A survey of techniques for recognizing parallel processable streams in computer programs*

by C. V. RAMAMOORTHY and M. J. GONZALEZ

The University of Texas
Austin, Texas

INTRODUCTION

State-of-the-art advances—in particular, anticipated advances generated by LSI—have given fresh impetus to research in the area of parallel processing. The motives for parallel processing include the following:

1. Real-time urgency. Parallel processing can increase the speed of computation beyond the limit imposed by technological limitations.
2. Reduction of turnaround time of high priority jobs.
3. Reduction of memory and time requirements for "housekeeping" chores. The simultaneous but properly interlocked operations of reading inputs into memory and error checking and editing can reduce the need for large intermediate storages or costly transfers between members in a storage hierarchy.
4. An increase in simultaneous service to many users. In the field of the computer utility, for example, periods of peak demand are difficult to predict. The availability of spare processors enables an installation to minimize the effects of these peak periods. In addition, in the event of a system failure, faster computational speeds permit service to be provided to more users before the failure occurs.
5. Improved performance in a uniprocessor multiprogrammed environment. Even in a uniprocessor environment, parallel processable segments of high priority jobs can be overlapped so that when one segment is waiting for I/O, the processor can be computing its companion segment. Thus an overall speed up in execution is achieved.

With reference to a single program, the term "parallelism" can be applied at several levels. Parallelism within a program can exist from the level of statements of procedural languages to the level of micro operations. Throughout this paper, discussion will be confined to the more general "task" parallelism. The term "task" (process) generally is intended to mean a self-contained portion of a computation which once initiated can be carried out to its completion without the need for additional inputs. Thus the term can be applied to a single statement or a group of statements.

In contrast to the way the term "level" was used above, task parallelism can exist at several levels within a hierarchy of levels. The statements of the main program of a FORTRAN program, for example, are said to be tasks of the first level. The statements within a subroutine called by the main program would then be second level tasks. If this subroutine itself called another subroutine, then the statements within the latter subroutine would be of the third level, etc. Thus a sequentially organized program can be represented by a hierarchy of levels as shown in Figure 1. Each

* This work was supported by NASA Grant NGR 44-012-144.

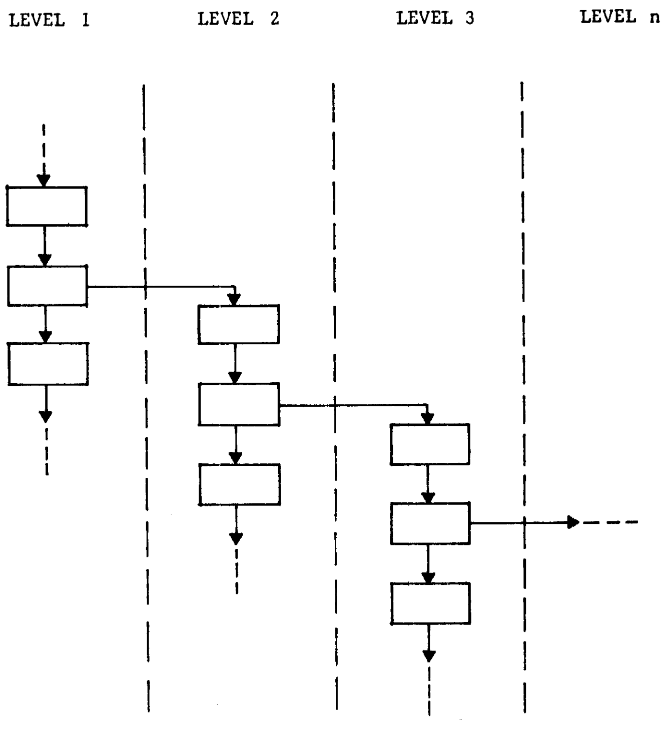


Figure 1—Hierarchical representation of a sequentially organized program

block within a level represents a single task; as before, a task can represent a statement or a group of statements.

Once a sequentially organized program is resolved into its various levels, a fundamental consideration of parallel processing becomes prominent—namely that of recognizing tasks within individual levels which can be executed in parallel. Assuming the existence of a system which can process independent tasks in parallel, this problem can be approached from two directions. The first approach provides the programmer with additional tools which enable him to explicitly indicate the parallel processable tasks. If it is decided to make this indication independent of the programmer, then it is necessary to recognize the parallel processable tasks implicitly by analysis of the relationship between tasks within the source program.

After the information is obtained by either of these approaches, it must still be communicated to and utilized by the operating system. At this point, efficient resource utilization becomes the prime consideration.

The conditions which determine whether or not two tasks can be executed in parallel have been investigated by Bernstein.[1] Consider several tasks, T_i, of a sequentially organized program illustrated by a flow chart as shown in Figure 2(a). If the execution of

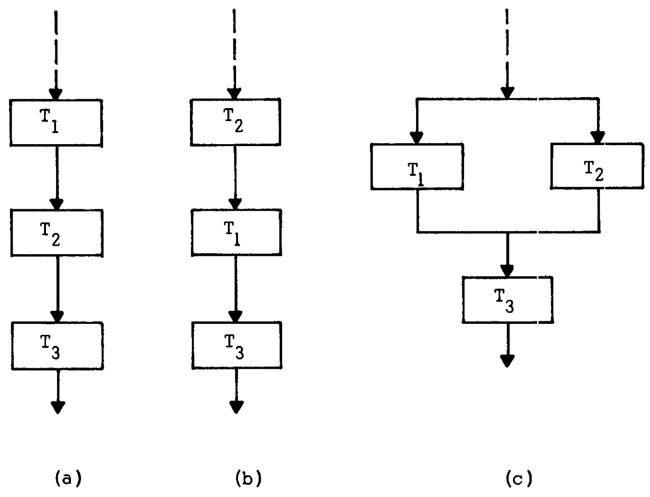


Figure 2—Sequential and parallel execution of a computational process

task T_3 is independent of whether tasks T_1 and T_2 are executed sequentially as shown in Figure 2(a) or 2(b), then parallelism is said to exist between tasks T_1 and T_2. They can, therefore, be executed in parallel as shown in Figure 2(c).

This "commutativity" is a necessary but not sufficient condition for parallel processing. There may exist, for instance, two processes which can be executed in either order but not in parallel. For example, the inverse of a matrix A can be obtained in either of the two ways shown below.

(1)	(2)
a) Obtain transpose of A	a) Obtain matrix of cofactors of A
b) Obtain matrix of cofactors of the transposed matrix	b) Transpose matrix of cofactors
c) Divide result by determinant of A	c) Divide result by determinant of A

Thus obtaining the matrix of cofactors and the transposition operation are two distinct processes which can be executed in alternate order with the same result. They cannot, however, be executed in parallel.

Other complications may arise due to hardware limitations. Two tasks, for example, may need to access the same memory. In this and similar situations, requests for service must be queuéd. Djkstra, Knuth, and Coffman[2,3,4] have developed efficient scheduling procedures for using common resources.

In terms of sets representing memory locations, Bernstein has developed the conditions which must be satisfied before sequentially organized processes can be executed in parallel. These are based on four separate ways in which a sequence of instructions can use a memory location:

(1) The location is only fetched during the execution of T_i.

(2) The location is only stored during the execution of T_i.

(3) The first operation within a task involves a fetch with respect to a location; one of the succeeding operations of T_i stores in this location.

(4) The first operation within a task involves a store with respect to a location; one of the succeeding operations of T_i fetches this location.

Assuming a machine model in which processors are allowed to communicate directly with the memory and multi-access operations are permitted, the conditions for strictly parallel execution of two tasks or program blocks can be stated as follows.

(1) The areas of memory which Task 1 "reads" and onto which Task 2 "writes" should be mutually exclusive, and vice-versa.

(2) With respect to the next task in a sequential process, Tasks 1 and 2 should not store information in a common location.

The conditions listed by Bernstein are sufficient to guarantee commutativity and parallelism of two program blocks. He has shown, however, that there do not exist algorithms for deciding the commutativity or parallelism of arbitrary program blocks.

As an example of what has been discussed here consider the tasks shown below which represent FORTRAN statements for evaluation of three arithmetic expressions.

$$X = (A+B) * (A-B)$$
$$Y = (C-D) / (C+D)$$
$$Z = X+Y$$

Because the execution of the third expression is independent of the order in which the first two expressions are executed, the first two expressions can be executed in parallel.

Parallelism within a task can also exist when individual components of compound tasks can be executed concurrently. In the same manner that individual processors can be assigned to independent tasks, individual functional units can be assigned to independent components within a task. The motivation remains the same—a decrease in execution time of individual tasks. The CDC 6600, for example, can utilize several arithmetic units to perform several operations simultaneously. This type of parallelism can be illustrated by the arithmetic expression which follows.

$$X = (A+B) * (C-D)$$

Normally, this expression would be evaluated in a manner similar to that shown in Figure 3(a). The independent components within the expression, however, permit parallel execution as shown in Figure 3(b) with the same results.

Explicit and implicit parallelsim

In the explicit approach to parallelism, the programmer himself indicates the tasks within a computational process which can be executed in parallel. This is normally done by means of additional instructions in the programming language. This approach can be illustrated by the techniques described by Conway, Opler, Gosden, and others[5,6,7]. FORK in the FORK and JOIN technique[5] indicates thep arallel processability of a specified set of tasks within a process. The next sequence of tasks will not be initiated until all

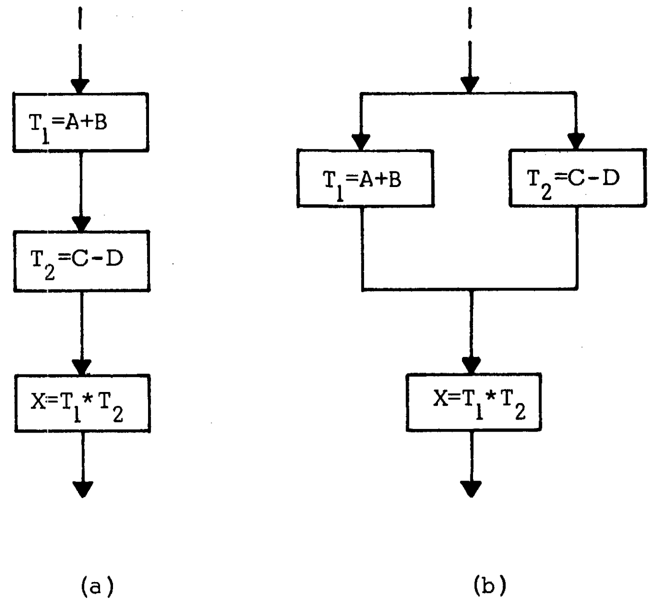


Figure 3—Illustration of parallelism within a compound task

the tasks emanating from a FORK converge to a JOIN statement.

In some instances, some of the parallel operations initiated by the FORK instruction do not have to be completed before processing can continue. For example, one of these branch operations may be designed to alert an I/O unit to the fact that it is to be used momentarily. The conventional FORK must be modified to take care of these situations. Execution of an IDLE statement, for example, permits processors to be released without initiation of further action.[7] The FORK and JOIN TECHNIQUE is illustrated in Figure 4.

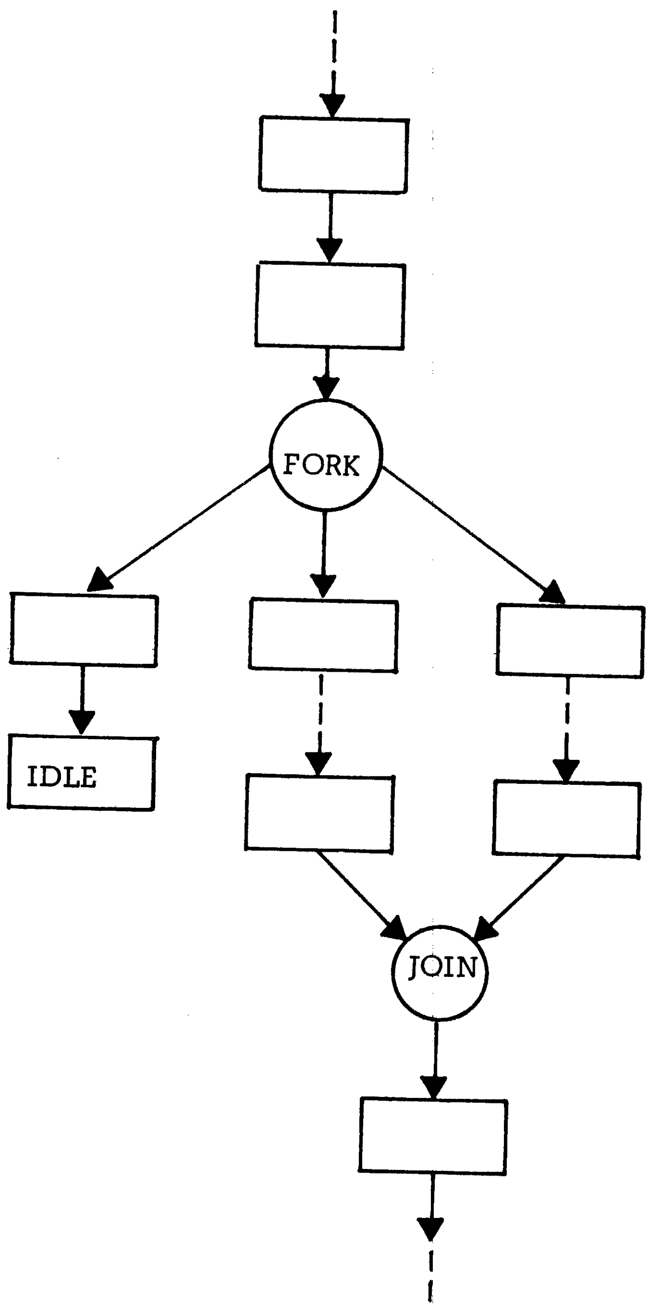


Figure 4—FORK and JOIN technique

Another example of the explicit approach is the PARALLEL FOR[7] which takes advantage of parallel operations generated by the FOR statement in ALGOL and similar constructs in other languages. For example, the sum of two $n \times n$ matrices consists essentially of n^2 independent operations. If n processors were available, the addition process could be organized such that entire rows or columns could be added simultaneously. Thus the addition of the two matrices could be accomplished in n units of time. Another example of this approach is the programming language PL/1 which provides the TASK option with the CALL statement which indicates concurrent execution of parallel tasks.

An additional way of indicating parallelism explicitly is to write a language which exploits the parallelism in algorithms to be implemented by the operating system. This is the case with TRANQUIL,[8,21] an ALGOL-like language to be utilized by the array processors of the ILLIAC IV. The situation is unique in that the language was created after a system was devised to solve an existing problem. "The task of compiling a language for the ILLIAC IV is more difficult than compiling for conventional machines simply because of the different hardware organization and the need to utilize its parallelism efficiently." A limitation of this approach is that programs written in that particular language can only be run on array-type computers and is, therefore, heavily machine dependent.

The implicit approach to parallelism does not depend on the programmer for determination of inherent parallelism but relies instead on indicators existing within the program itself. In contrast to the relative ease of implementation of explicit parallelism, the implicit approach is associated with complex compiling and supervisory programs.

The detection of inherent parallelism between a set of tasks depends on thorough analysis of the source program using Bernstein's conditions. Implementation of a recognition scheme to accomplish this detection is dependent on the source language. Thus a recognizer which is universally applicable cannot be implemented.

An algorithm developed by Fisher[9] approaches the problem of parallel task detection in a general manner. His algorithm utilizes the input and output sets of each task (process) to determine essential ordering and thus inherent parallelism. Given such information as the number of processes to be analyzed, the input and output set for each process, the given permissible ordering among the processes, and any initially known essential order among the processes, the algorithm generates the essential serial ordering relation and the covering for the essential serial ordering relation. This covering provides an indication of the tasks within the overall process which can be executed concurrently.

Basically, this work formalizes in the form of an algorithm the conditions for parallel processing developed by Bernstein. The conditions for parallel processing between two tasks are extended to an overall process

Detection of task parallelism—A new approach

The next subject covered in this paper involves implicit detection of parallel processable tasks within programs prepared for serial execution. An indication is desired of the tasks which can be executed in parallel and the tasks which must be completed before the start of the next sequence of tasks. Thus the problem can be broken down in two parts—recognizing the relationships between tasks within a level and using this information to indicate the ordering between tasks.

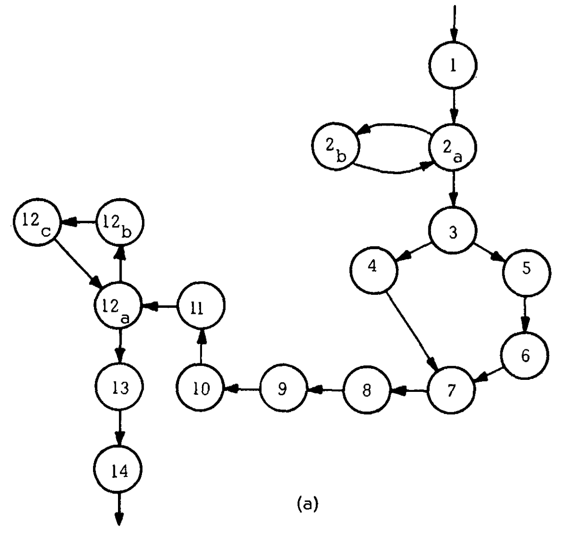
The approach presented here is based on the fact that computational processes can be modeled by oriented graphs in which the vertices (nodes) represent single tasks and the oriented edges (directed branches) represent the permissible transition to the next task in sequence. The graph (and thus the computational process) can be represented in a computer by means of a Connectivity Matrix, C.[10,11] C is of dimension $n \times n$ such that C_{ij} is a "1" if and only if there is a directed edge from node i to node j, and it is "0" otherwise. The properties of the directed graph and hence of the computational process it represents can be studied by simple manipulations of the connectivity matrix.

A graph consisting of a set of vertices is said to be *strongly connected* if and only if any node in it is reachable from any other. A *subgraph* of any graph is defined as consisting of a subset of vertices with all the edges between them retained. A *maximal strongly connected* (M.S.C.) *subgraph* is a strongly connected subgraph that includes all possible nodes which are strongly connected with each other. Given a connectivity matrix of a graph, all its M.S.C. subgraphs can be determined simply by well-known methods.[10] A given program graph can be *reduced* by replacing each of its M.S.C. subgraphs by a single vertex and retaining the edges connected between these vertices and others. After the reduction, the *reduced graph* will not contain any strongly connected components.

The paragraphs which follow will describe the sequence of operations needed to prepare for parallel processing in a multiprocessor computer a program written for a uniprocessor machine.

(1) The first step is to derive the *program graph* which identifies the sequence in which the computational tasks are performed in the sequentially code-program. Figure 5(a) illustrates an example program graph. The program graph is represented in the computer by its connectivity matrix. The connectivity matrix for the example is given in Figure 5(b).

(2) By an analysis of the connectivity matrix, the maximal strongly connected subgraphs are determined by simple operations.[10] This type of subgraph is illustrated by tasks 2 and 12 in Figure 5. Each M.S.C. subgraph is next considered as a single task, and the graph, called the reduced graph, is derived. The reduced graph does not contain any loops or strongly



(a)

	1	2_a	2_b	3	4	5	6	7	8	9	10	11	12_a	12_b	12_c	13	14
1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
2_a	0	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0
2_b	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
3	0	0	0	0	1	1	0	0	0	0	0	0	0	0	0	0	0
4	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0
5	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0
6	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0
7	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0
8	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0
9	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0
10	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0
11	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0
12_a	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	1	0
12_b	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0
12_c	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0
13	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1
14	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

(b)

Figure 5—Program graph of a serially coded program and its connectivity matrix

connected elements. In this graph, when two or more edges emanate from a vertex, a conditional branching is indicated. That is, the execution sequence will take only one of the indicated alternatives. A vertex which initiates the branching operation will be called a *decision* or *branch* vertex. The reduced graph for the example program graph is shown in Figure 6. In this graph, vertex 3 represents a branch vertex.

(3) The next step is to derive the final program graph and its connectivity matrix *T*. The elements of *T* are obtained by analyzing the inputs of each vertex in the reduced graph. An element, T_{ij}, is a "1" if and only if the j-th task (vertex) of the reduced graph has as one of its inputs the output of task i; otherwise T_{ij} is a "0". Figure 7 illustrates the final program for the example after consideration is given to the input-output relationships of each task. The connectivity matrix for the final program graph is shown in Figure 8.

From the sufficiency conditions for task parallelism, two tasks can be executed in parallel if the input set of one task does not depend on the output set of the other and vice versa. The technique outlined in Step 4 detects this relationship and uses it to provide an ordering for task execution.

(4) The vertices of the final program graph are partitioned into "precedence partitions"[12] as follows. Using the connectivity matrix *T*, a column (or columns) containing only zeroes is located. Let this column correspond to vertex v_1. Next delete from *T* both the column and the row corresponding to this vertex. The first precedence partiton is $P_1 = \{v_1\}$. Using the remaining portion of T, locate vertices $\{v_{21}, v_{22}, \ldots\}$ which correspond to columns containing only zeroes. The second precedence partition P_2 thus contains vertices $\{v_{21}, v_{22}, \ldots\}$. This implies that tasks in set $P_2 =$

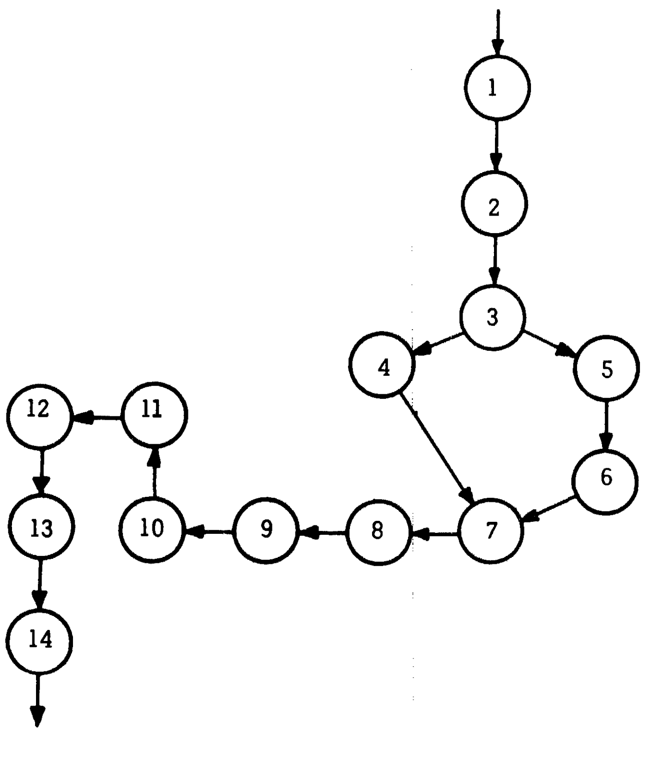


Figure 6—Reduced program graph of the serially coded program

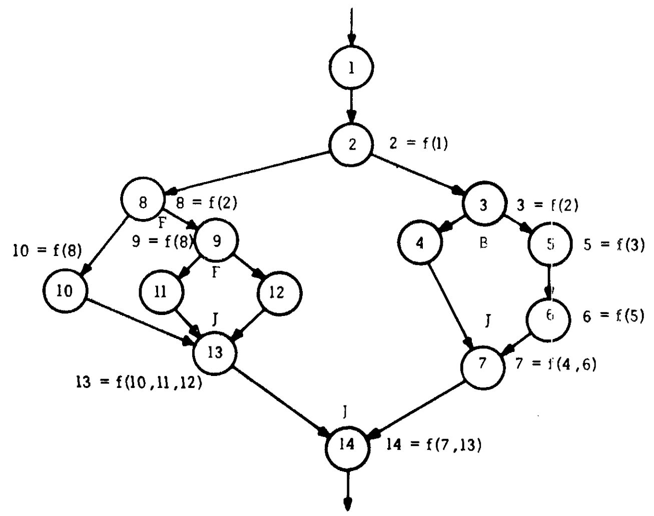


Figure 7—Final program graph of the parallel processable program

$$\underline{T} = \begin{array}{c|cccccccccccccc}
 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 & 14 \\
1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
2 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\
3 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
4 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
5 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
6 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
7 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\
8 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 \\
9 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 \\
10 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\
11 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\
12 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\
13 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\
14 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0
\end{array}$$

Precedence Partitions $\{1\}, \{2\}, \{3,8\}, \{4,5,9,10\}$
$\{6,11,12\}, \{7,13\}, \{14\}$

Figure 8—Connectivity matrix of the final program graph

$\{v_{21}, v_{22},\ldots\}$ can be initiated and executed in parallel after the tasks in the previous partition (i.e., P_1) have been completed. Next delete from T the columns and rows corresponding to vertices in P_2. This procedure is repeated to obtain precedence partitions $P_3,P_4,\ldots P_p$, until no more columns or rows remain in the T matrix. It can be shown that this partitioning procedure is valid for connectivity matrices of graphs which contain no strongly connected components.

The implication of this precedence partitioning is that if $P_1,P_2,\ldots P_p$ corresponds to times $t_1,t_2,\ldots t_p$, the earliest time that a task in partition P_i can be initiated is t_i.

The final program graph contains the following types of vertices: (1) The branch or decision type vertex from which the execution sequence selects a task from a set of alternative tasks. (2) The Fork vertex which can initiate a set of parallel tasks. (3) The Join vertex to which a set of parallel tasks converge after their execution. (4) The normal vertex which receives its input set from the outputs of preceding tasks. Figure 7a indicates the final program graph with the first three types of vertices indicated by B, F, and J, respectively.

(5) From precedence partitioning and the final program graph, a Task Scheduling Table can be developed. This table, shown in Table I, serves as an input to the operating system to help in the scheduling of tasks. For example, if the task being executed is a Fork task, a look-ahead feature of the system can prepare for parallel execution of the tasks to be initated upon completion of the currently active task.

(6) The precedence partitions of Step 4 provide an indication of the *earliest* time at which a task may be initiated. It is also desirable, however, to provide an indication of the *latest* time at which a task may be initiated. This information can be obtained by performing precedence partitions on the transpose of the T matrix. This process can be referred to as "row partitions". The implication here is that if task is in the partition corresponding to time period t_k, then t_k is the latest time that the task i can be initiated.

Using both the row and column partitions, the permissible initiation time for each task can be derived as shown in Table II. Task 4, for example, can be initiated during t_4 or t_5 depending on the availability of processors.

At this point it is desirable to clarify some possible misinterpretations of the implications of this method. The method presented here does not try to determine whether any or all of the iterations within a loop can be executed simultaneously. Rather the iterations executed sequentially are considered as a single task.

TABLE I—Task scheduling table

TIME	INPUTS TO TASKS	TASK NUMBER	TASK TYPE
t_1	-	1	
t_2	1	2	FORK
t_3	2	3	BRANCH
t_3	2	8	FORK
t_4	3	4	
t_4	3	5	
t_4	8	9	FORK
t_4	8	10	
t_5	5	6	
t_5	9	11	
t_5	9	12	
t_6	4,6	7	JOIN
t_6	10,11,12	13	JOIN
t_7	7,13	14	JOIN

For this reason, the undecidability problem introduced by Bernstein is not a factor here.

In addition, precedence partitions may place the successors of a conditional within the same partition. The interpretation of this is that only one of the successors will be executed, and it can be executed in parallel with the other tasks within that partition.

The FORTRAN parallel task recognizer

In order to determine the degree of applicability of the method described above, it was decided to apply the method to a sample FORTRAN program. This was accomplished by writing a program whose input consists of a FORTRAN source program; its output consists of a listing of the tasks within the *first level* of the source program which can be executed in parallel. The program written to accomplish this parallel task

TABLE II—Permissible task initiation time

COLUMN PARTITIONS		PERMISSIBLE TASK INITIATION PERIODS	
TIME	TASK		
t_1	1	TASK	TIME
t_2	2	1	t_1
t_3	3,8	2	t_2
t_4	4,5,9,10	3	t_3
t_5	6,11,12	4	t_4,t_5
t_6	7,13	5	t_4
t_7	14	6	t_5
ROW PARTITIONS		7	t_6
t_1	1	8	t_3
t_2	2	9	t_4
t_3	3,8	10	t_4,t_5
t_4	5,9	11	t_5
t_5	4,6,10,11,12	12	t_5
t_6	7,13	13	t_6
t_7	14	14	t_7

detection is known in its final form as a FORTRAN Parallel Task Recognizer.[13]

The recognizer, also written in FORTRAN, relies on indicators generated by the way in which the program is actually written. Consider the expressions given below.

$$X1 = f_1(A,B)$$
$$X2 = f_2(C,D)$$

Because the right-hand side of the second expression does not contain a parameter generated by the computation which immediately precedes it, the two expressions can be executed in parallel. If, on the other hand, the expressions were rewritten as shown below, the termination of the first computation would have to precede the initiation of the second.

$$X1 = f_1(A,B)$$
$$X2 = f_2(X1,C)$$

The recognizer performs this determination by comparing the parameters on the right-hand of the equality sign to outcomes generated by previous statements.

Other FORTRAN instructions can be analyzed similarly. Consider the arithmetic IF:

IF (X − Y) 3,4,5

Here the parameters within the parentheses must be compared to the outputs of preceding statements in order to determine essential order.

Other FORTRAN instructions are analyzed in a similar manner in order to generate the connectivity matrix for the source program. During this analysis the recognizer assigns numbers to the executable statements of the source program. After this is completed, the recognizer proceeds with the method of precedence partitions described earlier. Precedence partitions yield a list of blocks which contain the statement numbers which can be executed concurrently.

Figure 9 shows a block diagram of the steps taken by the recognizer to generate the parallel processable tasks within the first level of a FORTRAN source program.

Some statements within the FORTRAN set are treated somewhat differently. The DO statement, for example, does not itself contain any input or output parameters but instead generates a series of repeated operations. Because of the loop considerations mentioned earlier, and because the rules of FORTRAN require entrance into a loop only through the DO statement, all the statements contained within a DO loop are considered as a single task. A loop, however, may contain a large number of statements, and a great amount of potential parallelism may be lost if consideration is not given to the statements within the loop. For this reason, the recognizer generates a separate connectivity matrix for each DO loop within the program.

The recognizer itself possesses limitations which must be eliminated before it can be applied to programs of a complex nature. For example, only a subset of the entire FORTRAN set is considered for recogniton. This could be corrected by expanding the recognition process to include a more complete set of instructions.

In addition to the DO statement, loops can also be

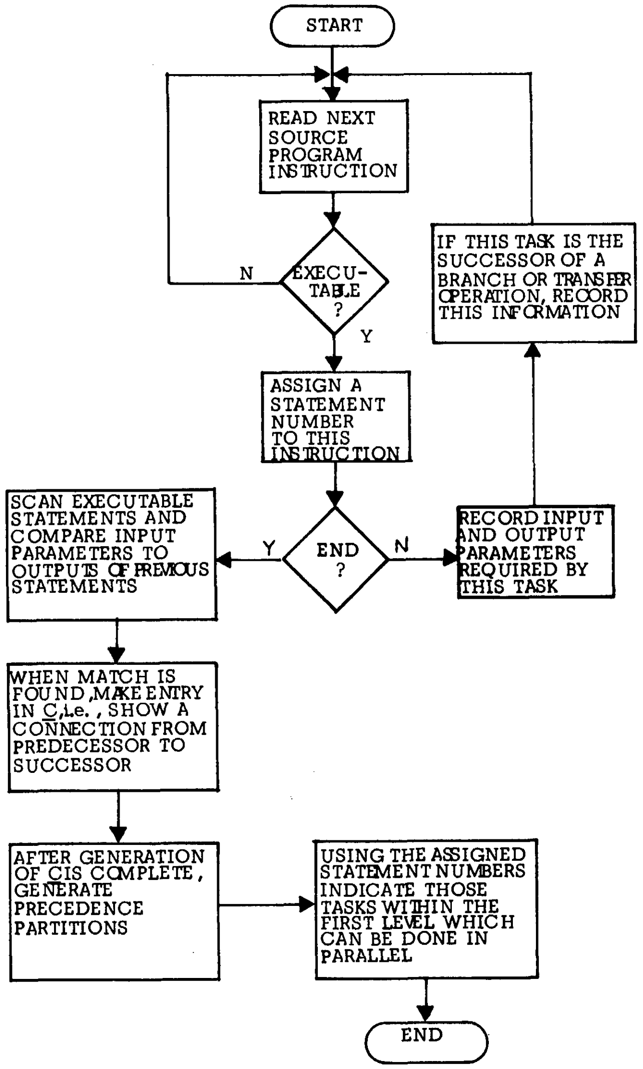

START
READ NEXT SOURCE PROGRAM INSTRUCTION
EXECU- TABLE ?
N
Y
IF THIS TASK IS THE SUCCESSOR OF A BRANCH OR TRANSFER OPERATION, RECORD THIS INFORMATION
ASSIGN A STATEMENT NUMBER TO THIS INSTRUCTION
END ?
Y
N
SCAN EXECUTABLE STATEMENTS AND COMPARE INPUT PARAMETERS TO OUTPUTS OF PREVIOUS STATEMENTS
RECORD INPUT AND OUTPUT PARAMETERS REQUIRED BY THIS TASK
WHEN MATCH IS FOUND,MAKE ENTRY IN C,i.e., SHOW A CONNECTION FROM PREDECESSOR TO SUCCESSOR
AFTER GENERATION OF CIS COMPLETE, GENERATE PRECEDENCE PARTITIONS
USING THE ASSIGNED STATEMENT NUMBERS INDICATE THOSE TASKS WITHIN THE FIRST LEVEL WHICH CAN BE DONE IN PARALLEL
END

Figure 9—Block diagram of the FORTRAN parallel task recognizer

```
     C        THIS IS A TEST PROGRAM DESIGNED TO CHECK PPS
              DIMENSION A1(10),A2(10),A3(10)
              INTEGER A1,A2,ABC,A2,X2,B,C,D
 1            READ 100,  (A1(I),I=1,10),B,C,D
 2            READ 100,  (A2(I),I=1,10),NS,NST,NSTU
 3            DO 10  I=1,10
 4            IF(A1(I)-A2(I))20,30,40
 5       20   X1=(A1(I))*(B-C)
 6            X2=D+(B/C)
 7            A3(I)=X1*X2
 8       10   CONTINUE
     C        THIS IS A TEST COMMENT
 9       30   PRINT 200,B,C,D
10       40   CALL ALPHA(A1,A2,ABC,B4,B5)
11            PRINT 3057,X1,X2,(A3(I),I=1,10)
12            CALL BETA(X1,X2,A3,B6)
13            IF(B4-B5)50,50,60
14       50   READ 315,E,F,G,H
15            X3=(E*F)+(G-H)
16            X4=B6+G                                              PARALLEL
17            X5=X3-X4                                             PROCESSABLE
18            X6=(B4+B5)*X5                                        TASKS
19       60   PRINT 4,X3,X4,X5                                        (1,2)
20            PRINT 52,(A1(I),I=1,10),ABC,C,(A3(I),I=1,10)           (3)
        100   FORMAT(10I2,3I3)                                        (9,10,11,12)
        200   FORMAT(1H0,8  B  C  D*,/,3I3)                           (13)
       3057   FORMAT(1H  ,2I3,10F7.1)                                 (14)
        315   FORMAT(4F7.4)                                           (15,16)
          4   FORMAT(3F7.4)                                           (17)
         52   FORMAT(12I3,10F7.1)                                     (18,19,20)
21            END

                      (a)                                              (b)
```

Figure 10—An example of the recognition process.

created by branch and transfer operations such as the IF and GO TO instructions. To eliminate these loops, it would be necessary to analyze the connectivity matrix in the manner mentioned earlier before beginning the process of precedence partitions. The recognizer does not presently perform this analysis.

Nested DO loops are not permitted, and the source program size is limited in the number of executable statements it may have and in the number of parameters any one statement can contain.

Some of these limitations could be eliminated quite easily; others would require a considerable amount of effort. To allow a source program of arbitrary size would require a somewhat more elaborate handling of memory requirements and associated problems. At the present time the recognizer consists of a main program and six subroutines. In its present form the recognizer consists of approximately 1300 statements.

The recognizer is presently written in such a manner that it will detect only first level parallelism. The method it uses, however, can be applied to parallelism at any level.

The theory of operation of the FORTRAN parallel task recognizer will be illustrated by applying the recognition techniques to a sample FORTRAN program. Figure 10(a) is a listing of the sample program showing the individual tasks. Figure 10(b) is a listing of the parallel processable tasks as determined by precedence partitions. The numbers to the left of the executable statements are the numbers assigned by the recognizer during the recognition phase.

Elimination of the limitations mentioned here and other limitations not mentioned explicitly will be the subject of future effort.

Observations and comments

Regardless of the manner in which the subject of parallel processing is approached, common problems arise. Prominent among these is a need to protect common data. If two tasks are considered for concurrent execution and one task accesses a memory location and the other amends it, then strict observance must be paid to the order in which this is done. The

FORTRAN recognizer, for example, may determine that two subroutines can be executed in parallel. At the present time no consideration is given to the fact that both subroutines may access common data through COMMON or EQUIVALENCE statements.

In order to truly optimize execution time for a program which is set up for parallel processing, it would be highly desirable to determine the time required for execution of the individual tasks within the process. It is not enough to merely determine that two tasks can be executed concurrently; the primary goal is that this parallel execution result in higher resource utilization and improved throughput. If the time required for the execution of one task is 100 times that of the other, for example, then it may be desirable to execute the two tasks serially rather than in parallel. The reasoning here is that no time would be spent in allocating processors and so forth.

Determination of task execution time, however, is not a simple matter. Exhaustive measurements of the type suggested by Russell and Estrin[14] would provide the type of information mentioned here.

Another problem area involves implementation of special purpose languages such as TRANQUIL. It was mentioned earlier that programs written in a language of this type are highly machine-limited. It would be highly desirable to be able to implement programs written in these languages in systems which are not designed to take advantage of parallelism. Along these lines, the programming generality suggested by Dennis[15] may be significant.

It should be pointed out that all the techniques which have been discussed here will create a certain amount of overhead. For this reason it is felt that a parallel task recognizer, for example, would be best suited for implementation with production programs. Thus even though some time would be lost initially, in the long run parallel processing would result in a significant net gain.

Conclusions

The method of indicating parallel processable tasks introduced here and illustrated in part by the FORTRAN Parallel Recognizer appears to provide enough generality that it is independent of the language, the application, the mode of compilation, and the number of processors in the system. It is anticipated that this method will remain as the basis for further effort in this area.

In addition to the comments made earlier, some possible future areas of effort include determination of possible parallelism of individual iterations within a loop. It is hoped that additional information can be provided to the operating system other than a mere indication of the tasks which can be executed in parallel. This would include the measurements mentioned earlier and an indication of the frequency of execution of individual tasks.

It is also hoped that a sub-language may be developed which can be added to existing languages to assist in the recognition process and the development of recognizer code.

Detection of parallel components within compound tasks

Several algorithms exist for the detection of independent components within compound tasks.[16,17,18,19] These algorithms are concerned primarily with detection of this type of parallelism within arithmetic expressions. The first three algorithms referenced above are summarized in [19] where a new algorithm is also introduced.

The arithmetic expression which will be used as an example for each algorithm is given below.

$$A + B + C + D * E * F + G + H$$

Throughout this discussion the usual precedence between operators will apply. In order of increasing precedence, the precedence between operators will be as follows: + and —, * and/, and $\uparrow$, where $\uparrow$ stands for exponentiation.

Hellerman's algorithm

This algorithm assumes that the input string is written in reverse Polish notation and contains only binary operators. The string is scanned from left to right replacing by temporary results each occurrence of adjacent operands immediately followed by an operator. These temporary results will be considered as operands during the next passes. Temporary results generated during a given pass are said to be at the same level and therefore can be executed in parallel. There will be as many passes as there are levels in the syntactic tree. The compilation of the expression listed above is shown in Figure 11.

Although this algortihm is simple and fast, it has two shortcomings. The first is a possible difficulty in implementation since it requires the input string to be in Polish notation; the second is its inability to handle operators which are not commutative.

i	INPUT STRING AFTER THE ith PASS	TEMPORARY RESULTS GENERATED DURING ith PASS
0	AB+C+DE*F*+G+H+	
1	R1 C+R2 F*+G+H+	R1=A+B R2=D*E
2	R3 R4+G+H+	R3=R1+C R4=R2*F
3	R5 G+H+	R5=R3+R4
4	R6 H+	R6=R5+G
5	R7	R7=R6+H

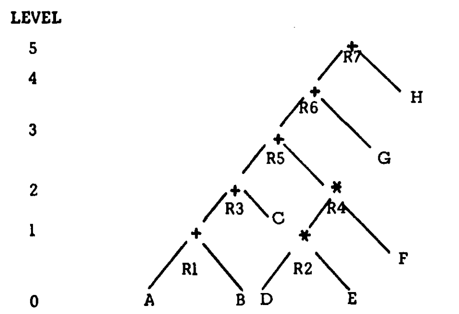


Figure 11—Parallel computation of A+B+C+D*E*F+G+H using Hellerman's algorithm

Stone's algorithm

The basic function of this algorithm is to combine two subtrees of the same level into a level that is one higher. For example, A and B, initially of level 0, are combined to form a subtree of level 1. The algorithm then searches for another subtree of level 1 by attempting to combine C and D. Since precedence relationships between operators prohibit this combination, the level of subtree (A+B) is incremented by one. The algorithm now searches for a subtree of level 2 by attempting to combine C, D, and E. Since this combination is also prohibited, subtree (A+B) is incremented to level 3. The next search is successful, and a subtree of level 3 is obtained by combining C, D, E and F. These two subtrees are then combined to form a single subtree of level 4.

In a similar manner the subtree (G+H), originally of level 1, is successively incremented until it achieves a level of 4; at that time it is combined with the other subtree of the same level to form a final tree of level 5.

The algorithm yields an output string in reverse Polish which does not expressly show which operations can be performed in parallel. Even though the output string is generated in one pass, the recursiveness of the algorithm causes it to be slow, and at least one additional pass would be required to specify parallel computations.

Squire's algorithm

The goal of this algorithm is to form quintuples of temporary results of the form:

Ri (operand 1, operator, operand 2, start level =max [end level op. 1; end level op. 2], end level= start level+1).

All temporary results which have the same start level can be computed in parallel. Initially, all variables have a start and end level equal to zero.

Scanning begins with the rightmost operator of the input string and proceeds from right to left until an operator is found whose priority is lower than that of the previously scanned operator. In the example the scan would yield the following substring:

D*E*F+G+H

Now a left to right scan proceeds until an operator is found whose priority is lower than that of the leftmost operator of the substring. This yields: D*E*F. At this point a temporary result R1 is available of the form:

R1(D,*,E,0,1).

The temporary result, R1, replaces one of the operands and the other is deleted together with its left operator The new substring is then:

R1*F+G+H.

The left to right scans are repeated until no further qunituple can be produced, and at that time, the right to left scan is re-initiated. The results of the process are shown in Figure 12.

Although the example shows the algorithm applied to an expression containing only binary operators, the algorithm can also handle subtraction and division with a corresponding increase in complexity.

A significant feature of this algorithm is that Polish notation plays no part in either the input string or the output quintuples. Because of the many scans and comparisons the algorithm requires, it becomes more complex as the length of the expression and the diversity of operators within the expression increase.

INITIAL STRING: A+B+C+D*E*F+G+H

RIGHT TO LEFT SCAN	LEFT TO RIGHT SCAN
D*E*F+G+H	R1*F+G+H
	R2+G+H
A+B+C+R2+G+H	R3+C+R2+G+H
	R4+R3+R2+H
	R4+R5+R2
	R6+R2
	R7

QUINTUPLES	Op.1	OPERATOR	Op.2	START	END
R1	D	*	E	0	1
R2	F	*	R1	1	2
R3	A	+	B	0	1
R4	C	+	G	0	1
R5	H	+	R3	1	2
R6	R4	+	R5	2	3
R7	R2	+	R6	3	4

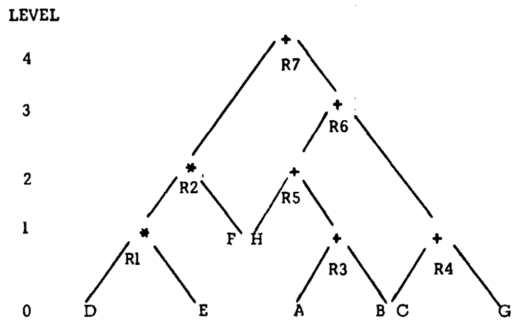


Figure 12—Parallel computation of A+B+C+D*E*F+G+H using Squire's algorithm

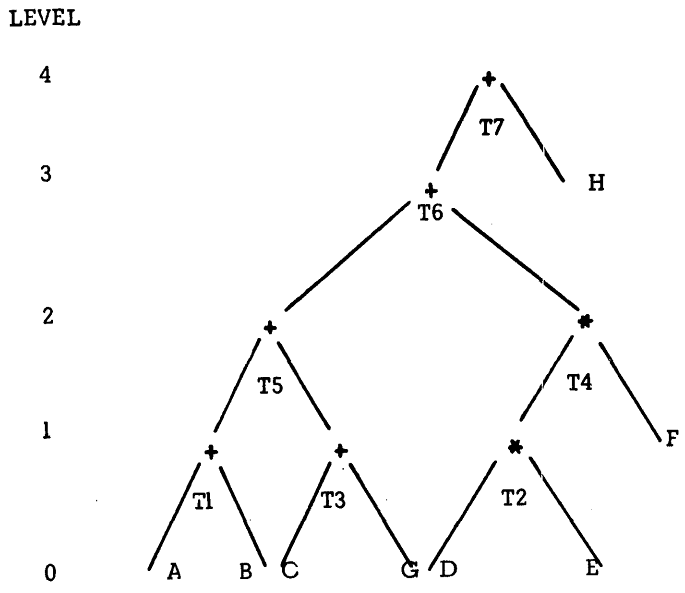


Figure 13—Parallel computation of A+B+C+D*E*F+G+H using Baer and Bovet's algorithm

Baer and Bovet's algorithm

The algorithm uses multiple passes. To each pass corresponds a level. All temporary results which can be generated at that level are constructed and inserted appropriately in the output string produced by the corresponding pass. Then, this output string becomes the input string for the next level until the whole expression has been compiled. Thus the number of passes will be equal to the number of levels in the syntactic tree. During a pass the scanning proceeds from left to right and each operator and operand is scanned only once.

The simple intermediate language which this algorithm produces is the most appropriate for multiprocessor compilation in that it shows directly all operations which can be performed in parallel, namely those having the same level number. The syntactic tree generated by this algorithm is shown in Figure 13.

A new algorithm

This section will introduce a technique whose goals are: (1) to produce a binary tree which illustrates the parallelism inherent in an arithmetic expression; and (2) to determine the number of registers needed to evaluate large arithmetic or Boolean expressions without intermediate transfers to main memory.

This technique is prompted by the fact that existing computing systems possess multiple arithmetic units which can contain a large number of active storages (registers). In addition, the superior memory bandwidths of the next generation of computers will simplify some of the requirements of this technique.

In the material presented below, a complex arithmetic expression is examined to determine its maximum computational parallelism. This is accomplished by repeated rearrangement of the given expression. During this process the given expression in reverse Polish form is also tested for "well formation", i.e., errors and oversights in the syntax, etc.

The arithmetic expression which was used as a model earlier will also be used here, namely A+B+C+D*E*F+G+H. The details of the algorithm follow:

(1) The first step is to rewrite the expression in reverse Polish form and to reverse its order.

+H+G+*F*E D+C+B+A

(2) Starting with the rightmost symbol of the string, assign a weight to each member of the string based on the following procedure:

Assign to symbol S_i the value $V_i = (V_{i-1}) + R_i$ $i = 1,2,\ldots,n$

where $R_i = 1 - \delta(S_i)$ given that

$\delta(S_i) = 0$ if S_i is a variable

$\delta(S_i) = 1$ if S_i is a unary operator

$\delta(S_i) = 2$ if S_i is a binaryoperator

and $V_{i-1} = V_{i-2}+R_{i-1}$, $V_{i-2} = V_{i-3} + R_{i-2}$, etc.,

such that $V_{i-(i-1)} = V_1 = R_1$, and $V_0 = 0$

Using this procedure, the following expression results:

	Root Node						
i	$\boxed{15}$	14	13	12	|	11	10
S_i	$\boxed{+}$	H	+	G	|	+	*
V_i	$\boxed{1}$	2	1	2	|	1	2

		V_m						
9	8	$\boxed{7}$	6	5	4	3	2	1
F	*	$\boxed{E}$	D	+	C	+	B	A
3	2	$\boxed{3}$	2	1	2	1	2	1

Note that for a "well-formed expression" of n symbols $V_n = 1$.

(3) At this point the root node of the proposed binary tree can be determined. Thus the given string can be divided into two independent sub-strings. To determine the root node, draw a line to the left of the first symbol with a weight of 1 ($i = 11$, $S_i = +$, $V_i = 1$) to the left of the symbol with the highest weight, $V_m(i=7, S_i=E, V_i=V_m=3)$. The two independent substrings consist of the strings to the left and to the right of this line. The root node will be the leftmost member of the string to the left of the line ($i=15$, $S_i=+$, $V_i=1$). Note that V_i also equals 3 for $i=9$; however V_m is chosen from the earliest occurrence of a symbol with the highest weight.

(4) The next step is to look for parallelism withni each of the new substrings. Consider the rightmost substring. Form a new substring consisting of the symbols within the values of $V_i=1$ to the right and to the left of V_m. Transpose this substring with the substring to the right of it whose leftmost member has a weight of $V_i=1$.

INITIAL RIGHTMOST	S_i	+	*	F	*	E	D	+	C	+	B	A
SUBSTRING	V_i	1	2	3	2	3	2	1	2	1	2	1

$$\longrightarrow \quad \longleftarrow$$

FINAL RIGHTMOST	i	11	10	9	8	7	6	5	4	3	2	1
SUBSTRING	S_i	+	+	C	+	B	A	*	F	*	E	D
	V_i	1	2	3	1	3	2	1	2	1	2	1

This procedure is repeated until the initial V_m occupies the position $i=2$ in the substring. For this example this is already the case. Thus the rightmost substring is in the proper form.

(5) The transposition procedure of step 4 is applied next to the leftmost substring. However, since the leftmost substring of this example consists of only two operands and one operator, no further operations are necessary.

(6) The resultant binary tree is shown in Figure 14. The numbers assigned to each node represent the final weight V_i of the symbol as determined in steps 1–5 above.

Some observations and comments on this algorithm are given below.

(1) The two branches on either side of the root node can be executed in parallel. Within each main branch, the transposition procedure of step 4 yields supplementary root nodes. The sub-branches on each side of the supplementary nodes can be executed in parallel.

(2) The number of levels in the binary tree can be

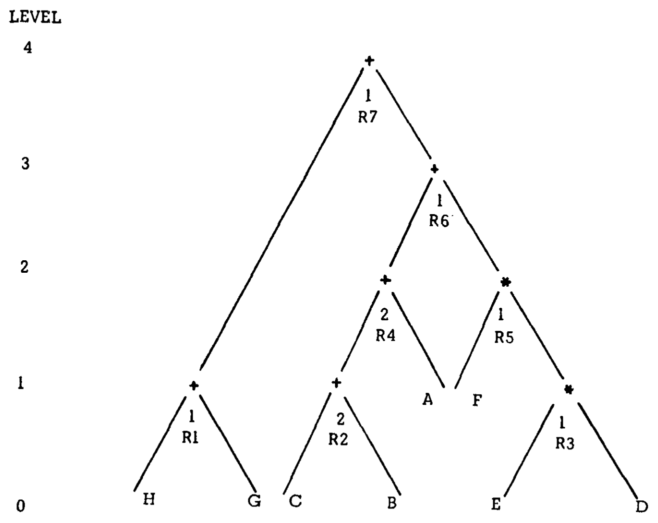


Figure 14—Binary tree for parallel computation of A+B+C+D*E*F+G+H

predicted from the Polish form of the original string.

No. of LEVELS = MAX [NUMBER OF 1's; Vm] in the substring (rightmost or leftmost) containing Vm.

(3) The tree is traversed in a modified postorder form.[20] The resulting expression is

$$D*E*F+A+B+C+G+H$$

(4) An added feature of this technique is that the number of registers required to evaluate this expression without intermediate STORE and FETCH operations is obtained directly from the binary tree. This information is provided by the highest weight assigned to any node within the tree. Thus for this example the expression could be evaluated using at most two registers without resorting to intermediate stores and fetches.

(5) This technique of recognizing parallelism on a local level has been applied to a single instruction, in particular, an arithmetic expression. It is worthwhile mentioning that each variable within the expression can itself be the result of a processable task. Thus this technique can be extended to a higher level of parallel stream recognition, i.e., level parallelism.

In order to implement the techniques mentioned here for components within tasks and the techniques mentioned earlier for individual tasks, several system features are desirable. Schemes for detecting parallel processable components within compound tasks are oriented primarily toward arithmetic expressions. For these situations string manipulation ability would be highly desirable. Since individual tasks are represented by a graph and its matrix, the ability to manipulate rows and columns easily would be very important. In this same area, an associative memory could greatly reduce execution time in the implementation of precedence partitions.

ACKNOWLEDGMENTS

The authors would like to thank the referees of the FJCC for their comments and suggestions which resulted in improvements of this paper.

REFERENCES

1 A J BERNSTEIN
Analysis of programs for parallel processing
IEEE Trans on EC Vol 15 No 5 757-763 Oct 1966

2 E W DJKSTRA
Solution of a problem in concurrent programming control
Comm ACM Vol 8 No 9 569 Sept 1965

3 D KNUTH
Additional comments on a problem in concurrent programming control
Comm ACM Vol 9 No 5 321-322 May 1966

4 E G COFFMAN R R MUNTZ
Models of pure time sharing disciplines for research allocation
Proc 1969 Natl ACM Conf

5 M E CONWAY
A multiprocessor system design
Proc FJCC Vol 23 139-146 1963

6 A OPLER
Procedure-oriented statements to facilitate parallel processing
Comm ACM Vol 8 No 5 306-307 May 1965

7 J A GOSDEN
Explicit parallel processing description and control in programs for multi- and uni-processor computers
Proc FJCC Vol 29 651-660 1966

8 N E ABEL P P BUDNIK D J KUCK
Y MURAOKA R S NORTHCOTE
R B WILHELMSON
TRANQUIL: A language for an array processing computer
Proc SJCC 57-68 1969

9 D A FISHER
Program analysis for multiprocessing
Burroughs Corp May 1967

10 C V RAMAMOORTHY
Analysis of graphs by connectivity considerations
Journal ACM Vol 13 No 2 211-222 April 1966

11 C V RAMAMOORTHY M J GONZALEZ
Recognition and representation of parallel processable streams in computer programs—II (task/process parallelism)
1969 Natl ACM Conf

12 C V RAMAMOORTHY
A structural theory of machine diagnosis
Proc SJCC 743-756 1967

13 M J GONZALEZ C V RAMAMOORTHY
Recognition and representation of parallel processable streams in computer programs
Symposia on Parallel Processor Systems Technologies and Applications Ed. L C Hobbs Spartan Books June 1969

14 E C RUSSELL G ESTRIN
Measurement based automatic analysis of FORTRAN programs
Proc SJCC 1969

15 J B DENNIS
Programming generality, parallelism and computer architecture
Proc IFIPS Congress 68 C1-C7

16 H HELLERMAN
Parallel processing of algebraic expressions
IEEE Trans on E C Vol 15 No 1 Feb 1966

17 H S STONE
One-pass compilation of arithmetic expressions for a parallel processor
Comm ACM Vol 10 No 4 220-223 April 1967

18 J S SQUIRE
A translation algorithm for a multiprocessor computer
Proc 18th ACM Natl Conf 1963

19 J L BAER D P BOVET
Compilation of arithmetic expressions for parallel computation

Proc IFIPS 68 B4-B10
20 D KNUTH
The art of computer programming, Vol. 1, fundamental algorithms
Addison-Wesley 316
21 R S NORTHCOTE
Software developments for the array computer ILLIAC IV,
Univ of Illinois Rpt No 313 March 1969