AD

AD650845

TECHNICAL REPORT ECOM -02463-2

DETECTION OF ESSENTIAL ORDERING IMPLICIT IN COMPILER LANGUAGE PROGRAMS

SECOND QUARTERLY REPORT

By

HARVEY W. BINGHAM
DAVID A. FISHER
WARREN L. SEMON

FEBRUARY 1967

DISTRIBUTION OF THIS DOCUMENT IS UNLIMITED

ECOM

UNITED STATES ARMY ELECTRONICS COMMAND · FORT MONMOUTH, N.J.
CONTRACT DA-28-043-AMC-02463(E)

Burroughs Corporation
Defense, Space and Special Systems Group
Paoli, Pennsylvania

ARCHIVE COPY

NOTICES

Disclaimers

The findings in this report are not to be construed as an official Department of the Army position, unless so designated by other authorized documents.

The citation of trade names and names of manufacturers in this report is not to be construed as official Government indorsement or approval of commercial products or services referenced herein.

Disposition

Destroy this report when it is no longer needed. Do not return it to the originator.

Technical Report ECOM 02463-2 February 1967

DETECTION OF ESSENTIAL ORDERING IMPLICIT IN COMPILER LANGUAGE PROGRAMS

Quarterly Technical Report
15 October 1966 to 20 January 1967

Report No. 2

Contract No. DA-28-043-AMC-02463(E)
DA Project No. 1E6-20501-A485-03-01

Prepared by

Harvey W. Bingham
David A. Fisher
Warren L. Semon

Burroughs Corporation
Defense, Space and Special Systems Group
Paoli, Pennsylvania

for

U. S. ARMY ELECTRONIC COMMAND
Fort Monmouth, N. J.

DISTRIBUTION OF THIS DOCUMENT IS UNLIMITED

ABSTRACT

This is the second report of an investigation to determine how implicit parallelism in programs written in compiler languages can be recognized and exploited by machines with highly parallel organizations. An algorithm is described which identifies the complete serial ordering among parts of a program based on the input-output sets of these parts, the ordering given by the programmer, and any known essential order among the program parts. The algorithm is proved and a demonstration given that a minimum number of comparisons of input-output sets are made. Application of the parallel recognition procedure to subroutines, loops, conditionals, recursive subroutines, and serial input-output device calls is explained. The effect of particular features of several compiler languages on parallelism are discussed. These features include loops, transfers of control, conditionals, and conditional sequences. Requirements for replacing iterative loop control by parallel paths of control are given. Alternative algorithms for recognizing essential ordering are suggested which can be executed more effectively on a highly parallel machine. Application of the given algorithm to the syntactic definition of a context-free language is also considered.

CONTENTS

	Page
ABSTRACT	ii
INTRODUCTION	1
ALGORITHM FOR DETECTING ESSENTIAL SERIAL ORDER	3
Definitions	6
Proof of Algorithm	8
Proof of Minimal Comparisons	13
ANALYSIS OF FORMAL PROGRAM STRUCTURES	16
Subroutines	16
Loops	16
Conditionals	17
Recursive Subroutines	18
Serial Input-Output Calls	19
PROGRAMMING LANGUAGE FEATURES EFFECTING PARALLELISM	20
Loops	20
FORTRAN DO Statement	21
ALGOL FOR Statement	23
COBOL PERFORM Statement	24
Unconditional Transfers	26
Conditionals	27
Sequence of Conditionals	27
Duration of Definition of an Instance	28
RELATED INVESTIGATIONS	30
Parallel Application of the Algorithm	30
Parallelism in Language Syntactic Definition	30
PROGRAM FOR THE NEXT INTERVAL	34
BIBLIOGRAPHY	35
DISTRIBUTION LIST	36

ILLUSTRATIONS

		Page
Table		
I	Reduction of Syntactic Classes	32
Figure		
1a	Algorithm for Essential Order Detection	4
1b	Symbology for Algorithm	5
2	Graph for Conditionals	18
3	$_{o}S$ Graph for Analysis of Recursive Subroutines	18
4	Serial Input-Output Calls	19

INTRODUCTION

The object of this study is to detect instances of parallelism implicit in programs written in compiler programming languages. The method chosen is to recognize the essential partial ordering between program parts since only parts which are not essentially ordered can be executed concurrently.

In this report an algorithm for formal analysis of programs is presented and proved which yields all instances of implicit parallelism between program parts based on input-output set intersections. Any initially known essential ordering is used. The number of input output set comparisons is minimal. At most, two consecutive iterations of a loop are necessary to determine the essential order for all iterations of the loop. Only one iteration need be analyzed for intra-loop essential order. The inter-loop essential order is determined by using both iterations. Sufficiency of this analysis is shown through application to language independent formal translator structures, including subroutines, loops, conditionals, recursive subroutines, and serial input or output calls.

Special features of particular programming languages affecting implicit recognition of essential ordering include loops, unconditional transfers, conditional statements, and parallel evaluation of a sequence of conditionals. The loop statements yield potentially the greatest opportunity for parallelism. Conditions for replacing iterative control by a number of parallel paths of control are given. Unconditional transfers may create loops or cross boundaries of scopes of variables. Data dependent conditions are a principal cause of essential ordering. The duration of definition of an instance of a variable provides essential information for efficient memory allocation.

Alternative algorithms which can be executed in parallel to achieve results comparable to the main algorithm are suggested. A method is indicated for reducing the complexity of syntactic definition in context-free languages by establishing classes of productions which can be recognized in parallel.

Most present programming languages presume that programs are to be written as a sequence of instructions. This permissible sequence, while it contains the essential ordering (i.e., it computes each value before that value is used), also contains much extraneous order (i.e., it orders computations for which the order is completely immaterial). In the previous report[1] we gave an algorithm which detects the essential ordering given a permissible ordering. In this report we extend the algorithm to permit detection of essential ordering given a consistent combination of essential and permissible ordering.

---

[1] Superscript numbers refer to references in the Bibliography.

In order to describe the algorithm, the meaning of some terms should be given:

A <u>process</u> is a transformation which generates a finite set of outputs from a finite set of inputs.

An <u>output</u> is the information possibly written into a register during a process.

An <u>input</u> is the information contained in a register at time of access by a process.

A <u>program</u> is a finite set of processes which can be partially ordered by their input-output set intersections.

A process <u>execution</u> is the application of the process transformation to its input set to produce its output set.

An ordered pair of inputs and outputs will be identified with each process. For process P_i, this pair will be represented as $\langle I_i, O_i \rangle$. It will be assumed that all outputs are unique, that is, every time a register is written into, a new name is created. This is done to keep separate the recognition of implicit parallelism in names from the potentially many-to-one mapping of names into registers.

The following relations between process pairs are used in this report:

${}_oT_k^i$ P_i must precede P_k is given.

${}_oS_k^i$ P_i may precede P_k is given.

T_k^i P_i must precede P_k.

S_k^i P_i must directly precede P_k.

If neither T_k^i nor T_i^k, then processes P_i and P_k can be executed in either order or concurrently. ${}_oT$, ${}_oS$, T, and S are, respectively, the sets of all true ${}_oT_k^i$, ${}_oS_k^i$, T_k^i, and S_k^i. The algorithm uses the given ${}_oT$ and ${}_oS$ to produce T and S.

A graphical representation of the effect of applying the algorithm was presented in the first report.[1] This representation is still appropriate for the revised algorithm with the following substitution. Each relation P_i R P_k was labeled as a directed R-arc from process i to process k. Replace each R by S. To distinguish between these S arcs and the S arcs used in the graphical illustrations in the first report, note that the reference process pair being analyzed partitions S into three disjoint sets: process pairs already analyzed, the reference process pair, and process pairs to be analyzed. Consequently, there is no need for separate symbols. The graphs used as illustrations in this report consider only ${}_oS$ and ${}_oT$.

ALGORITHM FOR DETECTING ESSENTIAL SERIAL ORDER

This section describes the algorithm (Figures 1a and 1b) for detection of essential serial ordering of processes from their input and output sets.

From previous partial analysis or explicit indicators, essential order is sometimes known to exist between processes of a program. Consequently, the algorithm of the previous report[1] has been extended to include any initially known essential order among processes as parameters to the algorithm. Therefore, the corresponding input-output set comparisons are avoided during the algorithm. When no essential serial order is known initially, the algorithm is equivalent to the previous one * The extended algorithm together with formal definitions of its parameters and the relations among them are now given. A proof of the algorithm is also provided. The number of input-output set comparisons made by the algorithm is shown to be minimal. The reader who does not wish to engage in the details of the proof can obtain the essence of the algorithm and minimal comparison argument from the definitions and subsection introductions.

*When $_0T$ is empty, then any T arc identified in step 3, Figure 1a, will not be in S. This will be explained later.

Given: ${}_oS$, ${}_oT$, N, I_j $(1 \le j \le N)$, O_j $(1 \le j \le N)$ and ${}_oS_k^i$ implies $i<k$

Find: ${}_MS$, ${}_MT$

Method:

```
Step 1.   for k = 2, 3, ..., N do
Step 2.     for i = k-1, k-2, ..., 1 do
Step 3.       if T_k^i = false then
Step 4.         if S_k^i = true then
Step 5.           if O_i ∩ I_k ≠ ∅ then
Step 6.             T_k^i ← true
Step 7.             go to step 15
Step 8.           otherwise (O_i ∩ I_k = ∅)
Step 9            S_k^i ← false
Step 10.          for j = 1, 2, ..., i-1 do
Step 11.            S_k^j ← S_k^j ∨ S_i^j ∧ T̄_k^j
Step 12.          for j = k+1, k+2, ..., N do
Step 13.            S_j^i ← S_j^i ∨ S_j^k
Step 14.       otherwise (T_k^i = true)
Step 15.         for j = 1, 2, ..., i-1 do
Step 16.           S_k^j ← S_k^j ∧ T̄_i^j
Step 17.           T_k^j ← T_k^j ∨ T_i^j
```

Figure 1a. Algorithm for Essential Order Detection

PARAME.

${}_oS$: initially given permissible ordering relation

${}_oT$: initially known essential ordering relation

N : number of processes in program

I_j : input set for process j

O_j : output set for process j

${}_MS$: covering relation for the complete essential ordering

${}_MT$: the complete essential ordering relation

SYMBOLS

ϕ : the empty set

$\cap$: set intersection

$\leftarrow$, $\vee$, $\wedge$, $\bar{\ }$ are respectively the binary operators "replaced by", "logical inclusive or", "logical and", and "logical complement", given in increasing binding order

SUBSCRIPTS AND SUPERSCRIPTS

For any relation $\longrightarrow$ ${}^{t}_{m}R^{i}_{k}$

- t → indicates transitive closure of ${}_mR$
- i → indexes the predecessor process
- k → indexes the successor process
- m → indicates the iteration of the algorithm which produces R; m does not appear in the algorithm description

R^i_k = true if and only if ${}_oR^i_k$ and no assignment has been made into R^i_k, or the last assignment into R^i_k was true

Figure 1b. Symbology for Algorithm

DEFINITIONS

Definition 1. Process, P_j.

A process P_j is an ordered pair of sets $\langle I_j, O_j \rangle$. I_j is called the input set for P_j. O_j is called the output set for P_j.

Definition 2. Program, P.

By a program P is meant a finite set of processes $\{P_j\}$ $(j = 1, 2, \ldots, N)$, for which the intersection of the input and output sets can be used to define a strongly anti-symmetric relation. That is, P can be ordered so that for any $P_i, P_k \in P$, $O_i \cap I_k \neq \emptyset$ implies $i < k$.

Definition 3. R_k^i, the arc from P_i to P_k.

For any relation $R \subseteq (P \times P)$ we will write R_k^i if and only if $P_i, P_k \in P$ and R relates P_i to P_k in that order.

Definition 4. ${}^tR$, the transitive closure of R.

For any relation $R \subseteq (P \times P)$ we will write ${}^tR$ to mean that relation such that ${}^tR_k^i$ if and only if there is a sequence of R arcs: $R_{j_1}^i, R_{j_2}^{j_1}, \ldots, R_{j_n}^{j_{n-1}}, R_k^{j_n}$. Note that ${}^tR$ is always transitive.

Definition 5. T, the essential serial ordering.

The relation $T \subseteq (P \times P)$ is the essential serial ordering among the processes of P. This order is imposed by the input-output set relation. That is, for any $P_i, P_k \in P$, T_k^i if and only if there is a sequence $O_i \cap I_{j_1} \neq \emptyset$, $O_{j_1} \cap I_{j_2} \neq \emptyset$, $\ldots, O_{j_n} \cap I_k \neq \emptyset$. Thus T is the transitive closure of the input-output set relation. Then T is transitive and since the input-output set relation is strongly anti-symmetric, so is T.

Definition 6. S, the cover for the essential serial order.

S is the covering relation for T. That is, for any $P_i, P_k \in P$, S_k^i if and only if T_k^i and there is no $P_j \in P$ such that both T_j^i and T_k^j. Note that ${}^tS = T$.

Definition 7. ${}_oT$, the known essential serial order.

The relation ${}_oT$ is any subset of the relation T.

Definition 8. ${}_oS$, the given permissible order.

The relation ${}_oS$ is the given ordering of the processes in P supplied by the programmer. ${}_oS$ is any strongly anti-symmetric relation $\subseteq (P \times P)$ such that $T \subseteq {}_o^tS$ and ${}_oT \subseteq {}^t({}_oT \cap {}_oS)$.* Note that T satisfies the requirements for ${}_oS$.

Definition 9. ${}_mR$, the relation R after m iterations.

For any relation R, we will write ${}_mR$ to mean the value of that relation after the mth iteration of the "for i" loop (steps 3 to 17) in the algorithm.

Convention 1. N, k, i, and M.

Hereafter we will write N to mean the number of processes in P; k and i will mean respectively the values of k and i during the $(m + 1)^{st}$ iteration of the "for i" loop; because N is finite (Definition 2) and the only loops in the algorithm are at steps 1 and 2, the algorithm terminates in a finite number of steps and we will write M to mean the total number of iterations of the "for i" loop.

Definition 10. ${}_mC$, the compared process pair relation.

We will write ${}_mC$ $(1 \leq m \leq M)$ to mean that relation such that ${}_mC_h^g$ if and only if $P_g, P_h \in P$ and $i = g$ and $k = h$ for some iteration j $(1 \leq j \leq m)$ of the "for i" loop. Note that ${}_MC_h^g$ if and only if $P_g, P_h \in P$ and $g < h$.

---

* For any program P and any relations Q, $R \subseteq (P \times P)$, $Q \subseteq R$ if and only if for all P_i, $P_k \in P$, Q_k^i implies R_k^i.

PROOF OF ALGORITHM

The algorithm (Figures 1a and 1b) generates the relations ${}_MS$ and ${}_MT$ given the relations ${}_oS$ and ${}_oT$, the input sets I_i $(1 \le i \le N)$, and the output sets O_i $(1 \le i \le N)$. It will be shown that ${}_MS = S$ and ${}_MT = T$. The algorithm functions as follows.

The body of the "for i" loop (steps 3 to 17) is executed once for each arc from P_i to P_k such that $P_i, P_k \in P$ and $i < k$. These arcs are sufficient because ${}_oS$, ${}_oT$, S, and T are strongly anti-symmetric. The order of the arcs (steps 1 and 2) guarantees that all sequences of arcs connecting two processes will be determined before the single arc connecting the processes is considered. Therefore, all indirect T paths can be determined without comparing the input-output sets of the end processes. For each iteration of the "for i" loop, if the arc from P_i to P_k is not already in ${}_mT$ (step 3), then it must either be in S, or P_i and P_k can be executed concurrently. Therefore, if the arc is not in ${}_m^tS$ (step 4), then P_i and P_k can be executed concurrently. If the arc is in ${}_m^tS$ (step 4), then the input-output comparision must be made (step 5). If the intersection is non-empty, then the arc is in S and T, and is added to ${}_{m+1}T$ (step 6). If the intersection is empty, then P_i and P_k can be executed concurrently and the arc will be deleted from ${}_mS$ (step 9). To ensure that the arc from P_i to P_k is the only arc deleted from ${}_m^tS$, arcs are added to ${}_{m+1}^tS$ (steps 10 through 13). Steps 10 and 11 guarantee that there is a sequence of ${}_{m+1}S$ arcs connecting to P_k from all P_j where ${}_mS_i^j$, while steps 12 and 13 guarantee that there are ${}_{m+1}S$ arcs connecting P_i to all P_j where ${}_mS_j^k$. Whenever there is an ${}_{m+1}T$ arc from P_i to P_k (step 14 or step 6), then steps 15 through 17 are performed. Since S is a cover, step 16 is included to ensure that no sequence of ${}_{m+1}T$ arcs ending in

an arc from P_i to P_k is an arc in ${}_{m+1}S$. Similarly, since T is transitive, step 17 includes all sequences of ${}_{m+1}T$ arcs ending in an arc from P_i to P_k as arcs in ${}_{m+1}T$.

<u>Lemma 1.</u> For all $m(0 \le m \le M)$, ${}_mT \subseteq T$.

Proof. ${}_oT \subseteq T$ by Definition 7. Assume for any $m(0 \le m < M)$ that ${}_mT \subseteq T$. During the $(m+1)^{st}$ iteration of the "for i" loop (steps 3 to 17), arcs are added to ${}_{m+1}T$ only at steps 6 and 17. If ${}_{m+1}T^i_k$ is added at step 6, then $O_i \cap I_k \neq \phi$ (step 5) and by Definition 5, T^i_k. If the arc ${}_{m+1}T^j_k$ is added at step 17, then ${}_mT^j_i$ (step 17). By hypothesis ${}_mT^j_i$ implies T^j_i. T^i_k since either $O_i \cap I_k \neq \phi$ (step 5) or ${}_mT^i_k$ (step 14). But T^j_i and T^i_k imply T^j_k, since T is transitive (Definition 5). Therefore, all ${}_{m+1}T$ arcs added during the $(m+1)^{st}$ iteration are in T. Since by hypothesis, all other ${}_{m+1}T$ arcs are in T it follows that ${}_{m+1}T \subseteq T$. By induction on m, ${}_mT \subseteq T$ for all $m(0 \le m \le M)$.

<u>Lemma 2.</u> For all $m\ (0 \le m \le M)$, ${}_mT \subseteq {}^t({}_mT \cap {}_mS)$.

Proof. ${}_oT \subseteq {}^t({}_oT \cap {}_oS)$ by Definition 8. Assume for any $m(0 \le m < M)$ that ${}_mT \subseteq {}^t({}_mT \cap {}_mS)$. If not ${}_{m+1}T \subseteq {}^t({}_{m+1}T \cap {}_{m+1}S)$, then either some ${}_{m+1}T$ arc was added or some ${}_mS$ arc was deleted during the $(m+1)^{st}$ iteration of the "for i" loop. If ${}_{m+1}T^i_k$ was added at step 6, then ${}_{m+1}S^i_k$ (step 4) and thus ${}^t({}_{m+1}T \cap {}_{m+1}S)^i_k$. If ${}_mS^i_k$ is deleted at step 9, then ${}_m\overline{T}^i_k$ (step 3), and since no ${}_{m+1}T$ arcs are added during the $(m+1)^{st}$ iteration, ${}_mT \subseteq {}^t({}_mT \cap {}_mS)$ implies ${}_{m+1}T \subseteq {}^t({}_{m+1}T \cap {}_{m+1}S)$. If ${}_mS^j_k$ is deleted at step 16 or ${}_{m+1}T^j_k$ is added at step 17, then either ${}_{m+1}T^i_k$ (step 6) and ${}_{m+1}S^i_k$ (step 4) or ${}_mT^i_k$ (step 14). In either case ${}^t({}_{m+1}T \cap {}_{m+1}S)^i_k$. But if ${}_mS^j_k$ was deleted or ${}_{m+1}T^j_k$ added

(steps 16 and 17), then ${}_mT_i^j$, and thus by hypothesis ${}^t({}_mT \cap {}_mS)_i^j$. But $j < i < k$ and all ${}_mS$ arcs deleted during the $(m+1)^{st}$ iteration are of the form ${}_mS_h^g$ where $h = k$ (step 16), and no ${}_mT$ arcs are deleted. Therefore, ${}^t({}_{m+1}T \cap {}_{m+1}S)_i^j$. Then ${}^t({}_{m+1}T \cap {}_{m+1}S)_k^j$, since both ${}^t({}_{m+1}T \cap {}_{m+1}S)_i^j$ and ${}^t({}_{m+1}T \cap {}_{m+1}S)_k^i$. In all cases then, ${}_{m+1}T \subseteq {}^t({}_{m+1}T \cap {}_{m+1}S)$. By induction on m, ${}_mT \subseteq {}^t({}_mT \cap {}_mS)$ for all $m (0 \le m \le M)$.

<u>Lemma 3.</u> For all $m (0 \le m \le M)$, $T \subseteq {}_m^tS$.

Proof. If $m = 0$, then by Definition 8, $T \subseteq {}_o^tS$. The only ${}_mS$ arcs which are deleted during the $(m+1)^{st}$ iteration of the "for i" loop are at steps 9 and 16. If ${}_mS_k^i$ is deleted at step 9, then with the exception of ${}_mS_k^i$ itself all sequences of ${}_mS$ arcs beginning with ${}_mS_k^i$ are retained in ${}_{m+1}S$ (steps 12 and 13). All sequences of ${}_mS$ arcs ending with ${}_mS_k^i$ are retained in ${}_{m+1}S$ (steps 10 and 11), with the exception of ${}_mS_k^i$ and ${}_mS_k^j$ (step 11) where $1 \le j < i$ (step 10) and ${}_mT_k^j$ (step 11). The arc ${}_mS_k^i$ need not be retained since $O_i \cap I_k = \phi$ (step 8). That is, $\overline{T}_k^i$. By Lemma 2, ${}_mT_k^j$ (step 11) implies ${}^t({}_mT \cap {}_mS)_k^j$. But ${}_m\overline{T}_k^i$ (step 3) so that the sequence of $({}_mT \cap {}_mS)$ arcs from P_j to P_k cannot contain an arc from P_i to P_k. Thus the arc ${}_m^tS_k^j$ is retained when ${}_mS_k^i$ (step 9) is deleted. If ${}_mS_k^j$ $(1 \le j < i)$ is deleted at step 16, then ${}_mT_k^i$ (step 3) and ${}_mT_i^j$ (step 16). By Lemma 2, ${}^t({}_mT \cap {}_mS)_k^i$ and ${}^t({}_mT \cap {}_mS)_i^j$, which implies there is a sequence of ${}_mS$ arcs from P_j to P_k other than the single arc ${}_mS_k^j$. Therefore, deleting the arc ${}_mS_k^j$ does not delete the arc ${}_m^tS_k^j$. Then in all cases $T \subseteq {}_m^tS$ implies $T \subseteq {}_{m+1}^tS$. By induction on m, $T \subseteq {}_m^tS$ for all m $(0 \le m \le M)$.

Lemma 4. For all $m(0 \le m \le M)$, ${}_mS \cap {}_mC \subseteq {}_mT \cap {}_mC$.

Proof. If $m = 0$, then ${}_oC = \phi$ and we are finished. Assume for any $m(0 \le m < M)$ that ${}_mS \cap {}_mC \subseteq {}_mT \cap {}_mC$. Let i, k be respectively the values of i and k during the $(m+1)^{st}$ iteration of the "for i" loop. Then consider any $P_g, P_h \in P$ such that ${}_{m+1}C^g_h$ and ${}_{m+1}S^g_h$. ${}_{m+1}C^g_h$ (Definition 10) implies that either ${}_mC^g_h$, or $g = i$ and $h = k$. ${}_{m+1}S$ arcs are added only at steps 11 and 13, and during the $(m+1)^{st}$ iteration none of the ${}_{m+1}S$ arcs added are in ${}_{m+1}C$ (steps 10 and 12). If ${}_mC^g_h$, then ${}_{m+1}S^g_h$ implies ${}_mS^g_h$ and by hypothesis ${}_mT^g_h$, but no ${}_mT$ arcs are deleted, so ${}_{m+1}T^g_h$. Otherwise, $g = i$ and $h = k$, and by the above argument ${}_{m+1}S^i_k$ implies ${}_mS^i_k$. Then if ${}_mT^i_k$ (step 14), ${}_{m+1}T^i_k$. If ${}_m\overline{T}^i_k$ (step 3), then ${}_mS^i_k$ (step 4) and ${}_{m+1}S^i_k$ (step 9) require that step 6 and not step 9 be executed. But by step 6, ${}_{m+1}T^i_k$. Then in any case, any arc in both ${}_{m+1}S$ and ${}_{m+1}C$ is also in ${}_{m+1}T$. By induction on m, ${}_mS \cap {}_mC \subseteq {}_mT \cap {}_mC$ for all m $(0 \le m \le M)$.

Lemma 5. For all $m(0 \le m \le M)$ and for any $P_j, P_g, P_h \in P$, if ${}_mC^g_h$, ${}_mT^j_g$ and ${}_mT^g_h$, then both ${}_mT^j_h$ and ${}_m\overline{S}^j_h$.

Proof. If $m = 0$, then ${}_oC = \phi$ and we are finished.

Assume Lemma 5 to be true for any $m(0 \le m < M)$. Then consider any $P_j, P_g, P_h \in P$ such that ${}_{m+1}C^g_h$, ${}_{m+1}T^j_g$, and ${}_{m+1}T^g_h$. By Lemma 1, ${}_{m+1}T^g_h$ implies T^g_h, and since T is strongly anti-symmetric (Definition 3), $g < h$. Similarly, ${}_{m+1}T^j_g$ implies $j < g$. If ${}_mC^g_h$, then ${}_mC^j_g$, since $j < g < h$. Then ${}_mT^j_g$ and ${}_mT^g_h$, since none of the ${}_{m+1}T$ arcs added during the $(m+1)^{st}$ iteration (step 6 and 17) are $\in {}_mC$. By hypothesis ${}_mC^g_h$, ${}_mT^j_g$, and ${}_mT^g_h$ imply ${}_mT^j_h$ and ${}_m\overline{S}^j_h$. But no ${}_mT$ arcs are deleted, so ${}_{m+1}T^j_h$. If ${}_{m+1}S^j_h$, then since ${}_m\overline{S}^j_h$, ${}_{m+1}S^j_h$ must have been added during the

$(m+1)^{st}$ iteration (step 11 or step 13). But it could not have been added at step 11, since then ${}_m\overline{T}^j_h$ (step 11). Neither could ${}_{m+1}S^j_h$ have been added at step 13, since then $h > k$ (steps 13 and 14), but by ${}_mC^g_h$, $h \le k$. Otherwise, ${}_{m+1}C^g_h$ and ${}_m\overline{C}^g_h$, that is, $g = i$ and $h = k$. Then ${}_mT^i_k$ (step 14) or ${}_{m+1}T^i_k$ was added (step 6). In either case for any $j < i$ (step 15), ${}_mT^j_g$ (that is ${}_mT^j_i$) implies ${}_{m+1}\overline{S}^j_k$ (step 16) and ${}_{m+1}T^j_k$ (step 17). Then in all cases ${}_{m+1}C^g_h$, ${}_{m+1}T^j_g$, and ${}_{m+1}T^g_h$ imply ${}_{m+1}T^j_h$ and ${}_{m+1}\overline{S}^i_h$. By induction on m, for all $m(0 \le m \le M)$ and any $P_j, P_g, P_h \in P$, ${}_mC^g_h$, ${}_mT^i_g$, and ${}_mT^g_h$ imply ${}_mT^j_h$ and ${}_m\overline{S}^j_h$.

<u>Lemma 6.</u> ${}_MS \subseteq {}_MC$.

Proof. From the algorithm we see that all S arcs added during the execution of the algorithm are of the form ${}_MS^j_k$ (steps 10 and 11), where $1 \le j < i < k \le N$, or of the form ${}_MS^i_j$ (steps 12 and 13), where $1 \le i < k < j \le N$. Then since ${}_oS \subseteq {}_MC$ (Definition 8), all arcs of ${}_MS$ must be in ${}_MC$.

<u>Theorem 1.</u> ${}_MT = T$.

Proof. By Lemma 4, ${}_MS \cap {}_MC \subseteq {}_MT$, but since ${}_MS \subseteq {}_MC$ (Lemma 6), then ${}_MS \subseteq {}_MT$. Therefore, ${}^t_MS \subseteq {}^t_MT$, and since $T \subseteq {}^t_MS$ (Lemma 3), we have $T \subseteq {}^t_MT$. ${}_MT \subseteq T$ (Lemma 1) and $T \subseteq {}_MC$ (Definition 5), so ${}_MT \subseteq {}_MC$. Therefore, ${}_MT$ is transitive since ${}_MT \cap {}_MC$ is transitive (Lemma 5) and ${}_MT \cap {}_MC = {}_MT$. But if ${}_MT$ is transitive, then ${}^t_MT = {}_MT$, and since we already have $T \subseteq {}^t_MT$, $T \subseteq {}_MT$. Finally, by Lemma 1, ${}_MT \subseteq T$, so ${}_MT = T$.

<u>Theorem 2.</u> ${}_MS = S$.

Proof. ${}_MS \subseteq {}_MT$ since ${}_MS \cap {}_MC \subseteq {}_MT$ (Lemma 4) and ${}_MS \subseteq {}_MC$ (Lemma 6). But ${}_MS \subseteq {}_MT$ implies ${}^t_MS \subseteq {}^t_MT$, and since ${}_MT = T$ (Theorem 1), ${}^t_MS \subseteq {}^tT$. Then since

T is transitive (Definition 5), ${}^tT = T$, and therefore ${}_M^tS \subseteq T$. But $T \subseteq {}_M^tS$ (Lemma 3) so ${}_M^tS = T$. By Lemma 5, for any $P_j, P_g, P_h \in P$, if ${}_MC_h^g$, ${}_MT_g^j$, and ${}_MT_h^g$, then ${}_M\bar{S}_h^j$. But ${}_MT = T$ (Theorem 1) and $T \subseteq {}_MC$ (Definition 5), so for any $P_j, P_g, P_h \in P$, if T_g^j and T_h^g, then ${}_M\bar{S}_h^j$. We already have ${}_M^tS = T$. Therefore ${}_MS = S$ by Definition 6.

PROOF OF MINIMAL COMPARISONS

It will now be shown that no algortihm can produce S and T from ${}_oS$ and ${}_oT$ with fewer comparisons between input and output sets. This will be done by first showing that one comparison must be made for each arc that is in S and not in ${}_oT$, and that one comparison must be made for each ${}_o^tS$ arc which is not in T. It will then be shown that each input-output set comparison in the algorithm identifies a unique arc which is either in S and not in ${}_oT$, or is in ${}_o^tS$ and is not in T, and that no comparison is made more than once.

<u>Lemma 7.</u> For all $m(0 \le m \le M)$, ${}_mS \subseteq {}_o^tS$.

Proof. ${}_oS \subseteq {}_o^tS$, by the definition of transitive closure. Assume for any $m(0 \le m < M)$ that ${}_mS \subseteq {}_o^tS$. During the $(m+1)^{st}$ iteration of the "for i" loop, arcs are added to ${}_{m+1}S$ only at steps 11 and 13. If ${}_{m+1}S_k^j$ is added at step 11, then ${}_mS_i^j$ (step 11) and ${}_mS_k^i$ (step 4). Therefore, by hypothesis ${}_o^tS_i^j$ and ${}_o^tS_k^i$. Then by the definition of transitive closure, ${}_o^tS_k^j$. Similarly, if ${}_{m+1}S_j^i$ is added at step 13, then ${}_mS_k^i$ (step 4) and ${}_mS_j^k$ (step 13), so that by hypothesis ${}_o^tS_k^i$ and ${}_o^tS_j^k$, and therefore ${}_o^tS_j^i$. Then ${}_{m+1}S \subseteq {}_o^tS$, since all ${}_{m+1}S$ arcs added during the $(m+1)^{st}$ iteration are in ${}_o^tS$ and by hypothesis all ${}_mS$ arcs are in ${}_o^tS$. By induction on m, ${}_mS \subseteq {}_o^tS$ for all $m(0 \le m \le M)$.

Theorem 3. The number of input-output set comparisons is minimal.

Proof. Let $P_i, P_k \in P$ such that ${}_o^tS_k^i$. For an algorithm to establish whether or not S_k^i, it must be at least able to determine whether T_k^i, since $S \subseteq T$. Unless it is given that T_k^i (that is, unless ${}_oT_k^i$), it must be shown that either $O_i \cap I_k \neq \phi$ or that there is a $P_j \in P$ such that both T_j^i and T_k^j (Definition 5). If S_k^i, then there is no $P_j \in P$ such that both T_j^i and T_k^j (Definition 6). Thus, the comparison $O_i \cap I_k$ must be made. If $\overline{T}_k^i$, then since T is transitive there can be no $P_j \in P$ such that both T_j^i and T_k^j. Thus, the comparison $O_i \cap I_k$ must be made. That is, one input-output set comparison must be made for each arc that is in S and not in ${}_oT$, and one comparison must be made for each ${}_o^tS$ arc which is not in T.

If during the $(m+1)^{st}$ iteration of the "for i" loop ($1 \leq m \leq M$), the comparison $O_i \cap I_k$ is made (step 5), then ${}_mS_k^i$ (step 4). But then ${}_o^tS_k^i$, since ${}_mS \subseteq {}_o^tS$ (Lemma 7). Also, $\overline{T}_k^i$ (step 3), and therefore ${}_o\overline{T}_k^i$, since the algorithm does not delete any arcs from T. If it happens that $O_i \cap I_k \neq \phi$, then the arc ${}_mS_k^i$ is not deleted during the $(m+1)^{st}$ iteration. But ${}_{m+1}S_k^i$ (steps 9 and 16) can not be deleted during any subsequent iteration of the "for i" loop. That is, ${}_MS_k^i$, and therefore S_k^i, since ${}_MS = S$ (Theorem 2). If $O_i \cap I_k = \phi$, then ${}_m\overline{T}_k^i$ (step 3) and step 6 is not executed during the $(m+1)^{st}$ iteration, so ${}_{m+1}\overline{T}_k^i$. But ${}_MT_k^i$ (steps 6 and 17) can not be added during any subsequent iteration of the "for i" loop. Thus, ${}_M\overline{T}_k^i$, and therefore $\overline{T}_k^i$, since ${}_MT = T$ (Theorem 1). That is, each input-output comparison $O_i \cap I_k$ in the algorithm identifies a unique arc (from P_i to P_k) which is either in S and not in ${}_oT$, or is in ${}_o^tS$ and is not in T. Finally, none of these comparisons is made more than once

since only the sets O_i and I_k are compared during the $(m+1)^{st}$ iteration (step 5), and no two iterations of the "for i" loop have the same value for the pair i, k (steps 1 and 2).

Therefore, every input-output comparison made by the algorithm is necessary.

ANALYSIS OF FORMAL PROGRAM STRUCTURES

The ability to apply the algorithm for detection of essential serial order with a nonempty $_oT$ allows more freedom in the use of the analysis. This facility is investigated relative to certain formal program structures and the advantages which are relevant to subroutines, conditionals and serial input-output calls are explained. With the explicit exclusion of arrays, it is shown that loops and recursive subroutines can be completely analyzed with only two instances of each process. Arrays will be considered in the next report.

SUBROUTINES

The advantages of the non-empty $_oT$ arise in the analysis of program structures such as subroutines. A subroutine (whether open or closed) need not be analyzed for each call, but may be analyzed only once and the results of that analysis used at each call on the subroutine. This is accomplished by first analyzing the subroutine and then using the resulting S and T relating the intra-subroutine processes as $_oS$ and $_oT$, respectively, for each program call on the subroutine. The program analysis will then identify all instances of parallelism without duplicating any comparison of the intra-subroutine input-output sets at the various subroutine calls.

An alternative method for handling subroutines reduces the number of processes used in the analysis and, therefore, the size of S and T. In this scheme, the subroutine is analyzed once separately from the program. Then rather than inserting the subroutine analysis results into the program at each call, the program is analyzed with each subroutine call serving as a single process. In this scheme, parallelism will not be found between processes where one of the processes is external to the subroutine but cannot be executed in parallel with the entire subroutine, and where the other process is interior to the subroutine. The above methods are not applicable to recursive subroutines, since the substitution process is nonterminating

LOOPS

The algorithm as described can be used to analyze a loop by stretching it out into a sequence of iterations. This analysis, however, cannot be performed until run time if the number of iterations is data dependent. Even if the number of iterations can be determined at compile time, the number of processes produced by flattening out the loop may exceed the handling capabilities.

A method for loop handling which takes advantage of the similarities between successive iterations of a loop and still recognizes those instances of parallelism determined by input-output set relations is now developed. Initially we will assume that the programs under consideration either do not contain arrays or that each array is treated as a single variable. This restriction guarantees that

input-output sets are not a function of the iteration. That is, for any iteration of a loop, if an instance of a variable appears in the input (or output) set for some process, then for any other iteration of the loop, another instance of that variable will appear in the input (or output) set of the corresponding process.

Since each iteration of a loop has the same processes in the same given order and with the same input-output names, analysis of any iteration of a loop will identify the intra-iteration parallelism for all iterations of the loop. The array handling technique mentioned above guarantees that analysis of any two consecutive iterations of a loop will identify all inter-iteration parallelism, since direct essential ordering of processes can exist only between processes in the same or consecutive iterations. Therefore, loops can be handled by analyzing only two consecutive iterations of each loop.

CONDITIONALS

There are several run-time philosophies which may be used in conjunction with conditionals. One approach permits both branches of the conditionals and the condition itself to be executed concurrently. When evaluation of the condition is complete, one of the branches will then be inhibited. This method reduces the duration of the program at the expense of performing some computation whose outputs will not be used.

An alternative approach will, however, be taken here. The goal will be to initiate each process as soon as possible without executing processes unnecessarily. This may be done by evaluating the condition before either branch of the conditional is initiated and then executing only the single necessary branch. This approach does not prohibit processes common to both branches of the conditional from being executed concurrently with the evaluation of the condition.

Conditionals can be analyzed as any other processes, except that the given ${}_oT$ will be nonempty. For example, let process P_1 be the condition, P_2 and P_3 be local to one branch of the conditional, P_4 and P_5 be local to the other branch of the conditional, and P_6 be common to both branches. Then ${}_oS$ will be the given order of the processes as shown in Figure 2. ${}_oT$, however, will have four arcs, one arc each to indicate the serial ordering between the condition evaluation and the processes local to the conditional branches. ${}_oS_3^1$ and ${}_oS_5^1$ are included to guarantee that ${}_oT \subseteq {}^t({}_oS \cap {}_oT)$.*

---

*The need for the requirement ${}_oT \subseteq {}^t({}_oS \cap {}_oT)$, introduced in Definition 8, is illustrated by this example. If ${}_oS_3^1$ were not included and $O_2 \cap I_3 = \emptyset$, then the algorithm would not generate ${}_MS_3^1$, since ${}_oT_3^1$, and consequently $S \neq {}_MS$, since ${}_M\bar{S}_3^1$, even though S_3^1.

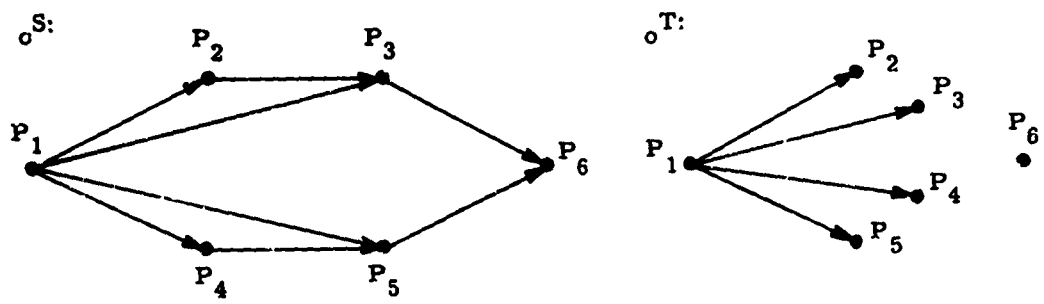

Figure 2. Graph For Conditionals

RECURSIVE SUBROUTINES

The analysis of recursive subroutines may be handled similar to loops. Since each level of call on a recursive subroutine has the same processes in the same given order and with the same input-output names, analysis of any level of recursion will identify the intra-level parallelism for all levels of recursion. Treating arrays as single variables guarantees that analysis of any two consecutive levels of recursion will identify all parallelism between processes in the same level, between processes in consecutive levels, and between processes in nonconsecutive levels where that parallelism also exists between consecutive levels. Therefore, recursive subroutines can be handled by analyzing two consecutive levels of recursion in the subroutine.

An example of $_oS$ for a recursive subroutine is shown in Figure 3. Each level of the given subroutine consists of a condition P_a, followed by alternative processes P_b and P_c and, in either case, terminating with P_d. The process P_b is a recursive call on the subroutine. Figure 3a shows $_oS$ for intra-level analysis, while Figure 3b shows $_oS$ for inter-level analysis. The primed and nonprimed P's represent processes in two consecutive levels of the recursion. Note that the process number for P_a must be less than that of P'_a, while the number of P_d must be greater than that of P'_d.

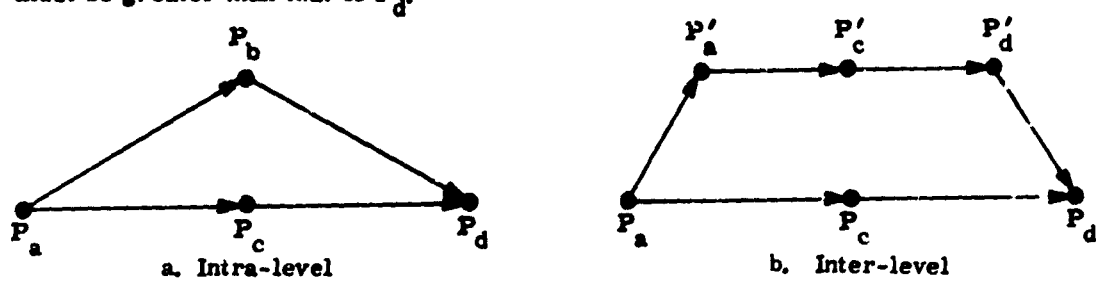

Figure 3. $_oS$ Graph for Analysis of Recursive Subroutines

SERIAL INPUT-OUTPUT CALLS

Many oneway input and output devices, such as card readers or line printers, are read or written serially. To ensure that the information received from (or transmitted to) these devices is interpreted (or displayed) as intended, the given order of reference must be maintained For example, if lines were sent to a line printer in any order other than that given by the programmer, the intended format would be disrupted. Thus, for each serial device a $_oT$ arc will connect those pairs of processes which include consecutive references to that device. Let P_1, P_2, P_3, P_4, P_5, and P_6, in that order, comprise a program, and let P_2, P_3 and P5 include reference to a particular serial input or output device. The $_oS$ and $_oT$ for this program will then appear as shown in Figure 4.

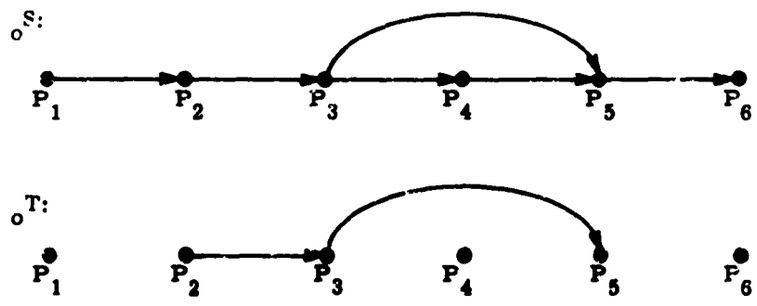

Figure 4. Serial Input-Output Calls

PROGRAMMING LANGUAGE FEATURES EFFECTING PARALLELISM

The language-independent approach to recognizing parallelism through study of formal translator structures permits identifying general aspects of essential ordering without getting involved with features of particular languages.

Specific language-dependent features are also important since the intended applications are programs written in actual languages. By considering the specific differences in actual languages, a comparative basis can be established for recommending that particular features be used for parallel recognition, that features be used as essential order indications, and, indeed, that languages contain particular features to aid in the recognition of parallelism.

Some specific features of languages will now be investigated. Loop statements yield the largest potential for parallelism, since each set of embedded loops multiplies the number of opportunities for parallel execution. The FORTRAN DO-, ALGOL FOR-, and COBOL PERFORM- statements are analyzed. Unconditional transfers cause problems in the recognition of loops and in crossing the scope boundary of variables. Conditionals which are data-dependent pose the principal impediment to parallelism. Evaluating groups of conditionals in parallel, rather than scattering them through a program, minimizes the number of separately ordered parts of a program. Duration of definition of an instance of a variable is important to the mapping of instances into memory on a noninterfering basis. A beginning on this analysis is reported.

LOOPS

Loops play a dominant role in programs written in present programming languages. They permit programmers to iteratively express repetitive processes with economy of program. The iterative nature of loop control is adequate for sequential execution. However, the iterative form impedes parallel setup of the control for loop bodies. Gosden[8] has concentrated on explicit loop constructs as the most promising sources for parallel activity. He proposes that a large fraction of all loops are parallel, both in the control and the loop bodies, and recommends explicitly adding the ability to specify loops as either parallel or iterative in the programming language.

We will now consider the control of loops and parallel establishment of multiple paths of control even when the control mechanism is iteratively expressed in the programming language.

Some opportunities to establish in parallel more than one execution of a loop body are determined by the algorithm. The algorithm requires for concurrent execution that not only must the control variable be independent of its predecessor control variable, but also independent of its predecessor loop body. The loop control statements in FORTRAN, ALGOL, and COBOL will be compared to see what other opportunities exist for establishing concurrent paths of control for loop bodies.

At compile time, if the number of executions of a loop is recognizable as an integer, then parallel paths may be established. If the number of executions is recognizable at execute time upon encountering the loop entry, then parallel path controls may be established at that time to initiate that number of loops.

Conditional statements within a loop body that can lead outside the loop with no intent to return are possible in ALGOL and FORTRAN. COBOL and ALGOL have explicit forms of loop control including condition evaluation to determine loop completion. Evaluating such a condition generally depends on loop-created data (otherwise an explicit form for indicating the number of iterations of the loop would have been used). Consequently, there is an essential order between cycles of the loop when a condition determines the exit. In some cases it may be possible to reformulate the loop to separate all condition evaluations from loop body execution.

FORTRAN and COBOL program units are characterized by static storage requirements determinable at compile time. ALGOL program units, on the other hand, assume dynamic storage requirements. The effect of this difference on loop control is to allow significantly more ways to defer to execute time the decision on number of loop executions in ALGOL, and to make loop executions essentially ordered.

Further interpretations and restrictions on these general ideas are developed in the following three descriptions of the particular loop statements in each language.

FORTRAN DO Statement*

A DO statement is of the form

DO n i = m_1, m_2, m_3

where: n is the statement label of an executable statement occurring as the terminal statement of the associated DO. The statement must follow the DO and be in the same program unit. The terminal statement may not be a GO TO of any form, arithmetic IF, RETURN, STOP, PAUSE, DO statement, nor a logical IF containing any of these forms. In effect, this allows only the DO loop control to follow execution of the terminal statement.

i is an integer variable name of the control variable,

m_1 is the initial parameter,

m_2 is the termination parameter,

m_3 is the incrementation parameter if present, otherwise +1 is implied.

*This description of the FORTRAN DO statement is adapted from reference 3.

Each m_1, m_2, and m_3 is either an integer constant or integer variable reference. At time of execution each must be positive and $m_1 < m_2$. The range of the DO is the set of executable statements following the DO statement through the terminal statement. Procedure actions required within the range are assumed to be temporarily within the range.

Redefining (by assigning of a new value to) any of i, m_1, m_2, m_3 is prohibited during the execution of the range of the DO. This means that the maximum number of executions is always known before first executing the range.

The DO statement execution sequence is 1) $i = m_1$; 2) execute range, if the terminal statement is reached; 3) $i = i + m_3$, if $i < m_2$ GO TO 2); 4) exit with DO satisfied.

Exiting from the range of a DO may occur by execution of a GO TO statement or an arithmetic IF, that is, exiting may occur without satisfying the DO.

A GO TO or arithmetic IF statement may not cause control to pass into the range of a DO from outside its range, except as described below for the extended range.

All values of the control variable can be assigned at compile time if the following two conditions hold: 1) m_1, m_2, and m_3 are integer constants, 2) there occurs no exit from the range of the DO by execution of a GO TO statement or an arithmetic IF statement. If these conditions hold, it is possible to establish $k = 1$ plus the greatest integer in $(m_2 - m_1)/m_3$ parallel assignments.

If condition 1) is relaxed to permit integer variables for any of the m_1, m_2, or m_3's, then at compile time it is possible to add the above computation for k as a control process which can then establish that number of parallel control paths for executing the ranges

Nested DO statements are possible so long as the range of the contained DO is a subset of the containing DO. Execution order is inside out. A complete nested nest of DO statements occurs when the first occurring terminal statement of any DO statement follows the last occurring DO statement and the first occurring DO statement of the set is not in the range of any DO statement. For such a completely nested nest of DO statements, an extended range is permitted for the innermost of the DO statements, from which control may pass external to the next and return to the innermost. No recursive use of the extended range is permitted.

It is not necessary that the range of an embedded DO statement be parallel for the range of an outer DO to be parallel. A nest of DO statements may be totally parallel, if all DO statements in the nest are parallel. In this case the product $k_1 \times k_2 \times \ldots \times k_n$ paths of control may be established.

ALGOL FOR Statement*

The syntax of the ALGOL FOR statement elements (given in a modified Backus normal form) which are important for this loop discussion are as follows.

⟨for list element⟩ ::= ⟨arithmetic-expression⟩ | ⟨arithmetic-expression⟩ step ⟨arithmetic-expression⟩ until ⟨arithmetic-expression⟩ | ⟨arithmetic-expression⟩ while ⟨Boolean-expression⟩

⟨for list⟩ ::= ⟨for list element⟩ | ⟨for list⟩, ⟨for list element⟩

⟨for clause⟩ ::= for ⟨variable⟩ := ⟨for list⟩ do

⟨for statement⟩ ::= ⟨for clause⟩⟨statement⟩ | ⟨label⟩ : ⟨for statement⟩

A for clause causes the statement which it precedes (the for loop body) to be repeatedly executed 0 or more times. In addition, it performs a sequence of assignments to the control variable from the for list.

The sequential execution expected is the following: 1) initialize the control variable by assignment from the value of the first for list element, 2) test for an invalid assignment; if it is invalid, go to the successor statement of the for statement, 3) execute the statement (exit if a go to leading outside the statement is encountered), 4) perform the next assignment from the next for list element in the order written to the control variable doing any necessary evaluation of arithmetic expressions, using the current values of primaries, and then go to (2) again.

In order to establish parallel paths of control for all executions of the loop body, the number must be known before any are executed. For this number to be known, there must not be any condition which is dependent upon loop-created data that can change this number. Consequently, for lists made up from for list elements of the AE or AE step AE until AE types (category1) are potentially unordered. Each for list element of the AE while BE type (category 2) imposes an essentially ordered sequence of loop body executions. A for list may consist of an alternating sequence of for lists from categories 1 and 2, in which case a similar sequence of potentially unordered and essentially ordered executions of the loop body exist. Any data-dependent conditional in the for which can cause exit from the loop body imposes essential order. Hereafter, we assume no such conditional and, thus, we consider only for list elements of category 1.

If no assignment is made into the control variable by any statement in the loop body, then all its values are obtained from the for list. ALGOL permits assignment to the control variable or to primaries in the arithmetic expressions of the step AE until AE parts of a for list element to be made in the loop body. If such assignments

*This is a partial syntax from reference 4 adapted by leaving undefined some non-terminal syntactic elements such as "⟨arithmetic-expression⟩".

are unconditionally made, and if they are a function of only values existing prior to the for statement, or of the prior control variable of these primaries, then the loop control may be separately analyzed from the rest of the loop body.

An apparently iterative sequence for establishing the values of the control may be replaced by parallel enumeration through recognition at compile time of the availability of the values of all primaries necessary for determining all for list elements. Should the values of all primaries be unsigned numbers, then the paths can actually be established at compile time. If any of these primaries is a variable and all such variable primaries have assignments into them restricted as stated, then the number of paths of control can be determined prior to first execution of the noncontrol portion of the loop body.

The control variable becomes undefined if exit results from exhaustion of the for list. The last value of the control variable is preserved if exit from the for statement occurs because of a go to in the loop body.

Side effects of a procedure call can cause assignments outside its body or exits other than the return to point of call. Such procedure calls occur in the for body or in the for list. Either of these can prevent or make indeterminate at compile time the establishment of parallel execution of the for body. Huxtable[5] has classified procedures as follows: normal - having no side effects, conditional sneaks - side effects are conditional on context, and unconditional sneaks. The conditions for recognizing normal procedures are as follows: no OWN variables, nonlocal assignments, abnormal exits, nor use of any switch; internal procedure calls limited to normal procedures; parameters exclude label and switch; and no explicit assignment to parameters called by name. Conditional sneaks are the same as normal except that explicit assignment to parameters called by name is permitted. All other procedures are assumed to be unconditional sneaks. He describes a technique for classifying procedures which involves discovering the total of all possible run-time procedure call structures of the program. Although further analysis might show that unconditional sneaks would not require essential ordering, the effort would likely be greater than the benefit gained.

COBOL PERFORM Statement[6]

The PERFORM statement is used to depart from the normal sequence of execution in order to execute one or more procedures either a specified number of times or until a specified condition is satisfied and then return control to the normal sequence - the statement following the PERFORM.

The four general formats are as follows:

1) PERFORM procedure-name-1 [THRU procedure-name-2]

2) PERFORM procedure-name-1 [THRU procedure-name-2] $\left\{\begin{matrix}\text{identifier-1}\\ \text{integer-1}\end{matrix}\right\}$ TIMES

3) PERFORM procedure-name-1 [THRU procedure-name-2] UNTIL condition-1

4) PERFORM procedure-name-1 [THRU procedure-name-2]

VARYING $\left\{\begin{matrix}\text{index-name-1}\\ \text{identifier-1}\end{matrix}\right\}$ FROM $\left\{\begin{matrix}\text{index-name-2}\\ \text{literal-2}\\ \text{identifier-2}\end{matrix}\right\}$ BY $\left\{\begin{matrix}\text{literal-3}\\ \text{identifier-3}\end{matrix}\right\}$ UNTIL condition-1

$\left[\begin{matrix}\text{AFTER} \left\{\begin{matrix}\text{index-name-4}\\ \text{identifier-4}\end{matrix}\right\} \text{FROM} \left\{\begin{matrix}\text{index-name-5}\\ \text{literal-5}\\ \text{identifier-5}\end{matrix}\right\} \text{BY} \left\{\begin{matrix}\text{literal-6}\\ \text{identifier-6}\end{matrix}\right\} \text{UNTIL condition-2}\\ [\text{AFTER} \ldots]\end{matrix}\right]$

Each procedure-name is the name of a section or paragraph in the Procedure Division. Each identifier represents a numeric elementary item described in the Data Division. In formats 2 and 4 with the AFTER option, each identifier represents a numeric item with no positions to the right of the assumed decimal point. Each literal represents a numeric literal.

There is no necessary relationship between procedure-name-1 and procedure-name-2, except that a consecutive sequence of operations is to be executed in every case beginning at procedure-name-1 and ending with procedure-name-2. In particular, GO TO and PERFORM statements may occur in the sequence. If there are two or more direct paths to the return point, then procedure-name-2 may be the name of a paragraph consisting of the EXIT statement, to which all of these paths must lead.

Format 1 corresponds to a call of a procedure without actual parameters. In format 2, the procedures are performed the number of times specified and, therefore, parallel paths of control may be unconditionally established. At PERFORM execution, the value of identifier-1 or integer-1 must not be negative. If the value is zero, control passes immediately to the statement following the PERFORM statement. Once initiated, any reference to identifier-1 has no affect on varying the number of times the procedures are executed. If given as integer-1, the control may be set up in parallel at compile time as long as the procedures in separate iterations are parallel. If given as identifier-1, the number of control setups may be determined from the value of identifier-1 on encountering the PERFORM at execute time.

The UNTIL condition parts of formats 3 and 4 preclude parallel execution, except in those cases where the conditions are fully evaluatable at compile time, or the ordered set of results of condition evaluations are determinable prior to executing the procedures. To achieve the equivalent of the ALGOL construct until AE, the condition would compare the index name (or identifier with the value of the desired limit).

Format 4 permits setting up one, two, or three control variables, testing their corresponding conditions, and if all are false, executing the procedures. After this, the last data-name is altered by the appropriate amount and the corresponding condition retested. When a condition is true, other than the first, that data-name is reinitialized, and the next preceding data-name is augmented and tested. When condition-1 is true, the PERFORM is finished.

In addition to the restriction on parallelism caused by the UNTIL condition parts, control variables other than the first are reinitialized to the FROM value during the PERFORM. Their values may be altered by the procedures from the values when the PERFORM was encountered; likewise, the index-names and identifiers occurring in the BY part of any of the control variables and any variables occurring as part of the conditions may be altered. Any such alteration will prevent parallel execution unless either numeric literals (or identifiers whose values are constrained similarly to the ALGOL primaries used in AE) are used for these alterations.

UNCONDITIONAL TRANSFERS

An unconditional transfer of control to another part of the program causes the following problems: 1) possible creation of loops and 2) crossing a boundary of scope of variables.

Potential creation of loops is detectable. An algorithm for detecting loops given the process connection matrix has been given by Marimont.[7] All program loops created by unconditional transfers include a backward jump. Not all backward jumps indicate loops, since the order of sequential programs can be scrambled using unconditional transfers. At compile time, the possible paths of control must be indicated. If paths are mutually exclusive, not only should the mechanism for enabling one path be provided, but also the outputs from paths not taken should be made invalid and the particular need for inputs required by such paths should be released.

During the execution of a serial program, crossing a boundary into the scope of a variable serves to reserve space for, but assign an undefined value to, any locally named (non-OWN) variable until some value assignment has been made to it. In ALGOL, in particular, a variable globally named the same as a locally named variable is inaccessible in the local block. Crossing a boundary out of the scope of a defined variable as a result of an unconditional transfer (or otherwise) should result in that local variable b[illegible] undefined and the storage allocated to it being released.

In parallel program execution, on the other hand, several instances of variables in different scopes having the same name may be accessed concurrently. Consequently, separate internal registers are required for all valid instances. Also, exiting a scope of a variable is not sufficient for making invalid or undefining the variable since some other process may still require it. Consequently, either a variable must remain defined until all segments of programs are executed which require it; or separate copies of the variable should be created for each use, in which case use becomes synonymouns with release of the storage for the variable.

CONDITIONALS

Language constructs for conditional establishment of paths of control based upon the result of a single condition are used in most programming languages. The form is to first evaluate the condition and then set up a path of control for the single path corresponding to the result. N-way branching constructs can be reduced to selection of one of N alternatives also based upon a single condition.

Conditional establishment of a single path of control presents a problem in deciding which instance of a register is referenced by a process following the condition when several instances could be meant, depending on the actual path executed. Alternatives for solution of this problem are based on the duration of definition of an instance.

SEQUENCE OF CONDITIONALS

Languages that group conditionals for evaluation with action selected as the first to be true (decision tables), or languages that have a list of condition - action pairs of which the first condition to be true selects the action (LISP) - provide opportunity for executing the conditionals in parallel. When logical relations among a group of conditions are evaluated to select a particular action process, the individual conditions may be evaluated in parallel as long as 1) evaluation of any conditional does not modify the result of some other conditional in the same group, and 2) all operands required for evaluation are defined.

The first qualification may be met by having temporary storage locations into which all instances of store operations associated with conditional evaluation are made, with stacking or tagging to indicate the creating condition. A conflict may occur because of name reuse in store operations temporary to or incidental to the condition evaluations. This conflict may be resolved by reference to a stack or tag associated with the name occurring in the nearest condition not following the condition being evaluated. By this means proper conditional evaluation will take place. The values of named variables that are later required and are created during evaluation of the satisfied condition may be permanently stored prior to continuing.

The second qualification requires defined (valid) operands for conditional evaluation. To permit recognition of these we must know which instance of each name is appropriate. For some conditionals, no instance of an operand need be appropriate. For example, assume that creation occurs as an action following some conditional evaluted earlier in the given sequence. That prior conditional must have become true and that action taken before the current conditional can be true. Therefore, in this case the conditional is inconsequential. Several creation points may be appropriate. With sequential evaluation, a stack can be used to show the order. With parallel evaluation, the completion order may be arbitrary, so an indication of the creating process is required to preserve the sequence as originally given. Parallel evaluation of conditionals appears generally to result in unnecessary work for the sake of finding the desired single action more rapidly.

When sufficient conditionals are evaluated to uniquely select the following action process, any other conditional evaluation can be suspended (and the temporaries there created can be released).

The single action selection is appropriate to sequential languages. If this is the case, the sequential language has implied false conditions as inputs to all but the first occurring condition The opportunity to sequentially test a particular conditional only occurs when all prior conditionals are false. Parallel evaluation of conditionals in such a case should select the first occurring true conditional for action. There are applications where multiple actions may be appropriate and therefore parallel action paths may be executed. The algorithm will identify these.

DURATION OF DEFINITION OF AN INSTANCE

Creation of instances of registers suggests that such registers are defined for all time after creation. In practical application, these instances have a last use as process input. At completion of this last use, the instance is of no further use and may be "unnamed", which serves to make it undefined thereafter. The interval between naming at creation and unnaming after last use is the duration of definition. This study has not been primarily concerned with questions regarding duration of definition, since such questions are more properly related to the allocation or mapping of names into memory without conflict. In order to exploit the duration of necessary definition among variables, a many-to-one mapping of names into a memory location on a non-overlapping duration basis is required.

The algorithm for determining essential process ordering uses less information than is required for determining the last use of a name. For example, the T relation between multi-input, multi-output processes eliminates a number of process comparisons, any of which may include the last use of any particular variable. Also, determination of any one name in the intersection of output and input sets is sufficient to establish essential order between two processes without completing all possible name comparisons in these sets. Determining the last use of a name requires checking all input sets of processes that are T successors of the creating process for occurrence of the particular created name. Some reduction in the amount of checking can be achieved in those cases where the language provides a limit on the scope of a variable and the durations are extended to this limit even if last use is earlier Other reduction may occur when only one of several separate instances may be referenced by an execution, in which case the originally formulated program must have included conditionals. The naming could be the same for all such mutually exclusive instances.

The recognition program for last use in present programming languages is impeded by the implicit reuse of names as outputs of processes with actual independent meaning Uniquely renaming these names having multiple meanings as proposed achieves separate registers for each so that no name has more than one meaning. The expense of doing this is that no register becomes undefined and thus no register can be reused in a program.

A comparison in the programmer-given name space of $O_i \cap O_k \neq \phi$ is an indication of multiple use of names. Each such name in the original order of processes determines a partition across the set of processes that use that name as input If a last use of a variable occurs in a particular statement, the use should be early in the statement evaluation so that the location can be freed, or so that the variable can be reassigned by a parallel path. Other variables having later use may be postponed in the particular statement evaluation. A suggested algorithm may be to minimize the duration in storage for any variable, since the duration is loosely related to the freedom to parallel process.

Duration of instances will be considered in more detail in a future report, where the problem of identifying the instances from names will be treated.

RELATED INVESTIGATIONS

The algorithm is sequentially formulated. Application of the algorithm to itself is suggested for analysis done on a highly parallel machine. Several alternative ways to view this algorithm are suggested.

Application of the algorithm to the syntactic definition of context-free languages to classify productions of the language that might be done in parallel is described. A method of determining the "essential complexity" of a programming language is suggested.

PARALLEL APPLICATION OF THE ALGORITHM

An alternative way of applying the algorithm is to analyze at each step in parallel all previously unanalyzed S relations then existing. For each, the result may be S, in which case T is extended; or the result may be unordered, in which case a new set of unanalyzed $_m$S relations will be created. This new set of S relations, if nonempty, connect processes further apart in the given order. If empty, then analysis is complete. Thus, for N total processes, there are no more than (N-1) sets of comparisons required to detect all instances of parallelism. If we are given a linear ordering of N processes, the worst case is N parallel processes. The first step would perform (N-1) input-output comparisons and produce (N-2) relations in $_m$S. Each of these relations connect processes two apart in the initial order. The second step would thus have (N-2) comparisons, and so on for (N-1) steps, until all $\frac{1}{2}$N(N-1) comparisons are made. It is necessary to retain the ability to link between any two disconnected chains of S-linked processes (each chain having 0 or more processes) until it is certain that there is no connecting S relation between the chains.

It is possible to develop T initially, and from T develop S. If all $\frac{1}{2}$N(N-1) comparisons are conceded as being required, they could all be done in parallel. Any nonempty intersection causes an entry in T. Alternatively, there may be a significant advantage in completing in parallel all comparisons of a particular process output (or output set) with all successor input sets. When done for all processes, T may be completed by forming the transitive closure.

PARALLELISM IN LANGUAGE SYNTACTIC DEFINITION

One question which has been explored is the applicability of the algorithm for the detection of parallelism to the syntactic definition of certain languages, such as ALGOL. The results might be classification of the productions of the language which can be applied in parallel, so that the syntax recognition part of the compilation process itself could be speeded up by the application of multiprocessing.

This would be essentially an experimental study, which to be effectively carried out would require a computer program for the algorithm because of the large number of syntactic classes to be considered. It appears likely that empirical separation of the syntactical elements of a compiler language would achieve results not much worse than those achieved by application of the algorithm.

Theoretically, it might be useful to have a description of classes of productions which can be performed in parallel. Some insight into the "essential complexity" of the language might be obtained in this way.

The "essential complexity" of the syntax of most programming languages may be reduced considerably by applying an algorithm developed by Parikh.[6] He shows that any context-free language can be replaced by an ambiguity preserving grammar, with all productions except S (the initial non-terminal) in the form:

$$A \rightarrow a\ A\ b$$

$$A \rightarrow c$$

where A is any specific nonterminal $\neq S$, and a, b, and c are strings of terminals with both a, b not null.

As an example of the effect of application of this replacement, let an arithmetic statement be defined by:

A_1: ⟨arithmetic statement⟩ ::= ⟨variable⟩ = ⟨arithmetic expression⟩

A_2: ⟨arithmetic expression⟩ ::= ⟨term⟩ | ⟨arithmetic expression⟩ ⟨add op⟩ ⟨term⟩

A_3: ⟨add op⟩ ::= + | -

T: ⟨term⟩ ::= ⟨factor⟩ | ⟨term⟩ ⟨mult op⟩ ⟨factor⟩

M: ⟨mult op⟩ ::= * | /

F: ⟨factor⟩ ::= ⟨integer⟩ | ⟨variable⟩ | ⟨arithmetic expression⟩

If we use the capital abbreviations preceding the nonterminal syntactic classes, the non-terminal vocabulary $V_n = \{A_1, A_2, A_3, M, T, F\}$. If we abbreviate integer by i and variable by v, the terminal vocabulary $V_t = \{i, v, +, -, *, /\}$.

In our language syntax subset, the initial nonterminal S is A_1. In effect, Parikh asserts that the only other nonterminals required are those which are directly recursive. In what follows, we show that the given definition of arithmetic statement, involving six syntactic classes (A_1, A_2, A_3, M, T, and F), can be written in terms of only two syntactic classes (A_1, T). Of course, A_1 is necessary since it is the "sentence." The single recursive syntactic class is not necessarily T (F or A_3 would have done as well). This illustrates the recognition and elimination of indirect recursion.

Table I shows the original productions rewritten as individual productions, and successive replacements yielding in the right column the reduction to two syntactic classes A_1 and T. Productions of the form T→T are eliminated as redundant.

Table I

Reduction of Syntactic Classes

Original	Replace A_2 by T	Replace F by T	Replace A_3 by + and - ; M by * and /	For two righthand T's substitute i and v for one T, then the other
$A_1 \to v=A_2$ $A_2 \to T$	$A_1 \to v=T$	$A_1 \to v=T$	$A_1 \to v=T$	$A_1 \to v=T$
$A_2 \to A_2A_3T$ $A_3 \to +$	$T \to TA_3T$ $A_3 \to +$	$T \to TA_3T$ $A_3 \to +$	$T \to T+T$	$T \to i+T$ $T \to v+T$ $T \to T+i$ $T \to T+v$
$A_3 \to -$ $T \to F$	$A_3 \to -$ $T \to F$	$A_3 \to -$	$T \to T-T$	$T \to i-T$ $T \to v-T$ $T \to T-i$ $T \to T-v$
$T \to TMF$ $M \to *$	$T \to TMF$ $M \to *$	$T \to TMT$ $M \to *$	$T \to T*T$	$T \to i*T$ $T \to v*T$ $T \to T*i$ $T \to T*v$
$M \to /$ $F \to A_2$	$M \to /$ $F \to T$	$M \to /$	$T \to T/T$	$T \to i/T$ $T \to v/T$ $T \to T/i$ $T \to T/v$
$F \to i$ $F \to v$	$F \to i$ $F \to v$	$T \to i$ $T \to v$	$T \to i$ $T \to v$	$T \to i$ $T \to v$

Application of the algorithm for the detection of parallelism to the syntactic definition of a programming language requires that the order of application of the productions be specified. The order chosen by the programmer in the syntax recognition portion of a compiler for a particular language presumably provides more nearly efficient recognition of the more likely strings. Therefore, some value judgment is required in selecting this order which is unrelated to the actual syntax. Experimental determination of the relative occurrence over a large set of programs of the various strings allowed by a language could be used as an aid in finding improved ordering for productions. Following all possible production paths will obviously recognize the string if correct, at the expense of following mostly incorrect production paths. A compromise in the number of levels in the syntax that matches the number of parallel paths that can be concurrently followed may be desirable. Introduction of other equivalent sets of productions (in other orders) may be done without reducing the power of the language. However, as our example indicates, such introduction may increase the number of productions very rapidly and complicate the recognition procedure so much that application of Parikh's result would be necessary to practicably reduce the number of syntactic classes to a minimum.

PROGRAM FOR THE NEXT INTERVAL

1. Develop techniques for recognizing instances of registers given the names in a program.

2. Continue to investigate formal program structures with emphasis on arrays.

3. Continue to identify the effect of language features on parallelism.

4. Examine criteria for partitioning a program into processes.

5. Discuss implementation of parallel analysis.

6. Describe a language for simulating the essential order detection given the instances.

BIBLIOGRAPHY

1. Detection of Implicit Computational Parallelism From Input-Output Sets, H. W. Bingham, D. A. Fisher, and W. L. Semon, December 1966, Technical Report ECOM-02463-1, Burroughs TR-66-4, AD645438.

2. Explicit Parallel Processing Description and Control in Programs for Multi- and Uni-Processor Computers, J. A. Gosden, 1966, FJCC, AFIPS, Vol. 29, pp. 651-660.

3 FORTRAN vs. Basic FORTRAN, October 1964, CACM, Vol. 7, No. 10, pp. 591-625.

4 Revised Report on the Algorithm Language ALGOL 60, P. Naur Ed., 1962-3, Computer Journal, Vol. 5, pp. 349-367.

5. On Writing an Optimizing Translator for ALGOL 60, D. H. R. Huxtable, ch. 9, in Introduction to System Programming, P. Wegner, Ed., 1964, Academic Press, pp. 137-155.

6 COBOL Edition 1965, Department of Defense, GPO.

7 A New Method of Checking the Consistency of Precedence Matrices, R. B. Marimont, 1959, JACM, Vol. 6, pp. 164-171.

8. On Context-Free Language, R. J. Parikh, October 1966, JACM, Vol. 13, No. 4, pp. 570-581.

DISTRIBUTION LIST

Second Quarterly Report
Contract DA 28-043 AMC-02463(E), Burroughs Corporation

	No. of Copies
Defense Documentation Center, ATTN: DDC-IRS, Cameron Station (Bldg 5), Alexandria, Virginia, 22314	50
Office of Assistant Secretary of Defense (Research and Engineering) ATTN: Technical Library, Rm. 3E1065, Washington, D.C. 20301	1
Director, Defense Atomic Support Agency, ATTN: Document Library Branch, Washington, D.C., 20301	1
Defense Intelligence Agency, ATTN: DIARD, Washington, D.C., 20301	1
Director, Defense Communication Agency, ATTN: CODE 333, Washington, D.C., 20305	1
Naval Ships Systems Command, ATTN: CODE 6312 (Technical Library), Main Navy Building, Rm. 1528, Washington, D.C., 20325	1
Naval Ships Systems Command, ATTN: CODE 6686B, Department of the Navy, Washington, D.C., 20360	1
Naval Ships Systems Command, ATTN: CODE 6454, Department of the Navy, Washington, D.C., 20360	1
Naval Ships Systems Command, ATTN: CODE 6745B, Department of the Navy, Washington, D.C., 20360	1
Director, U.S. Naval Research Laboratory, ATTN: CODE 2027 Washington, D.C., 20390	2
Commanding Officer and Director, U.S. Navy Electronics Laboratory, ATTN: Library, San Diego, California, 92101	1
Commander, U.S. Naval Ordnance Laboratory, ATTN: Technical Library, White Oak, Silver Spring, Maryland, 20910	1
AFSC STLO (RTSND), Naval Air Development Center, Johnsville, Warminster, Pennsylvania, 18974	1

Distribution List, Contract DA 28-043 AMC-02463(E),
Second Quarterly Report

	No. of Copies
Rome Air Development Center (EMTLD), ATTN: Documents Library, Griffiss Air Force Base, New York, 13440	1
Headquarters, Strategic Air Command, ATTN: DOCE, Offutt Air Force Base, Nebraska, 68113	1
Systems Engineering Group (SEPIR) Wright-Patterson Air Force Base, Ohio, 45433	1
Electronic Systems Division (ESTI), L. G. Hanscom Field, Bedford, Massachusetts, 01731	2
Headquarters, Ground Elct Engrg Installation Agency, ATTN: GEETR, Griffiss Air Force Base, New York, 13442	1
HQS, Air Weather Service, ATTN: AWSAE/SIPD, Scott Air Force Base, Illinois, 62225	1
U. S. Air Force Security Service, ATTN: ESD, San Antonio, Texas, 78241	1
Air Proving Ground CTR (PGBPS-12) ATTN: PGAPI Eglin Air Force Base, Florida, 32542	1
Rome Air Development Center, ATTN: Morris Knapp, Griffiss Air Force Base, N. Y., 13442	1
Headquarters, U. S. Army Munitions Command, Dover, New Jersey, 07801	1
Commanding Officer, 52D USASASOC, Fort Huachuca, Arizona, 85613	1
Commanding General, U. S. Army Combat Developments Command, ATTN: CDCMR-E, Fort Belvoir, Virginia, 22060	1
Commanding Officer, U. S. Army Sec Agcy Combat Dev Actv, Arlington Hall Station, Arlington, Virginia, 22212	1
Technical Support Directorate, ATTN: Technical Library Bldg 3330, Edgewood Arsenal, Maryland, 21010	1
Commanding Officer, U. S. Army Nuclear Defense Laboratory, ATTN: Library, Edgewood Arsenal, Maryland, 21010	2
Commanding General, U. S. Army Electronic Proving Ground, ATTN: Technical Information Center, Fort Huachuca, Arizona, 85613	1

Security Classification

DOCUMENT CONTROL DATA - R&D

(Security classification of title, body of abstract and indexing annotation must be entered when the overall report is classified)

1 ORIGINATING ACTIVITY (Corporate author) Burroughs Corporation Defense, Space and Special Systems Group Paoli, Pennsylvania	2a REPORT SECURITY CLASSIFICATION Unclassified 2b GROUP None

3 REPORT TITLE

Detection of Essential Ordering Implicit in Compiler Language Programs

4 DESCRIPTIVE NOTES (Type of report and inclusive dates)

Quarterly Progress Report, October 15, 1966 - January 20, 1967

5 AUTHOR(S) (Last name, first name, initial)

Bingham, Harvey W.
Fisher, David A.
Semon, Warren L.

6 REPORT DATE February 1967	7a TOTAL NO OF PAGES 42	7b NO OF REFS 8
8a CONTRACT OR GRANT NO DA 28-043-AMC-02463(E) b PROJECT NO ; 1E6 20501 A 485 c -03 d -01	9a ORIGINATOR'S REPORT NUMBER(S) TR-67-1 9b OTHER REPORT NO(S) (Any other numbers that may be assigned this report) ECOM-02463-2	

10 AVAILABILITY LIMITATION NOTICES

Distribution of this document is unlimited.

11 SUPPLEMENTARY NOTES	12 SPONSORING MILITARY ACTIVITY U. S. Army Electronics Command Fort Monmouth, New Jersey 07703 AMSEL-NL-P-1

13 ABSTRACT

This is the second report of an investigation was made to determine how implicit parallelism in programs written in compiler languages can be recognized and exploited by machines with highly parallel organizations. An algorithm is described which identifies the complete serial ordering among parts of a program based on the input-output sets of these parts, the ordering given by the programmer, and any known essential order among the program parts. The algorithm is proved and a demonstration given that a minimum number of comparisons of input-output sets are made. Application of the parallel recognition procedure to subroutines, loops, conditionals, recursive subroutines, and serial input-output device calls is explained. The effect of particular features of several compiler languages on parallelism are discussed. These features include loops, transfers of control, conditionals, and conditional sequences. Requirements for replacing iterative loop control by parallel paths of control are given. Alternative algorithms for recognizing essential ordering are suggested which can be executed more effectively on a highly parallel machine. Application of the given algorithm to the syntactic definition of a context-free language is also considered.

DD FORM 1 JAN 64 1473

Security Classification

14 KEY WORDS	LINK A ROLE	LINK A WT	LINK B ROLE	LINK B WT	LINK C ROLE	LINK C WT
Parallel Programming		4				
Program Flow Analysis		4				
Program Languages		2				
Context-Free Grammars		1				
Multiprocessing		1				

INSTRUCTIONS

1. ORIGINATING ACTIVITY: Enter the name and address of the contractor, subcontractor, grantee, Department of Defense activity or other organization (*corporate author*) issuing the report.

2a. REPORT SECURITY CLASSIFICATION: Enter the overall security classification of the report. Indicate whether "Restricted Data" is included. Marking is to be in accordance with appropriate security regulations.

2b. GROUP: Automatic downgrading is specified in DoD Directive 5200.10 and Armed Forces Industrial Manual. Enter the group number. Also, when applicable, show that optional markings have been used for Group 3 and Group 4 as authorized.

3. REPORT TITLE: Enter the complete report title in all capital letters. Titles in all cases should be unclassified. If a meaningful title cannot be selected without classification, show title classification in all capitals in parenthesis immediately following the title.

4. DESCRIPTIVE NOTES: If appropriate, enter the type of report, e.g., interim, progress, summary, annual, or final. Give the inclusive dates when a specific reporting period is covered.

5. AUTHOR(S): Enter the name(s) of author(s) as shown on or in the report. Enter last name, first name, middle initial. If military, show rank and branch of service. The name of the principal author is an absolute minimum requirement.

6. REPORT DATE: Enter the date of the report as day, month, year, or month, year. If more than one date appears on the report, use date of publication.

7a. TOTAL NUMBER OF PAGES: The total page count should follow normal pagination procedures, i.e., enter the number of pages containing information.

7b. NUMBER OF REFERENCES: Enter the total number of references cited in the report.

8a. CONTRACT OR GRANT NUMBER: If appropriate, enter the applicable number of the contract or grant under which the report was written.

8b, 8c, & 8d. PROJECT NUMBER: Enter the appropriate military department identification, such as project number, subproject number, system numbers, task number, etc.

9a. ORIGINATOR'S REPORT NUMBER(S): Enter the official report number by which the document will be identified and controlled by the originating activity. This number must be unique to this report.

9b. OTHER REPORT NUMBER(S): If the report has been assigned any other report numbers (*either by the originator or by the sponsor*), also enter this number(s).

10. AVAILABILITY/LIMITATION NOTICES: Enter any limitations on further dissemination of the report, other than those imposed by security classification, using standard statements such as:

(1) "Qualified requesters may obtain copies of this report from DDC."

(2) "Foreign announcement and dissemination of this report by DDC is not authorized."

(3) "U. S. Government agencies may obtain copies of this report directly from DDC. Other qualified DDC users shall request through _____."

(4) "U. S. military agencies may obtain copies of this report directly from DDC. Other qualified users shall request through _____."

(5) "All distribution of this report is controlled. Qualified DDC users shall request through _____."

If the report has been furnished to the Office of Technical Services, Department of Commerce, for sale to the public, indicate this fact and enter the price, if known.

11. SUPPLEMENTARY NOTES: Use for additional explanatory notes.

12. SPONSORING MILITARY ACTIVITY: Enter the name of the departmental project office or laboratory sponsoring (paying for) the research and development. Include address.

13. ABSTRACT: Enter an abstract giving a brief and factual summary of the document indicative of the report, even though it may also appear elsewhere in the body of the technical report. If additional space is required, a continuation sheet shall be attached.

It is highly desirable that the abstract of classified reports be unclassified. Each paragraph of the abstract shall end with an indication of the military security classification of the information in the paragraph, represented as (*TS*), (*S*), (*C*), or (*U*).

There is no limitation on the length of the abstract. However, the suggested length is from 150 to 225 words.

14. KEY WORDS: Key words are technically meaningful terms or short phrases that characterize a report and may be used as index entries for cataloging the report. Key words must be selected so that no security classification is required. Identifiers, such as equipment model designation, trade name, military project code name, geographic location, may be used as key words but will be followed by an indication of technical context. The assignment of links, rules, and weights is optional.