers is Estrin's fixed plus variable structure computer [ESTR 60]. The multiprocessor consists of three parts: a fixed part which is basically a standard uniprocessor—originally an IBM 7094, now a XDS Sigma 7—but there is no restriction and it could very well be a multiprocessor; a variable part which is composed of several special-purpose units, and a supervisory system which links the fixed and variable structure computers. These alterations and/or connections can be performed either electronically under control of the supervisory system, electromechanically by means of switches, or mechanically by changing modules in the variable structure itself.

A similar concept can be found in the design of macromodular computers [CLAR 67]. A macromodular system is composed of modules such as registers, adders, memories, control units, etc. The modules are assembled into cells with built-in communications between cells. Explicit data and control communications are by cable. The system consists of a processing network, for handling and propagation of data, and a sequencing network which by means of control nodes directs the ordering of operations. Control is asynchronous; that is, initiation of an opera-

tion depends on completion of its predecessors. Parallelism is obtained through special control elements: a "control branch module" which is the juxtaposition of control branch elements duplicating their input signal and a "rendez-vous module" which produces its output signal only when all its inputs have arrived (this is a hardware realization of the FORK-JOIN concept; see Section II.A). An interlock unit prevents conflicts from arising when two independent sequences both require the same data-processing elements by holding one of the sequences while the other is executing (see the semaphore concept in Section II.B). Juxtaposition of interlocking units allows the compatibility of more than two sequences. An example of a small general-purpose computer built following this approach can be found in [ORNS 67].

The largest existing very high-speed general-purpose computers, the IBM 360/91 and the CDC 6600 and CDC 7600, can be classified as nonhomogeneous multiprocessors, as was STRETCH, the largest computer of the previous generation [BUCH 62]. This is due to the fact that their main computing units consist of special-purpose functional units. This concept was first applied in the BULL GAMMA 60 [DREI 58] where special elements were performing arithmetic, logical functions, comparisons, and conversions from and to different codes. In the IBM 360/91 [ANDE 67] parallel execution of different instructions is made possible as follows. First, the conventional look-ahead procedures are applied to allow concurrency of decoding and instruction and operand access as well as execution of different instructions by means of storage interleaving, buffering and arithmetic instruction concurrency. It is mainly because of this latter factor that the IBM 360/91 can be considered as a multiprocessor since its computing unit consists of a fixed-point unit, a floating-point adder, and a floating-point multiplier-divider which can operate concurrently and independently. In the CDC 6600 [THOR 64], 10 peripheral processors having limited arithmetic capabilities share not only a large core memory and 24 operating registers, but also a set of 10 special-purpose functional units (9 in the 7600

[BONS 69]), performing addition, multiplication (2), division, long add, shift, Boolean operations, increment (2), and branching, all of which can function independently.

A variant to these nonhomogeneous machines has been proposed by Aschenbrenner, et al. [ASCH 67] in a system called IMP (for Intrinsic MultiProcessing), and further refined by Flynn [FLYN 70]. In this design four rings of eight sequence control units, comparable to the CPU's of conventional general-purpose computers minus the arithmetic capabilities, share a set of seven functional units (floating-point adders (2), floating-point multiplier, floating-point/fixed-point divider, and fixed-point "catch all" units (2)). At each minor cycle a sequence control unit in each ring can send a request to the functional units. The latter are heavily pipe-lined so that they can accept inputs at every minor cycle. Rings are assigned priorities so that conflicts for obtaining a functional unit can be resolved (the priorities may be rotated between rings every major cycle). Simulations have shown that for problems exhibiting a fair amount of parallelism, both sequence control units and functional units are kept busy. The former because of the load, store, branch instructions and their related indexings, the latter because they are shared by 32 processors.

C. Array Processors

A number of applications require manipulations of large sets of ordered data, such as vectors, matrices, and multidimensional arrays. Most often, operations on the members of the set are similar and can be performed concurrently, e.g., multiplication of all elements of a vector by a scalar, the first part of a dot product, etc. One approach which has been taken for the realization of fast computing techniques for such processing is the use of array processors.

The first array processor SOLOMON I led the way to its improved successors, first the SOLOMON II [SLOT 62, GROG 63, BALL 62] and now the very powerful ILLIAC IV [BARN 68]. The latter is an array of 64 processing elements, PEs, which have the capabilities of a standard CPU [DAVI 69] with a memory of 2048 64-bit words. Under control of a

control unit (CU) all PEs execute synchronously the same instruction, while at the same time the CU can fetch/decode the next instruction. Thus, the cost of decoding and timing circuits is shared by 64 PEs. Of course, some PEs can be left inoperative, if necessary by means of a status bit triggered by the CU. This status bit is part of a test result register, present in each PE, which holds the results of conditional tests. Therefore, conditional branches are accomplished by PE tests. As can be seen, an immediate advantage of array processing is that one CU allows parallel operations of 64 PEs. However, this is not sufficient since many applications, where synchronous operations of the above type are possible, might require processing on a grid. Consequently, computing on one point of the grid requires knowledge of the values computed at neighboring points.

To effect this, ILLIAC IV provides a direct data path from each PE to its orthogonally neighboring PEs. That is, processor i is directly connected to processors $i + 1$, $i - 1$, $i + 8$, and $i - 8$ (where these numbers are taken modulo 64). To go from processor i to processor j , where j is different from the 4 numbers above, a sequence of one-step routings is mandatory (the maximum number of routing steps necessary is seven). In fact, simulations have shown that routings of one step are most common and routings over two steps are very rare. The CU broadcasts instructions and common constants/operands to all PEs, therefore reducing memory needs for the PEs and allowing overlapping of common operand fetches with other operations. Input/output, as well as executive programs such as compiling and loading, are processed by a standard uniprocessor (a Burroughs B6500). However, reading and writing on and from the PE memories is performed in parallel.

A possible weakness of such a design is that the synchronous operations of the PEs, plus the fact that each PE has its own memory, constrain the addressing of the variables. In order to avoid some of these difficulties, several schemes of skewed storage have been investigated [KUCK 68]. For example, a straight storage of an $(n + 1) \times$

$(n + 1)$ matrix A in an array of $(n + 1)$ PEs is shown in Figure A5. The address a_i for PE $_i$ is the common address determined by the CU for all PEs plus the contents of index register XR_i . Access to one row of the matrix is easily achieved by setting all index registers to the same value. However, any column is entirely contained in one PE memory. Thus, operation on a single column will have the machine work at only $1/(n + 1)$ th of its capacity. However, in the skewed storage of Figure A6, rows and columns can be accessed equally well in parallel. For example, accessing the i th row demands the loading of the PEs registers with i for all of them, while for accessing the i th column, the j th index register is loaded with $(j - i) \bmod (n + 1)$.

With the same assumptions as in Section A of this Appendix, one can compute the time and occupancy factor of an array processor of N PEs for the multiplication of $A(p, n)$ by $B(n, q)$, assuming straight storage. Let $\delta_{NB} = 0$ if $q \bmod N = 0$ and $\delta_{NB} = 1$ otherwise;

$$M_A = p \bmod N, r_A = \left\lfloor \frac{n}{N} \right\rfloor, r_B = \left\lfloor \frac{q}{N} \right\rfloor$$

where $\lfloor x \rfloor$ is the greatest integer smaller than or equal to x . It can be shown [BAER 68A] that

$$\begin{aligned} T_N &= \frac{pnq(t_m + t_a)}{(t_a + t_m)[p(r_B + \delta_{NB})(Nr_A + M_A)]} \\ &= \frac{nq}{(r_B + \delta_{NB})(Nr_A + M_A)} \end{aligned}$$

and

$$F_N = \frac{q}{N(r_B + \delta_{NB})}$$

Assuming

$$N = 64, \quad p = q = n = 100$$

and

$$3t_a = t_m$$

we have:

$$T_{64} = \frac{100 \times 100}{(1 + 1)(64 + 36)} = 50$$

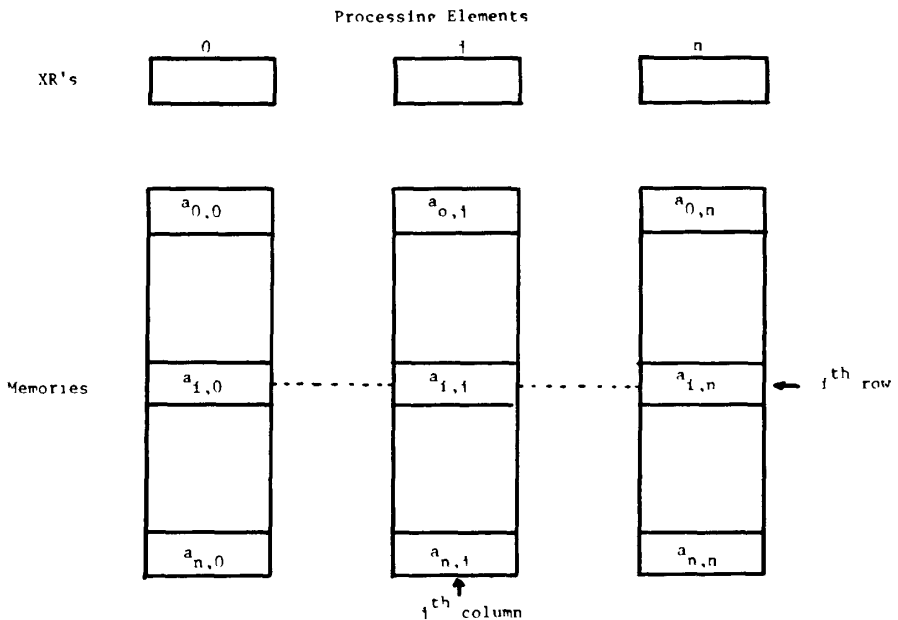


FIG A5 Straight storage matrix in ILLIAC IV

and

$$F_{64} = \frac{100}{64 \times 2} = \frac{100}{128} = .78$$

while if $N = 8$ we get $T_8 \cong 7$ and $F_8 = .89$. As can be seen $F_N \rightarrow 1$ when q becomes large with respect to N , while speed up is dependent equally on p , n and q .

Another implementation of array processing, more flexible and less powerful, has been proposed [SENZ 65]. This machine, VAMP (Vector Arithmetic Multiprocessor Computer), consists of a linear array of n processors which perform synchronously under control of a screen register. By means of a logical accumulator, compress and expand instructions facilitate the storage and handling of sparse arrays. In VAMP, memory accessing can be done in two modes. The first one is similar to ILLIAC IV's. The second one, an extension of the classical indirect addressing, consists of fetching n words from the memory (and there may be repeats of the same word) into the registers

of the arithmetic unit and then feeding back these words to the memory address registers. Thus, words belonging to the same module can be accessed and sent to the processor's registers. This scheme lends itself more easily to automatic programming.

D. Pipe-Line Computers

The concept of pipe-line computing is best explained by the following example. Consider the addition of two vectors A and B of same dimension n giving a vector C . There are n additions $a_i + b_i$ to be performed. If an addition could be decomposed into p consecutive steps of approximately the same time length, such that each step depends only on the outcome of the previous one, then the addition of the two vectors could be done as in Table A1.

In Table A1, t corresponds to the worst case of the longest step. After pt , at the end of each incremental time t , one addition will be performed. Then, the addition of the two vectors will take $(n + p)t$ time units instead

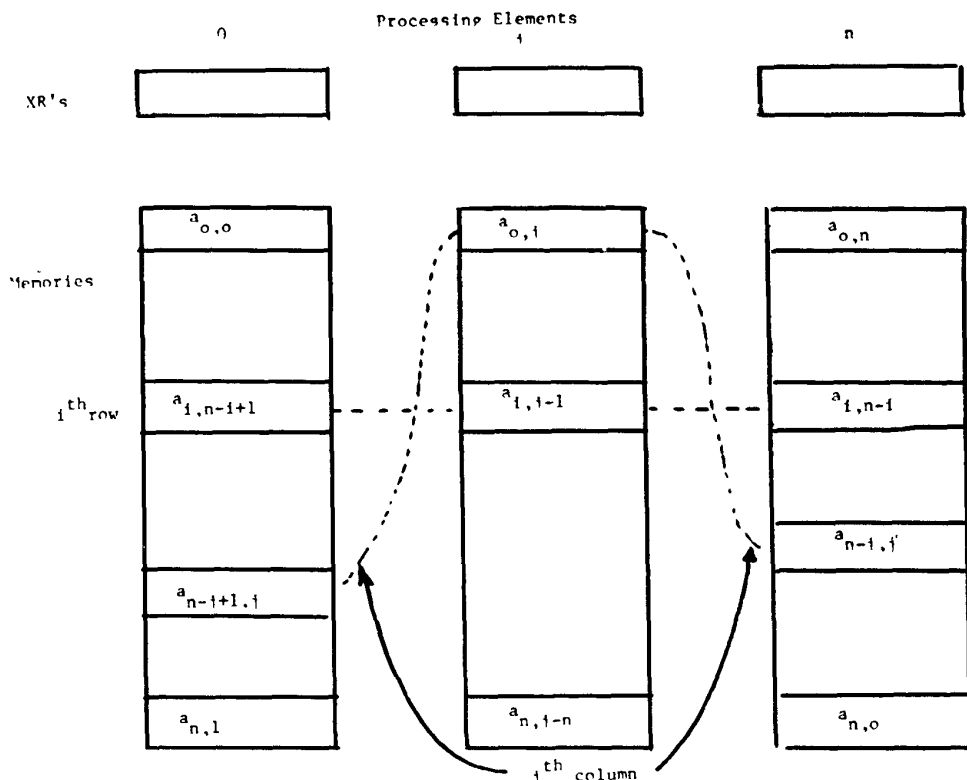


FIG. A6 Skewed matrix storage in ILLIAC IV.

TABLE A1 ADDITION OF TWO VECTORS

	Step 1	Step 2	Step p
Time t	$a_1 + b_1$		
$2t$	$a_2 + b_2$	$a_1 + b_1$	
$\vdots$			
pt	$a_p + b_p$	$a_{p-1} + b_{p-1}$	$a_1 + b_1$
$(p+1)t$	$a_{p+1} + b_{p+1}$	$a_p + b_p$	$a_2 + b_2$
nt	$a_n + b_n$	$a_{n-1} + b_{n-1}$	$a_{n-p+1} + b_{n-p+1}$
$\vdots$			
$(n+p-1)t$			$a_n + b_n$

of n add times required in a conventional computer (note, however, that n add times will be less than npt for a comparable technology). The feasibility of such units has been long established [COTT 65], and has been applied in a number of nonhomogeneous systems where each functional unit performs specific functions. For example, both

floating-point units of the IBM 360/91 utilize this technique [ANDE 67], all nine units of the CDC 7600 are pipe-lined [BONS 69], and a complete module for performing vector instructions following the above scheme can be added to an existing IBM model 360 [RUGG 69].

The first system to be built completely

on the pipe-line concept is CDC's STAR-100 (String Array) [CONT 71] computer. Its arithmetic unit consists of two floating-point pipe-lines and a string unit. STAR-100 is meant to be a very powerful computer oriented toward scientific computing. Its instruction repertoire includes a number of vector instructions besides the usual set of index, branch, arithmetic, etc. These vector instructions are divided into normal vector, sparse vector, and vector macro classes. The normal and sparse vector classes are extensions of their arithmetic equivalent on scalars, e.g., vector add, vector multiply, etc. The vector macros are extremely powerful instructions such as AVERAGE, POLY EVAL, DOT PRODUCT, etc. Pipe-line 1 performs add, subtract, multiply, and half of the vector instructions. Pipe-line 2 takes care of divide, square root, and all vector instructions. Pipe-line 1 is divided into an add pipe with 4 steps, and a multiply pipe of 6 steps. Pipe-line 2 possesses the same add structure as pipe-line 1, a divider, and a multipurpose unit of 26 segments. The string unit takes care of fixed-point, logical control, decimal, and edit instructions.

At the present time it is difficult to evaluate the power of a system like STAR-100. Only some rather idealistic cases can be analyzed [CHEN 71]. Since the STAR-100 announcement, two other powerful systems based on the same concept have appeared: Texas Instruments' Advanced Scientific Computer [WATS 72], and the British MU5 [IBBE 71].

Also, one could think of pipe-lining complete arithmetic units. For example, Bussell [BUSS 62] investigated, by means of queueing theory, but mostly by simulation, the use of a variable structure computer as a generalized pipe-line computer, composed of two units and appropriate buffers, for the solution of partial differential equations. He reported that this method presented some advantages over other parallel configurations.

Finally, nothing prevents pipe-lining being used with other types of organization, such as IMP (see part B, above) or push-down stack organization [STON 70]. In the latter case, some means of generating code efficiently are necessary (see Section III.A).