Analysis and detection of parallel processable code

D. J. Evans and Shirley A. Williams*

Department of Computer Studies, Loughborough University of Technology, Loughborough, Leicestershire

In this paper a method is developed for locating potential parallelism within serial programs written in Algo-type languages.

(Received December 1977; revised October 1978)

1. Introduction

Over the past few years computers have been designed and developed which possess a number of processing units, thus giving facilities to increase the throughput of computation (Ramamoorthy and Gonzalez, 1969).

Methods have been formulated and implemented by which a programmer may indicate where parts of his program may be executed on different processors at the same time (i.e. in parallel) (Anderson, 1965). This, although useful in some circumstances, raises three main problems:

1. The onus is on the programmer to detect and express all possible parallelism in his program.
2. Small changes in the program may mean that the programmer has to reorganise all the parallelism he has written in to the program.
3. Programs already in existence will have to be rewritten.

It would be beneficial to be able to examine automatically programs and indicate the relationships between parts of the code. Methods of converting serial FORTRAN programs into a form suitable for execution in parallel have been investigated by Kuck *et al.* (Kuck, 1975; Towle, 1976). In this paper we shall develop a means of locating parallelism in algorithm type languages (ALGOL) using a notation as developed by Bernstein (1966) in an extremely informative paper.

2. Usage of private and shared memories

In 1965 Wilkes introduced the idea of using a slave memory of fast core to save on the fetch time from main memory. Although such time delays are now of less significance it is possible to apply this concept to a multiprocessing environment by allowing each processor to have a private memory, which can be used in the same manner as Wilkes' slave memory.

Thus there are two types of memory structures that can be used in parallel processing, either (*a*) all processors use the same main memory or (*b*) each processor has attached to it a private memory in which information that is currently being processed can be stored. Usually when the process has finished the information is returned to main memory.

This difference can be emphasised by considering two processors A and B operating in parallel, and *A* alters location *l* before *B* fetches it. Then, with shared memory *B* will fetch the value altered by *A*, whereas using private memory *B* will fetch the value that was there previously.

3. Relationships between segments of code

We define the following notation which is used extensively in this work. Bernstein (1966) defines four categories (W, X, Y and Z) corresponding to the four different ways that a sequence of instructions, or a subprogram, P, can use a memory location. They are as follows:

W—'The location is only fetched during the execution of P'. (3.1a)

X—'The location is only stored during the execution of P'. (3.1b)

Y—'The first operation involving this location is a fetch. One of the succeeding operations of P stores in this location'. (3.1c)

Z—'The first operation involving this location is a store. One of the succeeding operations of P fetches from this location'. (3.1d)

Bernstein also developed conditions to test if two P's can be executed in parallel using this notation.

Stanza

We need to define a term that will be similar to P with which the relationship between different parts of a program can be described. Since ALGOL-type languages are being considered in this report an equivalent to P could be: 'A group of compound statements appearing adjacently in a program'. The term 'block' is suitable for this definition but because of its specific meaning in ALGOL-type languages the term 'stanza' is used instead.

Classes of relationships

Let us now consider the relationships between two stanzas executed one directly after the other if the program was being executed serially. We shall call these two stanzas S_i and S_{i+1}, $1 \leqslant i < N$, where N is the total number of stanzas in the program.

Five possible relationships that can exist between two such adjacent stanzas are:

1. Prerequisite(PR) Stanza S_i must fetch what it requires before S_{i+1} stores its results. (3.2a)
2. Conservative(CV) Stanza S_i must store its results before S_{i+1} does. (3.2b)
3. Commutative(CM) Stanza S_i may be executed before or after S_{i+1} is executed but not at the same time. (3.2c)
4. Contemporary(CT) Stanzas S_i and S_{i+1} can be executed at the same time and the locations used may be accessed in any order. (3.2d)
5. Consecutive(CC) Stanza S_i must store its results before S_{i+1} fetches what it requires. (3.2e)

For completeness, let us define a sixth relationship, which cannot exist within a serial program.

6. Synchronous(SN) Stanzas S_i and S_{i+1} must both have the same inputs, i.e. S_i (or S_{i+1}) cannot store its results until S_{i+1} (or S_i) has fetched its inputs. (3.2f)

For each stanza (S_i) we can form Bernstein sets such that:

W_i represents the set of all locations that are only fetched during the execution of S_i. (3.3a)

X_i represents the set of all locations that are only stored during the execution of S_i. (3.3b)

*now at Department of Computer Science, The University of Reading, Whiteknights Park, Reading RG6 2AX.

Y_i represents the set of all locations for which the first operation is a fetch, and one of the succeeding operations of S_i is a store. (3.3c)

Z_i represents the set of all locations for which the first operation is a store and one of the succeeding operations of S_i fetches that location. (3.3d)

To reduce some of the complexities of this work we shall use the following abbreviations:

WY_i represents the set of all locations that fetch values not set in S_i.
Thus:

$$WY_i = W_i \cup Y_i \tag{3.3e}$$

XYZ_i represents the set of all locations that receive a new value in S_i.
Thus:

$$XYZ_i = X_i \cup Y_i \cup Z_i \tag{3.3f}$$

We shall also introduce a set V that represents all locations that are fetched before being stored, after S_i and S_{i+1} have been executed. The calculation of V will, in general, be a nontrivial matter, so V may be considered to be the full set of variables, which are assigned to within both stanzas S_i and S_{i+1}.

Using this notation we can redefine the relationships given earlier.

1. *Private memories*

1.1. *Prerequisite*

Stanza S_i must fetch what it requires before S_{i+1} stores its results,

$$\text{i.e.}\quad WY_i \cap XYZ_{i+1} \neq \emptyset \tag{3.4}$$

(where $\emptyset$ is the null set).

Stanza S_{i+1} must not require information computed in S_i, since S_i will not necessarily be completed,

$$\text{i.e.}\quad XYZ_i \cap WY_{i+1} = \emptyset \tag{3.5}$$

Locations that are modified in both S_i and S_{i+1} must not be used elsewhere without being reset first, since the values of such locations are undefined,

$$\text{i.e.}\quad XYZ_i \cap XYZ_{i+1} \cap V = \emptyset \tag{3.6}$$

Thus, (3.5) and (3.6) are the conditions to be satisfied for S_i and S_{i+1} to be prerequisite.

1.2. *Conservative*

Stanza S_i must store its results before S_{i+1} does

$$\text{i.e.}\quad XYZ_i \cap XYZ_{i+1} \cap V \neq \emptyset \tag{3.7}$$

Stanza S_{i+1} must not require information computed in S_i.

$$\text{i.e.}\quad XYZ_i \cap WY_{i+1} = \emptyset \tag{3.8}$$

This is the condition that needs to be satisfied for S_i and S_{i+1} to be conservative.

1.3. *Commutative*

Stanza S_i may be executed before or after S_{i+1}. The inputs of S_{i+1} (S_i) must not coincide with the outputs of S_i (S_{i+1})

$$\text{i.e.}\quad WY_i \cap XYZ_{i+1} = \emptyset \tag{3.9}$$

$$XYZ_i \cap WY_{i+1} = \emptyset \tag{3.10}$$

Locations that are modified by S_i and S_{i+1} must not be used elsewhere without being reset first, since the value of such locations are undefined,

$$\text{i.e.}\quad XYZ_i \cap XYZ_{i+1} \cap V = \emptyset$$

However in view of (3.9) and (3.10)

$$XYZ_i \cup XYZ_{i+1} = (X_i \cup Y_i \cup Z_i) \cap (X_{i+1} \cup Y_{i+1} \cup Z_{i+1}) = (X_i \cup Z_i) \cap (X_{i+1} \cup Z_{i+1})$$

$$\text{so}\qquad (X_i \cup Z_i) \cap (X_{i+1} \cup Z_{i+1}) \cap V = \emptyset \tag{3.11}$$

Thus the conditions for S_i and S_{i+1} to be commutative stanzas are (3.9), (3.10) and (3.11).

1.4. *Contemporary*

Stanza S_i and S_{i+1} can be executed at the same time and the locations used may be stored and fetched in any order. There must be no dependencies between the inputs and outputs of S_i and S_{i+1},

$$\text{i.e.}\quad WY_i \cap XYZ_{i+1} = \emptyset \tag{3.12}$$

$$XYZ_i \cap WY_{i+1} = \emptyset \tag{3.13}$$

Locations that are modified by both stanzas S_i and S_{i+1} must not be used elsewhere without being reset first, since the value of such locations are undefined,

$$\text{i.e.}\quad XYZ_i \cap XYZ_{i+1} \cap V = \emptyset$$

However in view of (3.12) and (3.13)

$$XYZ_i \cap XYZ_{i+1} = (X_i \cup Y_i \cup Z_i) \cap (X_{i+1} \cup Y_{i+1} \cup Z_{i+1}) = (X_i \cup Z_i) \cap (X_{i+1} \cup Z_{i+1})$$

$$\text{so}\qquad (X_i \cup Z_i) \cap (X_{i+1} \cup Z_{i+1}) \cap V = \emptyset \tag{3.14}$$

Thus the conditions for two stanzas to be contemporaries are (3.12), (3.13) and (3.14).

1.5. *Consecutive*

Stanza S_i must store its results before S_{i+1} fetches what it requires

$$\text{i.e.}\quad XYZ_i \cap WY_{i+1} \neq \emptyset \tag{3.15}$$

Now let us consider how these relationships exist with shared memory.

2. *Shared memory*

2.1. *Prerequisite*

Stanza S_i must fetch what it requires before S_{i+1} stores its results,

$$\text{i.e.}\quad WY_i \cap XYZ_{i+1} \neq \emptyset \tag{3.16}$$

Since both stanzas are now using the same memory for storing into and fetching from this means that the last fetch of S_i must be completed before the first store of S_{i+1}, and so the relationship degenerates into a consecutive one.

2.2. *Conservative*

Stanza S_i must store its results before S_{i+1} does,

$$\text{i.e.}\quad XYZ_i \cap XYZ_{i+1} \cap V \neq \emptyset \tag{3.17}$$

This becomes the last store of S_i, must be completed before the first store of S_{i+1} can be done and so this relationship also becomes a consecutive one.

2.3. *Commutative*

Stanza S_i may be executed before or after S_{i+1}. The inputs to stanzas $S_i(S_{i+1})$ must not be altered by the outputs of $S_{i+1}(S_i)$

$$WY_i \cap XYZ_{i+1} = \emptyset \tag{3.18}$$

$$XYZ_i \cap WY_{i+1} = \emptyset \tag{3.19}$$

Locations that are modified by S_i and S_{i+1} must not be used elsewhere without being reset first, since the value of such locations are undefined,

$$\text{i.e.}\quad XYZ_i \cap XYZ_{i+1} \cap V = \emptyset \tag{3.20}$$

However in view of (3.18) and (3.19)

$$XYZ_i \cap XYZ_{i+1} = (X_i \cup Y_i \cup Z_i) \cap (X_{i+1} \cup Y_{i+1} \cup Z_{i+1}) = (X_i \cup Z_i) \cap (X_{i+1} \cup Z_{i+1})$$

$$\text{so}\qquad (X_i \cup Z_i) \cap (X_{i+1} \cup Z_{i+1}) \cap V = \emptyset \tag{3.21}$$

Thus the conditions for S_i and S_{i+1} to be commutative are (3.18), (3.19) and (3.21).

2.4. *Contemporary*

Stanzas S_i and S_{i+1} can be executed at the same time and the locations used may be stored or fetched in any order.

There must be no dependencies between the inputs and outputs of stanzas S_i and S_{i+1},

$$\text{i.e.}\quad WY_i \cap XYZ_{i+1} = \emptyset \qquad (3.22)$$

$$XYZ_i \cap WY_{i+1} = \emptyset \qquad (3.23)$$

Partial results being prepared by $S_i(S_{i+1})$ must not be overwritten by $S_{i+1}(S_i)$.

$$\text{i.e.}\quad Z_i \cap XYZ_{i+1} = \emptyset \qquad (3.24)$$

$$XYZ_i \cap Z_{i+1} = \emptyset \qquad (3.25)$$

Combining (3.22) and (3.24)

$$(WY_i \cup Z_i) \cap XYZ_{i+1} = \emptyset \qquad (3.26)$$

Combining (3.23) and (3.25)

$$XYZ_i \cap (WY_{i+1} \cup Z_{i+1}) = \emptyset \qquad (3.27)$$

Locations that are modified in both S_i and S_{i+1} must not be used elsewhere without being reset first, i.e.

$$XYZ_i \cap XYZ_{i+1} \cap V = \emptyset \qquad (3.28)$$

However, in view of (3.26) and (3.27)

$$XYZ_i \cap XYZ_{i+1} = (X_i \cup Y_i \cup Z_i) \cap (X_{i+1} \cup Y_{i+1} \cup Z_{i+1}) = X_i \cap X_{i+1} \qquad (3.29)$$

so

$$X_i \cap X_{i+1} \cap V = \emptyset \qquad (3.30)$$

Thus, the conditions for two stanzas to be contemporary are (3.26), (3.27) and (3.30).

2.5. *Consecutive*
Stanzas S_i must store its results before S_{i+1} fetches what it requires

$$XYZ_i \cap WY_{i+1} = \emptyset \qquad (3.31)$$

It is possible to expand on the relationships previously defined for two stanzas to cover n stanzas, which exist in a serial order $\{S_1, S_2, \ldots, S_n\}$.

3. *Expanded relationships*
3.1. *Prerequisite*
Stanza S_k must fetch what it requires before S_{k+1} stores its results for all k such that $1 \leqslant k < n$.

3.2. *Conservative*
Stanza S_k must store its results before S_{k+1} does for all k such that $1 \leqslant k < n$.

3.3. *Commutative*
The set of stanzas $\{S_{i_1}, S_{i_2}, \ldots, S_{i_n}\}$ may be executed in any possible order of the set $\{i_1, i_2, \ldots, i_n\}$, which is any permutation of the set $\{1, 2, \ldots, n\}$, providing S_{i_k} is completed before S_{i_k+1} commences.

3.4. *Contemporary*
Stanzas $\{S_1, S_2, \ldots, S_n\}$ can be executed at the same time, the ordering of fetching and storing being of no consequence.

3.5. *Consecutive*
Stanza S_k must be completed before S_{k+1} commences for all k such that $1 \leqslant k < n$.

Again for completeness a sixth relationship can be defined, which, however, cannot exist within a serial program.

3.6. *Synchronous*
Stanzas $\{S_1, S_2, \ldots, S_n\}$ must all receive the same input set.

Using the Bernstein type of notation as before we can show what conditions are necessary for a relationship. These values can be seen in **Table 1.**

4. Assignment stanzas
A stanza which just contains assignment statements can be called an *Assignment stanza* or an *As-stanza* **Figs. 1**(*a*) and **2**(*a*) are examples of As-stanzas. The Bernstein sets formed from these expressions are shown in Figs. 1(*b*) and 2(*b*), respectively. Assume that the two stanzas are written so that Fig. 1(*a*) is executed immediately before Fig. 2(*a*). Then by carrying out the tests, on the Bernstein sets (see **Fig. 3**), as given in Table 1 we see that if private memory is available to each processor then these stanzas are prerequisite, if the processors need to use main memory then they are consecutive.

Table 1

	Two stanzas		n stanzas	
	Private memories	*Shared memories*	*Private memories*	*Shared memories*
Prerequisite (PR)	$XYZ_i \cap WY_{i+1} = \emptyset$ $XYZ_i \cap XYZ_{i+1} \cap V = \emptyset$	same as consecutive	$XYZ_k \cap (WY_{k+1} \cup \ldots \cup WY_n) = \emptyset$† $XYZ_k \cap (XYZ_{k+1} \cup \ldots \cup XYZ_n) \cap V = \emptyset$†	same as consecutive
Conservative (CV)	$XYZ_i \cap WY_i = \emptyset$	same as consecutive	$XYZ_k \cap (WY_{k+1} \cup \ldots \cup WY_n) = \emptyset$†	same as consecutive
Commutative (CM)	$WY_i \cap XYZ_{i+1} = \emptyset$ $XYZ_i \cap WY_{i+1} = \emptyset$ $(X_i \cup Z_i) \cap (X_{i+1} \cup Z_{i+1}) \cap V = \emptyset$	$WY_i \cap XYZ_{i+1} = \emptyset$ $XYZ_i \cap WY_{i+1} = \emptyset$ $(X_i \cup Z_i) \cap (X_{i+1} \cup Z_{i+1}) \cap V = \emptyset$	$WY_k \cap XYZ_l = \emptyset$†† $(X_k \cup Z_k) \cap ((X_{k+1} \cup Z_{k+1}) \cup \ldots \cup (X_n \cup Z_n)) \cap V = \emptyset$†	$WY_k \cap XYZ_l = \emptyset$†† $(X_k \cup Z_k) \cap ((X_{k+1} \cup Z_{k+1}) \cup \ldots \cup (X_n \cup Z_n)) \cap V = \emptyset$†
Contemporary (CT)	same as commutative	$(WY_i \cup Z_i) \cap XYZ_{i+1} = \emptyset$ $XYZ_i \cap (WY_{i+1} \cup Z_{i+1}) = \emptyset$ $(X_i \cup Y_{i+1}) \cap V = \emptyset$	same as commutative	$(WY_k \cup Z_k) \cap XYZ_l = \emptyset$†† $Y_k \cap (X_{k+1} \cup \ldots \cup X_n) \cap V = \emptyset$†
Consecutive (CC)		No conditions necessary as this implies $XYZ_k \cap WY_{k+1} \neq \emptyset$		

†for all k such that $1 \leqslant k < n$
††for all k such that $1 \leqslant k \leqslant n$ and for all l such that $1 \leqslant l \leqslant n$ and $l \neq k$

Code		Bernstein representation
$A1 \leftarrow B1 + C1;$	W	B1, B2
$A2 \leftarrow A1 * B1;$	X	A2
$C1 \leftarrow B1 + B2;$	Y	C1
	Z	A1

Fig. 1(*a*) **Fig. 1**(*b*)

$A3 \leftarrow B1 + B2;$	W	B1, D1
$B2 \leftarrow B1/D1;$	X	D2
$D2 \leftarrow A3 - D1;$	Y	B2
	Z	A3

Fig. 2(*a*) **Fig. 2**(*b*)

Result

$$XYZ_1 \cap WY_2 = (X_1 \cup Y_1 \cup Z_1) \cap (W_1 \cup Y_2) = \emptyset$$

$$XYZ_1 \cap XYZ_2 \cap V = (X_1 \cup Y_1 \cup Z_1) \cap (X_2 \cup Y_2 \cup Z_2) \cap V = \emptyset$$

$$WY_1 \cap XYZ_2 = (W_1 \cup Y_1) \cap (X_2 \cup Y_2 \cup Z_2) \neq 0$$

where $V \equiv 1$

Fig. 3 Example 1

5. Loops

Let us now consider a stanza which is also the body of a loop, i.e. *Do-stanza*. This stanza will be executed a number of times (the exact number depending on the control variable initially considered to be a constant step size). If each iteration is considered as a separate stanza, and we construct each stanza such that the relationship for each iteration can be formed. Then for the case where each iteration will only vary in locations accessed via the control variable certain short cuts are available which can be recognised by carrying out these tests in the following order.

1. *Total independence*
If every assignment is to members of arrays indexed by the control variable, and any mentioned array assigned to within the loop, does not at any stage have any member fetched, then, each iteration of that loop is completely independent of each other.

2. *Total dependence*
If the relationship between the first iteration (I_1) and the second iteration (I_2) of the loop is *consecutive*; then all iterations of the loop must be done sequentially, see (Evans and Smith, 1977).

3. *Partial dependence*
The relationship between the first iteration (I_1) and the ($k + 1$)th iteration (I_{k+1}) is the same as that between I_n and I_{k+n} (provided that $k + n$ is not greater than the number of iterations in the loop), and that the step size is constant for the loop.

Note for the loops being considered the only difference between two iterations will be the elements of arrays accessed by the control variable. Otherwise the loop would have been found to be totally dependent.

Thus, by establishing relationships between I_1 and I_k ($k > 1$ and $k \leqslant N$). For all k until one of the following is satisfied, i.e.

(*a*) the relationship is consecutive

(*b*) $k = N$.

From which we can state:

(*a*) We have established that $I_1, I_2, \ldots, I_k$ can be executed by the series of relationships found and the next k iterations must be done sequentially after these—but within that group they bear the same relationship to each other as the previous k do to each other.

(*b*) The whole loop is computed using the relationships already established.

Nested loops

Let us now consider a Do-stanza of which the execution is controlled by more than one control variable. The previous tests for single loops can be expanded provided that any array that is indexed by a control variable is not later indexed by the same control variable in a different subscript position (see later).

The extensions to the tests are:

1. *Total independence*
If this test is true for each of the loops, then all iterations will be *contemporary*.

2. *Total dependence*
For each loop $L_1, L_2, \ldots, L_l$ establish the relationship for I_1 and I_2. If this is *consecutive* then for that particular loop each of its iterations must be executed consecutively.

3. *Partial dependence*
For each loop $L_1, L_2, \ldots, L_l$ for which total dependence has not been shown we need to establish relationships between I_1 and I_k (for $k > 1$ and $k \leqslant N$) as described above for one loop.

The handling of both types of loops is discussed in more detail in Williams (1978). In Appendix 1 an example of how potential parallelism may be detected in a nested loop is given.

Variate positioning of control subscripts

If an array is indexed by a given control variable in one subscript position and is later indexed by the same control variable in a different subscript position, it becomes difficult to predict the usage of a particular element of the array. This practice does not appear to be very common. However, this situation can be coped with adequately by calculating for all the affected loops, the relationships (3.5)-(3.15) between all iterations of all such loops.

6. IF stanzas

Let us consider a simple ALGOL-type IF statement, the variables used here can be divided into three following categories and are those used

1. in testing the condition
2. if the condition is true
3. if the condition is false.

For an IF stanza S_N we can represent these categories by three Bernstein sets, i.e.

W_{NC} The variables tested in the conditional (*NB* These will only be fetched variables—so there is no need for a X, Y or Z set).

$$\left.\begin{array}{l} W_{NT} \\ X_{NT} \\ Y_{NT} \\ Z_{NT} \end{array}\right\} \text{The variables used, if the condition is true}$$

$$\left.\begin{array}{l} W_{NF} \\ X_{NF} \\ Y_{NF} \\ Z_{NF} \end{array}\right\} \text{The variables used, if the condition is false}$$

Thus for any execution of an IF stanza the variables used will be given by the sets,

$$W_{NC} \cup W_{NT} \cup X_{NT} \cup Y_{NT} \cup Z_{NT}$$

or

$$W_{NC} \cup W_{NF} \cup X_{NF} \cup Y_{NF} \cup Z_{NF},$$

depending upon the value of the condition.

Let us now look at the ways we can establish the relationship between:

(*a*) An assignment stanza S_{N-1} and S_N

(*b*) S_N and an assignment stanza S_{N+1}.

1. S_{N-1} and S_N

(*a*) Test that the variables used in the condition of S_N are not assigned to in the stanza S_{N-1}, i.e.

$$XYZ_{N-1} \cap W_{NC} = \emptyset \qquad (6.1)$$

Otherwise the stanzas S_{N-1} and S_N would need to be executed consecutively.

(*b*) If the relationship (6.1) is true then test the relationships between stanzas S_{N-1} and S_{NT} and stanzas S_{N-1} and S_{NF} in the manner described previously for adjacent assignment stanzas in Section 4. Given two relationships as formed above, say, R_1 and R_2, then it is possible to ascertain if the condition of S_N is true, then the relationship between S_{N-1} and S_N is R_1 otherwise it is R_2.

2. *S_N and S_{N+1}*

Again we need to establish the relationships R_1 and R_2 between S_{NT} and S_{N+1}. In addition, we will also need to include with both S_{NT} and S_{NF} the variables used in S_{NC} when establishing relationships R_1 and R_2, as this test will be carried out whether or not S_{NT} or S_{NF} is executed. So the input set for S_{NT} now becomes:

$$(W_{NT} \cup Y_{NT} \cup W_{NC})$$

and similarly the input set for S_{NF} becomes:

$$(W_{NF} \cup Y_{NF} \cup W_{NC}) .$$

So the commutativity test between S_{NT} and S_{N+1} using private memory (see Section 3) becomes:

$$(W_{NT} \cup Y_{NT} \cup W_{NC}) \cap XYZ_{N+1} = \emptyset \qquad (6.2)$$

$$(W_{NT} \cup Y_{NT} \cup Z_{NT}) \cap WY_{N+1} = \emptyset \qquad (6.1)$$

$$(X_{NT} \cup Z_{NT}) \cap (X_{N+1} \cup Z_{N+1}) = \emptyset \qquad (6.3)$$

so again, it is possible to calculate the two relationships, R_1 and R_2, and to state if the condition of S_N is true then the relationship between S_{NT} and S_{N+1} is R_1, otherwise it is R_2.

Multiple adjacent IF stanzas

When two adjacent IF stanzas are considered we examine four possible relationships (see **Fig. 4**) although only one of these will be used for any particular pass through these statements. In general the path to be taken will not be known until run time (Evans and Smith, 1977).

Since the number of paths through *n* adjacent IF stanzas is 2^n,

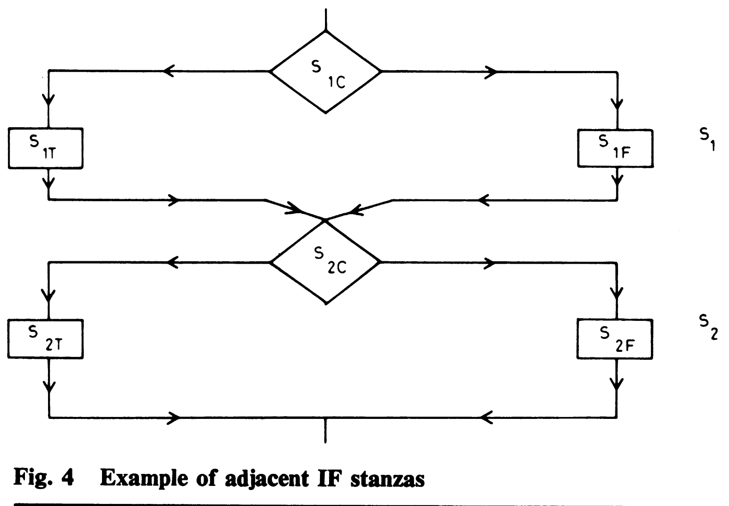

Fig. 4 Example of adjacent IF stanzas

then for practical purposes it will be necessary to limit the number of adjacent IF stanzas considered at any one time. However for two adjacent IF stanzas the extra work is not onerous and the gains should be worthwhile.

Appendix 2 gives an example of how parallelism may be detected between two IF stanzas.

7. Procedures

In ALGOL, a procedure is defined before it is called. In the body of the procedure we can use three types of variables:

1. Local variables
2. Global variables
3. Parameters

1. *The local variables* will have no effect on parallelism since by definition they cannot be used elsewhere.

2. *The global variables* are affected by the external environment and must be included in the Bernstein sets for this procedure.

3. *The actual variables* forming the parameters are not known until the procedure is called. However from the procedure declaration we will know if a variable is called by 'name' or by 'value'. Those parameters that are called by 'value' are of Bernstein type *W*, since they cannot be assigned to and those that are called by 'name' provide data to the procedure and can accept a result which can be considered to be of type *Y*. So for a procedure S_i, we can form the two Bernstein sets:

W_{iB}, X_{iB}, Y_{iB}, Z_{iB}—global variables (body)
W_{iP}, Y_{iP} —parameters

Thus for each time a call is made to that procedure the W_{iP} and Y_{iP} sets are formed and added to the body variables, checking that if the same variable appears again it is placed in the appropriate set (e.g. if a variable is in X_{iB} and W_{iP} and W_{iP} it should be put in Y_i). Then this new Bernstein set which has been formed can be used in comparison with what appears adjacent to the procedure call, in the same manner as described earlier.

8. Additional considerations

Robinson and Torsun (1976) have found that more than 85% of ALGOL 60 program statements are assignments, procedure calls, simple FOR loops and IF conditionals; we have indicated for all these areas where potential parallelism may be found.

The next largest area found in Robinson and Torsun (1976) were declarations (7½%). Although there is some potential for parallelism here, its feasibility will depend on how a particular machine configuration is able to allocate storage.

Other areas that should be considered are unconditional jumps (e.g. GOTO), loops which are iterated a conditional number (e.g. the step size or control variable may be assigned to within the body of the loop and WHILE clauses) and multiple case statements (e.g. SWITCH).

9. Implementation

Using the theory already outlined it is envisaged that it will be feasible to add two stages to an existing multi-pass compiler to detect potential parallelism. To this end two ALGOL 68-R programs have been constructed, copies of which are available from the authors.

1. *Analyser*

This program divides a given ALGOL-type language (Williams, 1978) into stanzas. Analyser arbitrarily limits the size of a stanza to be a specific program construct (e.g. a loop) or a collection of statements using not more than fifteen different variables. The variables used within a stanza are classified as belonging to the sets *W*, *X*, *Y* and *Z* depending on their usage.

Stanza S_1	W_i	X_i	Y_i	Z_i
$a1 \leftarrow b1 * b2;$	$b1, b2$	$a1$	—	—
$a2 \leftarrow a1 * b1;$	$b1, b2$	$a2$	—	$a1$
$c1 \leftarrow a1 + c1;$	$b1, b2$	$a2$	$c1$	$a1$

Fig. 5

Fig. 5 illustrates how these sets are formed for a stanza consisting of three assignment stanzas.

We believe that a thorough analysis of programs and parallel computers would be necessary to ensure the optimum size of a stanza and offer the above merely as a guideline.

2. *Detector*

This program will take the stanzas produced by the Analyser and carry out the tests described earlier to determine which parallel relationship if any exists between stanzas. The present version of the Detector program examines the relationship between pairs of stanzas or within a loop for a parallel computer system with private memories.

With the information from the Analyser and Detector programs it will be possible when compiling a serial program to identify parts of program that may be run in parallel. Williams (1978) details the programs described here and discusses possible extensions to these techniques. She also explains how some of the tasks carried out by the Analyser and Detector can be 'fitted' into a multi-pass compiler.

10. Conclusion

In this paper methods of finding parallelism within ALGOL-type programming languages have been discussed. Obviously, we may combine the techniques for handling two structures (e.g. a conditional in a loop or a loop in a conditional) and so will be able to handle most ALGOL-type programming situations. The way a program is written may greatly affect the location of parallelism, but by concentrating our efforts on the most frequently used programming structures it is hoped to find a large proportion of all possible parallelism occurring in a program.

Appendix 1 Example of Nested DO stanza

```
FOR i1 ← 1 STEP 1 UNTIL 10 DO                      ┐
  FOR i2 ← 1 STEP 1 UNTIL 10 DO               ┐    │
    FOR i3 ← 1 STEP 1 UNTIL 10 DO        ┐    │    │ L1
    BEGIN                                │    │ L2 │
      a[i1, i2, i3] ← b[i1, i2, i3 + 2];  │ L3 │    │
      b[i1, i2, i3] ← d;                 │    │    │
      c[i1, i2, i3] ← b[i1 + 3, i2, i3 + 3] │ │    │
    END                                  ┘    ┘    ┘
```

Bernstein Sets for the loop body

W d
X $a[\,,\,,\,], c[\,,\,,\,]$
Y $b[\,,\,,\,]$
Z $\emptyset$

The whole of the loops are not totally independent as one array (b) appears both on the righthand side and lefthand side of expressions within the body of the loop.

Loop-L3

W $d, b[i1 + 3, i2, i3 + 3], b[i1, i2, i3 + 2]$
X $a[i1, i2], c[i1, i2], b[i1, i2, i3]$
Y $\emptyset$
Z $\emptyset$

Since $b[i1, i2,\ ]$ appears in W and X then the loop $L3$ is not partially independent.

Bernstein's sets when $i3 \leftarrow 1$

W_1 $d, b[i1, i2, 3], b[i1 + 3, i2, 4]$
X_1 $a[i1, i2, 1], b[i1, i2, 1], c[i1, i2, 1]$
Y_1 $\emptyset$
Z_1 $\emptyset$

Bernstein's sets when $i3 \leftarrow 2$

W_2 $d, b[i1, i2, 4], b[i1 + 3, i2, 5]$
X_2 $a[i1, i2, 2], b[i1, i2, 2], c[i1, i2, 2]$
Y_2 $\emptyset$
Z_2 $\emptyset$

carrying out the relationship tests gives that these two iterations are contemporary.

Bernstein's sets when $i3 \leftarrow 3$

W_3 $d, b[i1, i2, 5], b[i1 + 3, i2, 6]$
X_3 $a[i1, i2, 3], b[i1, i2, 3], c[i1, i2, 3]$
Y_3 $\emptyset$
Z_3 $\emptyset$

again, carrying out the relationship tests gives that iteration 1 and iteration 3 are consecutive.

So for the whole of loop $L3$ the iterations can be carried out in pairs that are contemporaries, and each set of pairs must be executed consecutively. Using the notation where CC stands for consecutive, CT stands for contemporary and $L3_N$ is the N^{th} iteration of $L3$ we can write the relationship as:

$$\text{CC}(\text{CT}(L3_1, L3_2), \text{CT}(L3_3, L3_4), \ldots, \text{LT}(L3_9, L3_{10}))$$

Loop-L2

W $d, b[i1 + 3, i2]$
X $a[i1, i2], c[i1, i2]$
Y $b[i1, i2]$
Z

Since all arrays indexed by $i2$ only appear once loop $L2$ is partially independent (*NB* $b[i1,\ ,\ ]$ is a different array to $b[i1 + 3,\ ,\ ]$).

i.e. $\text{CT}(L2_1, L2_2, \ldots, L2_{10})$

Loop-L1

W $d, b[i1 + 3]$
X $a[i1], c[i1]$
Y $b[i1]$
Z $\emptyset$

Not partially independent because $b[\ ]$ appears in both W and Y.

Bernstein's sets when $i1 \leftarrow 1$

W_1 $d, b[4]$
X_1 $a[1], c[1]$
Y_1 $b[1]$
Z_1 $\emptyset$

Bernstein's sets when $i1 \leftarrow 2$

W_2 $d, b[5]$
X_2 $a[2], c[2]$
Y_2 $b[2]$
Z_2 $\emptyset$

carrying out the relationship tests gives that these two iterations are contemporary.

Bernstein's sets when $i1 \leftarrow 3$

W_3 $d, b[5]$
X_3 $a[3], c[3]$
Y_3 $b[3]$
Z_3 $\emptyset$

carrying out the relationship tests gives that the first and third iterations of loop $L1$ are contemporary.

Bernstein's sets with $i1 \leftarrow 4$

W_4	$d, b[6]$
X_4	$a[4], c[4]$
Y_4	$b[4]$
Z_4	$\emptyset$

carrying out the relationship tests gives that the first and fourth iterations of loop $L1$ are consecutive.

So the relationship is

$$CC(CT(L1_1, L1_2, L1_3), CT(L1_4, L1_5, L1_6)$$
$$CT(L1_7, L1_8, L1_9), L1_{10})$$

So assuming an availability of 60 processing units, the 1000 iterations can be executed in the time taken to execute 20 iterations of the loop sequentially.

Appendix 2 Example of two adjacent IF stanzas

```
IF a = b THEN          ┐
BEGIN                  │
  c ← d + 2*a;         │
  g ← c                │
END                    │
ELSE                   │  S1
BEGIN                  │
  c ← d + a + b;       │
  e ← f − c            │
END                    ┘
IF e = b THEN          ┐
BEGIN                  │
  f ← a + y;           │
  y ← b                │
END                    │  S2
ELSE                   │
BEGIN                  │
  y ← a − b;           │
  e ← a + b            │
END                    ┘
```

Bernstein sets for S_1 *and* S_2

W_{1C}	a, b
W_{1T}	d, a
X_{1T}	g
Y_{1T}	$\emptyset$
Z_{1T}	c
W_{1F}	d, a, b, f
X_{1F}	e
Y_{1F}	$\emptyset$
Z_{1F}	c
W_{2C}	e, b
W_{2T}	a, b
X_{2T}	f
Y_{2T}	y
Z_{2T}	$\emptyset$
W_{2F}	a, b
X_{2F}	y, e
Y_{2F}	$\emptyset$
Z_{2F}	$\emptyset$

1. S_{1T} *and* S_2 (using private memories)

$(X_{1T} \cup Y_{1T} \cup Z_{1T}) \cap W_{2C}$
$(g \cup \emptyset \cup c) \cap (e, b) = \emptyset$
$\therefore$not inherently consecutive.

(*a*) S_{1T} *and* S_{2T}

$(X_{1T} \cup Y_{1T} \cup Z_{1T}) \cap (W_{2T} \cup Y_{2T})$
$(g \cup \emptyset \cup c) \cap ((a, b) \cup y) = \emptyset$
$\therefore$consecutive.
$(X_{1T} \cup Y_{1T} \cup Z_{1T}) \cap (X_{2T} \cup Y_{2T} \cup Z_{2T})$
$(g \cup \emptyset \cup c) \cap (f \cup y \cup \emptyset) = \emptyset$
$\therefore$prerequisite.
$(W_{1T} \cup Y_{1T} \cup W_{1C}) \cap (X_{2T} \cup Y_{2T} \cup Z_{2T})$
$((d, a) \cup \emptyset \cup (a, n)) \cap (f \cup y \cup \emptyset) = \emptyset$
$\therefore$contemporary
i.e. S_{1T} and S_{2T} are contemporary.

(*b*) S_{1T} *and* S_{2F}

$(X_{1T} \cup Y_{1T} \cup Z_{1T}) \cap (W_{2F} \cup Y_{2F})$
$(g \cup \emptyset \cup c) \cap ((a, b) \cup \emptyset) = \emptyset$
$\therefore$conservative
$(X_{1T} \cup Y_{1T} \cup Z_{1T}) \cap (X_{2F} \cup Y_{2F} \cup Z_{2F})$
$(g \cup \emptyset \cup c) \cap ((y, e) \cup \emptyset \cup \emptyset) = \emptyset$
$\therefore$prerequisite
$(W_{1T} \cup Y_{1T} \cup W_{1T}) \cap (X_{2F} \cup Y_{2F} \cup Z_{2F})$
$((d, a) \cup \emptyset \cup (a, b)) \cap ((y, e) \cup \emptyset \cup \emptyset) = \emptyset$
$\therefore$contemporary
i.e. S_{1T} and S_{2F} are contemporary.

2. S_{1F} *and* S_2

$(X_{1F} \cup Y_{1F} \cup Z_{1F}) \cap W_{2C}$
$(e \cup \emptyset \cup c) \cap (e, b) \neq \emptyset$
$\therefore S_{1F}$ and S_2 are inherently consecutive.

So the possible relationships are

$$CT(S_{1T}, S_2) \quad \text{or} \quad CC(S_{1F}, S_2)$$

depending whether S_{1C} is true or false.

References

Anderson, J. P. (1965). Program Structures for Parallel Processing, *CACM*, Vol. 8 No. 12, pp. 786-788.

Bernstein, A. (1966). Analysis of programs for parallel processing, *IEEE Transactions on Electronic Computers*, EC 15 pp. 757-763.

Evans, D. J. and Smith, S. A. (1977). An Implicit Approach to Determining Parallelism in a Computer Program, Report No. Computer Studies 57, Loughborough University of Technology.

Kuck, D. J. (1975). Parallel Processing of Ordinary Programs, Report No. UIUCDCS-R-75-767, University of Illinois at Urbana-Champaign.

Ramamoorthy, C. B. and Gonzalez, M. J. (1969). A survey of techniques for recognising parallel processable streams in computer programs, *Proc. AFIPS Fall Joint Computer Conference*, pp. 1-15.

Robinson, S. K. and Torsun, I. S. (1976). Simplicity: An Empirical Analysis of Algol and Cobol, Private Communication.

Towle, R. A. (1976). Control and Data Dependence for Program Transformations, Report No. UIUCDCS-R-76-788, University of Illinois at Urbana-Champaign.

Wilkes, M. V. (1965). Slave Memories and Dynamic Storage Allocation, *IEEE Transactions on Electronic Computers*, EC 14 pp. 270-271.

Williams, S. A. (1978). Approaches to the Determination of Parallelism in Computer Programs, Ph.D. Thesis Loughborough University of Technology.