# Analysis of Programs for Parallel Processing

A. J. BERNSTEIN, MEMBER, IEEE

*Abstract*—**A set of conditions are described which determine whether or not two successive portions of a given program can be performed in parallel and still produce the same results. The conditions are general and can be applied to sections of the program of arbitrary size. The conditions are interesting because of the light they shed on the structure of programs amenable to parallel processing and the memory organization of a multi-computer system.**

## INTRODUCTION

IN THIS PAPER we will consider properties that a program must have in order for it to be processed effectively by a multi-computer system. Specifically, we will develop a set of conditions concerning two successive portions of a single program which determine whether or not the two subprograms can be performed in parallel and still produce the same results. The conditions are of interest primarily for the light they shed on the structure of programs amenable to parallel processing and the memory organization of a multi-computer system.

Programs written today are generally composed in a sequential fashion. The programmer specifies a number of tasks to be performed in a particular order by a single processor. Looping and branching in programs are not exceptions to this sequential ordering. Looping is just a shorthand way of describing a number of successive iterations of the same task, while branching allows the processor to choose among a number of alternative sequential paths.

Although a program is written in sequential fashion, the idea of processing it in parallel is not new. It was recognized very early that considerable time could be saved if certain input-output tasks were made to run concurrently with the execution of the main program. The parallel operation of the SOLOMON Computer[1,2] is considerably more sophisticated. This machine consists of an array of small, identical computers (called "processing elements") acting under the direction of a supervisory unit. Each of the computers has its own logic and memory and can communicate with adjacent units. The limitations of this machine are rather severe, however, since the processing elements cannot operate independently. Thus, at any given time, a processing element can either be idle or can execute the command being broadcast to all processing elements by the supervisory unit. SOLOMON is therefore severely restricted in the kind of tasks it can perform. In particular, it is suited for problems which involve a number of identical, independent calculations (e.g., the solution of partial differential equations).

The close relationship between parallel processing and time sharing has been pointed out by Gill.[3] Here a program is time shared on a single processor. This is useful when some part of a program must await a signal from a slower device. Rather than having the processor remain idle, it could execute some later portion of the program. When the signal is finally received, it could then revert back to its original computations. Control can thus be made to oscillate back and forth between various parts of a single program.

Very little is known about how parallel processing can be used for a more general class of programs, although the importance of the problem has been pointed out.[4] One suggestion that has been made[5,6] is to incorporate within a language a statement which explicitly specifies that a number of succeeding tasks can be performed in parallel. In this paper we will describe conditions which guarantee that two program blocks can be performed in parallel and from this obtain some information about the structure of an efficient parallel processor.

It should be pointed out that the question of whether or not two tasks can be performed in parallel is a function of not only the algorithm being performed, but also of the way it has been programmed. Thus, care must be taken not to program procedures which can be performed in parallel in a manner which does not admit of such execution. For example, certain recursive (hence sequential) computations can also be performed using non-recursive procedures which allow parallel operation. (See the example at the end of the paper for a simple illustration of this.)

Consider the flow chart in Fig. 1 indicating the

Manuscript received December 19, 1965; revised April 12, 1966.
The author is with the General Electric Research and Development Center, Schenectady, N. Y.

[1] D. L. Slotnick, W. C. Borck, and R. C. McReynolds, "The Solomon computer," *AFIPS Conf. Proc.* (*FJCC*), vol. 22, Washington, D. C.: Spartan Books, 1962, pp. 97–107.

[2] J. R. Ball, R. C. Bollinger, T. A. Jeeves, R. C. McReynolds, and D. H. Shaffer, "On the use of the Solomon parallel-processing computer," *AFIPS Conf. Proc.* (*FJCC*), vol. 22, Washington, D. C.: Spartan Books, 1962, pp. 137–146.

[3] S. Gill, "Parallel programming," *The Computer Journal*, vol. 1, pp. 2–10, April 1958.

[4] S. Fernbach, "Computers in the USA—today and tomorrow," presented at the *1965 IFIP Congress*, New York, N. Y.

[5] A. Opler, "Procedure-oriented language statements to facilitate parallel processing," *Communications of the ACM*, vol. 8, pp. 306–307, May 1965.

[6] M. E. Conway, "A multiprocessor system design," *AFIPS Conf. Proc.* (*FJCC*), vol. 24, Baltimore, Md.: Spartan Books, 1963, pp. 139–146.

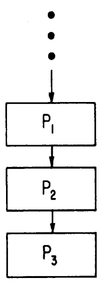

Fig. 1.

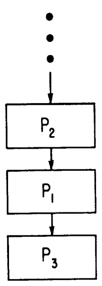

Fig. 2.

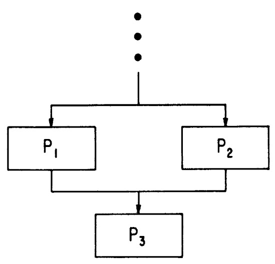

Fig. 3.

basically sequential organization of a program. Such a flow chart indicates that the execution of subprogram $P_1$ is to precede the execution of subprogram $P_2$. Program block $P_3$ represents the total portion of the program following $P_2$. In some situations it is essential that the subprograms be performed in the indicated order. For example, a result computed in $P_1$ may be required before $P_2$ can be performed. Program blocks which must be performed in the specified order will be called "sequential." On the other hand, $P_1$ and $P_2$ may be completely independent. Not only can the ordering shown in Fig. 1 be reversed as in Fig. 2, without altering the ultimate result of the program, but in fact the executions of $P_1$ and $P_2$ may be interlaced with each other. In other words, some initial part (possibly null) of $P_1$ can be performed followed by an initial part of $P_2$, followed by the next portion of $P_1$, etc. If this type of interlacing can be done in an arbitrary way and with arbitrary data input to the program, then the block diagram shown in Fig. 3 can be used to represent the process. The parallel paths indicate that the time relationship between the performance of $P_1$ and $P_2$ is arbitrary. From a practical point of view, this means that the two subprograms could be given to two separate, autonomous processors to work on simultaneously. The results produced after both had completed their work would be the same as the results produced by a single machine processing $P_1$ and $P_2$, in either order. Program blocks which are independent in the above sense will be called "parallel."

A third category exists which lies between the extremes of the parallel and sequential situations. In this case, $P_1$ and $P_2$ can be performed in either order, as shown in Figs. 1 and 2, but not in parallel. The composition of two commutative functions is an example of this situation. Program blocks satisfying this condition will be called "commutative." Although commutative subprograms do not seem to be an important class at this time, they will be treated briefly because of their relationship to the parallel situation. In addition, tests for whether two successive blocks are commutative or parallel can be used in conjunction with each other to form a test which determines whether two blocks which are not successive in the program can be performed in parallel.

## Machine Models

Since we are dealing with transformations on arbitrary algorithms, it is not surprising that undecidable questions arise. Thus it can be shown that there do not exist algorithms for deciding the commutativity or parallelism of arbitrary program blocks (see Appendix). What we will be concerned with here, however, is a more practical result. Sets of sufficient (but not necessary) conditions will be developed which guarantee parallelism and commutativity of two program blocks. In order for the conditions to be simple, we have demanded that they treat the program as a set of strings without requiring or obtaining any information about what the program actually does.

It will be shown that the conditions depend on the memory locations which are either modified or referenced in the course of executing the subprograms. Thus it is necessary to be able to determine which memory locations are processed in these two ways. In most cases, this is easily done when the program is written in a language such as FORTRAN, so we will illustrate the conditions using this language. Programs written in a symbolic machine language such as FAP can also be checked if it is understood that the set of memory locations includes not only the main storage of the machine but also such addressable registers as the accumulator, index, and sense registers. In addition, indirect addressing may prove to be an unsurmountable problem since it may prevent one from determining which memory locations are being dealt with. It should be noted that the conditions and their significance with respect to memory organization in a multi-computer system are independent of any particular language.

As might be expected from the close relationship between the conditions for parallel processing and the memory locations processed by the program, the conditions themselves depend on the model of the memory structure assumed for the machine. We will examine two such models in this paper. In model A, shown in Fig. 4(a), each processor communicates directly with a single large memory. The important feature here is that one processor can modify information which is to be fetched by the other. In model B, slave memories associated with each processor have been added. The purpose of the slave memory is to act as a storage buffer

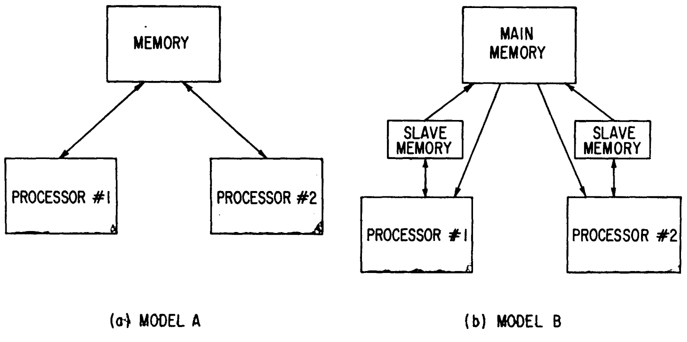

Fig. 4.

for its associated processor. Thus a processor may fetch information from the main memory but any information to be stored is kept in the slave memory. When a fetch operation is called for, the processor first searches its slave memory to see if that location has previously been stored there. If not, the location is obtained directly from the main memory. When both processors have completed their tasks, the information in the slave memories is transferred to the appropriate locations in the main memory. An example of a slave memory which operates in this fashion is the one proposed by Wilkes.[7] It will be seen that a computer organized in this fashion is much better suited for parallel operation.

## Conditions for Parallel and Commutative Transformations

Memory locations can be used in two ways by an instruction. A location can be fetched (referenced) in which case a copy of the contents of that location is brought to the processor. The information in the location itself remains intact. On the other hand, a location can be stored (modified), in which case new information is written into the location. The original contents of the location are destroyed.

It should be noted that although a subprogram may fetch a memory location, it does not necessarily use the information thus obtained. Similarly the information which a subprogram stores in a memory location may be identical with the previous contents of that location and thus no modification has taken place. It is not feasible to check for such degenerate situations without using information about what the program actually does.

Based on the above classification, we can think of four different ways that a sequence of instructions, or subprogram, $P_i$, can use a memory location.

1) The location is only fetched during the execution of $P_i$.
2) The location is only stored during the execution of $P_i$.
3) The first operation involving this location is a fetch. One of the succeeding operations of $P_i$ stores into this location.
4) The first operation involving this location is a store. One of the succeeding operations of $P_i$ fetches this location.

Let $W_i$, $X_i$, $Y_i$ and $Z_i$ be the sets of memory locations falling into categories 1, 2, 3, and 4, respectively. We will obtain conditions for commutativity and parallelism in terms of these four sets.

It is clear that only the states of memory locations in $X_i$, $Y_i$, and $Z_i$ and not the state of the entire memory are modified by the execution of $P_i$. Similarly, the execution of $P_i$ depends only on the state of the locations referenced by $P_i$—$W_i$, $Y_i$, and $Z_i$—and not on the state of the entire machine. It is important to notice a distinction in this regard between $Z_i$ on one hand and $W_i$ and $Y_i$ on the other. Whereas the information extracted by $P_i$ from $W_i$ and $Y_i$ had been established in the machine prior to the execution of $P_i$, the information extracted from $Z_i$ is actually computed within $P_i$ itself.

Consider a parallel transformation for the model $A$ machine. In order for $P_1$ in Fig. 3 to see the same initial machine state as $P_1$ in Fig. 1, regardless of how fast $P_2$ is executed along the other parallel path, we require that

$$(W_1 \cup Y_1) \cap (X_2 \cup Y_2 \cup Z_2) = \phi \tag{1}$$

where $\phi$ denotes the empty set. In addition, since we do not want $P_2$ destroying results which $P_1$ is computing for later reference, we require that

$$Z_1 \cap (X_2 \cup Y_2 \cup Z_2) = \phi. \tag{2}$$

Conditions (1) and (2) can be combined to form

$$(W_1 \cup Y_1 \cup Z_1) \cap (X_2 \cup Y_2 \cup Z_2) = \phi. \tag{3}$$

On the other hand, we do not want $P_2$ to require information computed in $P_1$ since the execution of $P_1$ will no longer necessarily precede that of $P_2$. Thus

$$(X_1 \cup Y_1 \cup Z_1) \cap (W_2 \cup Y_2) = \phi. \tag{4}$$

Also, for the same reason as mentioned for (2)

$$(X_1 \cup Y_1 \cup Z_1) \cap Z_2 = \phi. \tag{5}$$

Combining (4) and (5)

$$(X_1 \cup Y_1 \cup Z_1) \cap (W_2 \cup Y_2 \cup Z_2) = \phi. \tag{6}$$

Finally, we must insure that the partial state of the machine upon which the execution of $P_3$ depends is the same in the parallel and sequential models. The memory locations involved in this partial state are those contained in $W_3 \cup Y_3$. Of the locations modified by the combined effect of $P_1$ and $P_2$, only those modified by both $P_1$ and $P_2$ are affected by the order in which the two programs are executed. Thus we require

$$(X_1 \cup Y_1 \cup Z_1) \cap (X_2 \cup Y_2 \cup Z_2) \cap (W_3 \cup Y_3) = \phi. \tag{7}$$

[7] M. V. Wilkes, "Slave memories and dynamic storage allocation," *IEEE Trans. on Electronic Computers (Short Notes)*, vol. EC-14, pp. 270–271, April 1965.

In view of (3) and (6), however, we have

$$(X_1 \cup Y_1 \cup Z_1) \cap (X_2 \cup Y_2 \cup Z_2) = X_1 \cap X_2. \quad (8)$$

Thus, the final condition is

$$X_1 \cap X_2 \cap (W_3 \cup Y_3) = \phi. \quad (9)$$

Together, (3), (6), and (9) assure that the programs of Figs. 1 and 3 will compute the same results for any input data assuming model $A$ for the parallel processor.

Consider the conditions for parallel processing using model $B$. It is necessary to make one further assumption about the operation of the slave memories. We stipulate that when $P_1$ and $P_2$ have been executed by the two processors, the contents of the memory associated with $P_1$ are loaded into the main memory before those of the memory associated with $P_2$. Since neither processor can modify information that the other must reference, the much weaker condition (4) replaces (3) and (6). In addition, (7) is no longer required because of the above stipulation on the procedure for loading the main memory. Thus for a model $B$ computer, (4) alone is sufficient for parallel operations.

The conditions for commutativity are strongly related to those derived above. Although (2) and (5) are no longer required, (1), (4), and (7) still hold. Simplifying (7) we obtain the following three conditions for commutativity

$$(W_1 \cup Y_1) \cap (X_2 \cup Y_2 \cup Z_2) = \phi \quad (10)$$

$$(W_2 \cup Y_2) \cap (X_1 \cup Y_1 \cup Z_1) = \phi \quad (11)$$

$$(X_1 \cup Z_1) \cap (X_2 \cup Z_2) \cap (W_3 \cup Y_3) = \phi. \quad (12)$$

The conditions for the three situations are summarized in Table I where the letters $A$, $B$, and $C$ stand for parallel operation model $A$, parallel operation model $B$, and commutativity, respectively. Thus the letter $B$ at the intersection of row $W_2$ and column $Z_1$ indicates that $W_2 \cap Z_1 = \phi$ is a condition for parallel operation using model $B$. Letters with overbars indicate that the indicated set intersection must be further intersected with the set $W_3 \cup Y_3$ to form the condition. Overbars thus indicate weaker conditions.

TABLE I

| | $W_1$ | $X_1$ | $Y_1$ | $Z_1$ |
|---|---|---|---|---|
| $W_2$ | | $ABC$ | $ABC$ | $ABC$ |
| $X_2$ | $AC$ | $\overline{AC}$ | $AC$ | $A\overline{C}$ |
| $Y_2$ | $AC$ | $ABC$ | $ABC$ | $ABC$ |
| $Z_2$ | $AC$ | $A\overline{C}$ | $AC$ | $A\overline{C}$ |

Note that although pure procedure is an extremely useful concept in connection with parallel operation, the above stated conditions still apply since it is the memory locations which are used by procedures which determine whether or not they can be carried on in parallel.

CHECKING THE CONDITIONS

Determining sets $W_i$, $X_i$, $Y_i$, and $Z_i$ is, in most cases, simple for a program written in a FORTRAN-like language since the memory locations involved are specifically stated in the instructions. Thus the arithmetic instruction

$$A = \text{TEMP} + 1$$

references the location TEMP and modifies the location $A$, while the instruction

$$A = A + 1$$

first references $A$ and then modifies it. Table II gives a partial listing of those FORTRAN instructions which modify and reference memory locations. Omitted from the table are the input-ouput statements. Clearly the input statements modify while the output statements reference the specified memory locations. In addition, statements such as EQUIVALENCE and COMMON cause several variables to be assigned to the same location. Such correspondence must be taken into account in forming the sets.

Because of the way categories 3 and 4 are defined, it is not sufficient to simply determine whether a particular location is fetched or stored. The ordering of these operations must also be established. Thus in analyzing a program block, the instructions must be examined in the sequence of their intended execution. If the program block contains branching, then all possible branches must be examined since it cannot be anticipated in advance which branch the program will take on execution. If a particular memory location is referenced in one branch and modified in another, then one must consider that that memory location is both referenced and modified by the program block.

TABLE II

| Statement | Locations Modified | Locations Referenced |
|---|---|---|
| assignment statement | variable on LHS of equal sign | all variables on RHS of equal sign |
| ASSIGN S TO $N$ | $N$ | |
| GO TO $N$, (S$_1$, $S_2 \cdots S_m$) | | $N$ |
| DO $S$ $I = N_1, N_2, N_3$ | $I$* | $N_1, N_2, N_3$ |
| CALL name ($A_1, A_2 \cdots A_n$) | $A_1, A_2 \cdots A_n$** | $A_1, A_2 \cdots A_n$** |
| FUNCTION name ($A_1, A_2 \cdots A_n$)*** | | $A_1, A_2 \cdots A_n$ |
| GO TO ($S_1, S_2, \cdots S_m$), $I$ | | $I$ |
| IF (arith. expression) $S_1, S_2, S_3$ | | all variables in arith. exp. |
| IF (flip flop) $S_1, S_2$ | flip flop | flip flop |

* The variable $I$ may or may not be modified depending on whether the loop is exited prematurely or is exited after being satisfied.

** Actually not all the $A_i$ are modified or referenced. The subsets of the $A_i$ which fall into the two categories can be determined directly from the subprogram.

*** The actual referencing takes place when the function is called.

Branching may, in some cases, cause the above procedure to include cells in some set which do not belong. Such a situation arises when a program is written in a way which prevents one from ever entering a particular branch under any circumstances. This may be extremely difficult to detect and so the procedure takes the conservative point of view that every branch may be entered. This may cause some parallel or commutative blocks to be classified as sequential, but never vice versa. One other difficulty in determining the exact contents of a given set should be mentioned. Consider the case where a statement involves a member of an array, but the exact coordinates of that member are the results of some previous calculation. Although it is reasonable to allow some set, say $X_1$, to contain a member $A$(SUB1), it may be impossible without actually executing the program to check (4) if $Y_2$ contains a member $A$(SUB2). Here again one must take the conservative point of view that unless it can be demonstrated otherwise there exists a set of input data which causes SUB1 = SUB2, and so $P_1$ and $P_2$ are sequential.

One additional point in connection with branching must be examined. Consider the situation illustrated in Fig. 5, where the program $P_1$ has as a possible branch one which eliminates the execution of $P_2$. Although $P_1$ and $P_2$ may be parallel program blocks for a model $A$ machine, it may not actually be possible to execute them in parallel without disrupting the execution of the succeeding program block $P_3$. This would be the case when locations modified by $P_2$ were referenced by $P_3$. Thus in order to execute $P_1$ and $P_2$ in parallel, one would require the additional condition

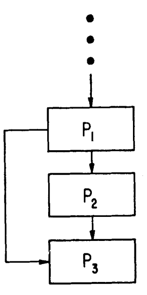

Fig. 5.

$$(W_3 \cup Y_3) \cap (X_2 \cup Y_2 \cup Z_2) = \phi \tag{13}$$

where $P_3$ represents the entire succeeding portion of the program after $P_2$. It should be noted that (13) is not necessary for the model $B$ machine since the processors do not write directly into the main memory. Conditions for other types of branching situations can easily be derived.

## Loops

The above conditions assume a special form when applied to the successive iterations of a DO loop. Consider the loop shown in Fig. 6(a). The actual program described is shown in Fig. 6(b) and the question to be investigated is under what conditions is this equivalent to the program of Fig. 6(c). We will consider only the model $B$ computer. The conditions are

$$(W(I) \cup Y(I)) \cap (X(J) \cup Y(J) \cup Z(J)) = \phi; \quad 1 \le J < I \le N. \tag{14}$$

It should be pointed out, however, that it is not actually necessary to form the four sets for all $I$ in the given range. Since each pass through the loop is really a repetition of the same process using a different value of the index, it follows that each of the sets can be obtained from a single representative set by substituting the appropriate value of $I$. Thus there exists a non-null intersection in (14) if for example both $X(I)$ and $Y(I)$ contain the same fixed location (e.g., TEMP$\epsilon X(I)$, TEMP$\epsilon Y(I)$) or if they both refer to different parts of the same array (e.g., $A(I)\epsilon X(I)$; $A(I-\Delta)\epsilon Y(I)$, $1 \le \Delta < N$).

An additional problem which can be handled using the above techniques is the decomposition of loops. Here we would like to determine the conditions under which the program of Fig. 7(a) is equivalent to the program of Fig. 7(b). The goal is no longer to break the program into parallel parts; hence only the commutativity of the blocks is actually of concern. In Fig. 7(b), the order of $P_1(I)$ and $P_2(J)$ has been reversed for all $J$ such that $J < I$. Thus, the following conditions are sufficient for the transformation:

$$\left.\begin{aligned} (W_1(I) \cup Y_1(I)) \cap (X_2(J) \cup Y_2(J) \cup Z_2(J)) &= \phi \\ (W_2(J) \cup Y_2(J)) \cap (X_1(I) \cup Y_1(I) \cup Z_1(I)) &= \phi \\ (X_1(I) \cup Z_1(I)) \cap (X_2(J) \cup Z_2(J)) \cap (W_3 \cup Y_3) &= \phi \end{aligned}\right\} \quad 1 \le J < I \le N. \tag{15}$$

This transformation is particularly useful for the simplification of programs if $P_1(I)$ and $P_2(I)$ can be chosen so that one of them is actually independent of the index $I$. In such a case, the loop is said to contain an invariant calculation. If $P_1(I)$ is an invariant calculation, then Fig. 7(b) reduces to Fig. 8 which is considerably simpler to perform since the program $P_1$ is only executed once. This kind of simplification has been discussed by Gear.[8]

[8] C. W. Gear, "High speed compilation of efficient object code," *Communications of the ACM*, vol. 8, pp. 483–488, August 1965.

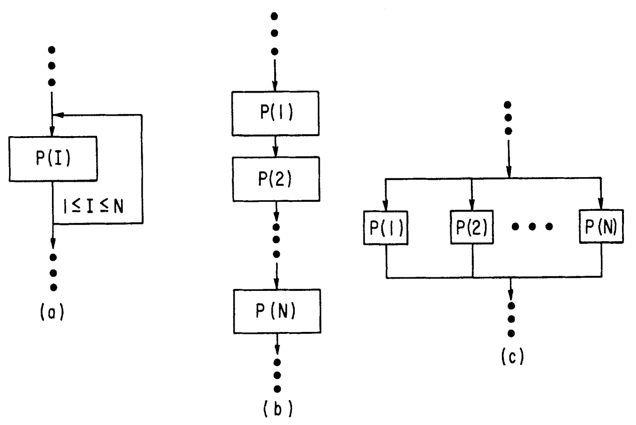

Fig. 6.

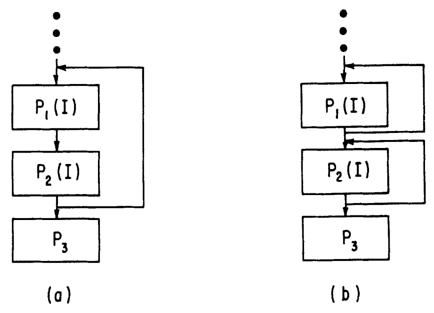

Fig. 7.

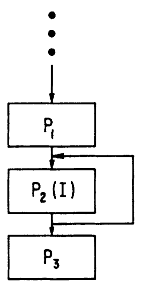

Fig. 8.

## Section IV—Example

The following example will be used to illustrate the conditions. A test will be carried out on the hypothetical subprogram of Fig. 9 to determine whether the DO loop consisting of statements 2 through 16 can be performed as shown in Fig. 6(c). The sets $W(N)$, $X(N)$, $Y(N)$, and $Z(N)$ are shown in Fig. 10. Next to the memory locations in each of the sets are the relevant statement numbers of the program. The program has been constructed to be sequential for the $B$ (and hence, the $A$) machine to illustrate the following conflicts:

1) $Z(N) \cap Z(N+1) = \{\text{ALPHA}\}$.

This conflict violates a condition of the form (6) for model $A$ machines. It causes no difficulty for model $B$ machines.

2) $X(N-1) \cap W(N)$

$$= \{\text{BETA};\ C(I, N),\ (2 \leq I \leq NN)\}.$$

Notice that BETA appears in both $W(N)$ and $X(N)$ because of the way it is used in the two branches of the IF statement. Since it is not known which branch will be taken on the $N$th pass through the loop, there is no way of telling whether the value of BETA calculated on some previous pass will be utilized.

3) $Z(N-1) \cap W(N) = \{C(1, N)\}$

4) $Y(N-1) \cap Y(N) = \{\text{MAX}\}$.

Notice that in contrast to the other set intersections there is nothing essentially sequential about the way MAX appears in the program. Thus if statements 1 and

```
 1     MAX=10
 2     DO 16 N=1, NN
 3     DO 7 I=1, NN
 4     SUM (I,N)=MAX
 5     DO 7 K=1, NN
 6     SUM (I,N)=SUM (I,N)+A (I,K)* C(K,N)
 7     CONTINUE
 8     DO 9 I=1, NN
 9     C (I,N+1)=SUM (NN,N)+B (I)
10     IF (C(1,N+1)) 11, 13, 13
11     BETA=C(1, N+1)
12     GO TO 14
13     ALPHA=BETA+1.
14     GAMMA=ALPHA**3
15     MAX=MAX+2
16     CONTINUE
```

Fig. 9.

| | | | | | |
|---|---|---|---|---|---|
| | $W(N)$ | | | $X(N)$ | |
| $NN$ | | (3) | $C(I, N+1)$ | $2 \leq I \leq NN$ | (9) |
| $A(I, K)$ | $1 \leq I \leq NN$ | (6) | BETA | | (11) |
| | $1 \leq K \leq NN$ | | GAMMA | | (14) |
| $C(K, N)$ | $1 \leq K \leq NN$ | (6) | | | |
| $B(I)$ | $1 \leq I \leq NN$ | (9) | | | |
| BETA | | (13) | | | |
| | $Y(N)$ | | | $Z(N)$ | |
| MAX | | (4), (15) | SUM $(I, N)$ | $1 \leq I \leq NN$ | (4), (6) |
| | | | $C(1, N+1)$ | | (9), (10) |
| | | | ALPHA | | (13), (14) |

Fig. 10.

15 were deleted and the statement

$$\text{MAX} = 10 + 2*(N - 1)$$

were inserted between 2 and 3, the program would perform the same calculation but MAX would now be an element of $Z(N)$. Thus essentially parallel calculations can be disguised in sequential form.

## Appendix

Only the proof of undecidability with respect to the parallel situation will be given here. The proof for commutativity follows exactly the same lines. Recall that two program blocks are parallel if and only if they produce the same results when performed sequentially or in parallel for all possible sets of input data. Thus, since we must allow for input data of arbitrary size, we will assume that this information is written on an arbitrarily long input tape.

*Theorem*

The parallelism of two program blocks is undecidable.

*Proof:* Suppose there exists an algorithm for determining whether any two program blocks can be performed in parallel. We will show that this assumption implies a solution to the halting problem for an arbitrary Turing Machine $T$ starting with an arbitrary initial tape.

Let $n$ be an arbitrary integer stored on the input tape, and let $n_k$ be the $k$ most significant digits of $n$. Figure 11 is a flow diagram description of a program to be tested by the algorithm. Notice that if $T$ never halts

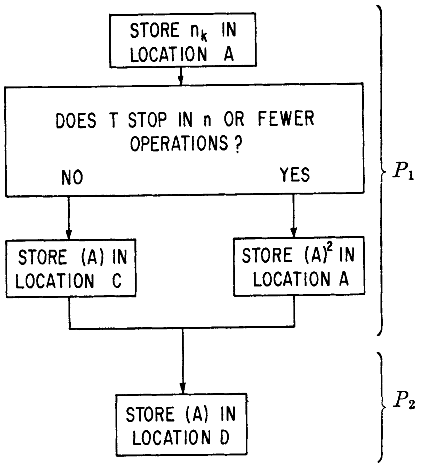
STORE $n_k$ IN LOCATION A
DOES T STOP IN n OR FEWER OPERATIONS ?
NO
YES
STORE (A) IN LOCATION C
STORE $(A)^2$ IN LOCATION A
STORE (A) IN LOCATION D
$P_1$
$P_2$

Fig. 11.

then for all input data $n$, $P_1$ takes the no-branch. In this case $P_1$ and $P_2$ can be performed in parallel. If $T$ eventually stops, however, there exist values of $n$ for which $P_1$ takes the yes-branch. In this case, $P_1$ and $P_2$ must be performed sequentially since $P_1$ computes information required for the execution of $P_2$. Thus determining whether $P_1$ and $P_2$ can be performed in parallel is equivalent to solving the halting problem for $T$.

## Acknowledgment

The author would like to thank Dr. J. Johnston, Dr. P. Lewis, Dr. R. Shuey, Dr. R. Stearns, and Dr. D. Younger of the Information Studies Branch of the G.E. Research and Development Center for the suggestions and ideas offered by them in connection with this work.

# Error Codes for Arithmetic Operations

HARVEY L. GARNER

*Abstract*—**This paper classifies error correcting codes for arithmetic and provides the necessary and sufficient conditions for the various code classes. Particular attention is given to the arithmetic properties of the codes. It is shown that the Brown codes and the Henderson codes are examples of a general class of nonseparate, nonsystematic codes. Henderson's systematic code is shown to be the only systematic, nonseparate code. Finally, conditions are given for the existence of code classes for both radix complement and diminished radix complement coded arithmetic systems.**

Manuscript received July 29, 1965; revised April 11, 1966. This research was partially supported by Air Force Contract AF30(602)-3546.

The author is with the Department of Electrical Engineering, University of Michigan, Ann Arbor, Mich.

## I. Introduction

ERROR CORRECTING codes for arithmetic operations have received considerable attention. Peterson has shown that all separate checking codes are residue codes [9]. Brown has introduced the $AN$ codes [1] and Henderson has given examples of systematic codes [7]. Since Henderson does not consider the code arithmetic, it is impossible to determine whether these codes as presented by Henderson were meant to be separate or nonseparate. It is shown that both the Brown codes and the Henderson codes are members of the same general class of nonseparate codes