APPLIED DATA RESEARCH, INC.

N/R - 3rd. SEMIANNUAL - SEE AD 719 417 93

APPLIED DATA RESEARCH, INC.
LAKESIDE OFFICE PARK • WAKEFIELD, MASSACHUSETTS 01880 • (617) 245-9540

FOURTH SEMI-ANNUAL TECHNICAL REPORT
(14 July 1971 - 13 January 1972)
FOR THE PROJECT
COMPILER DESIGN FOR THE ILLIAC IV

VOLUME I

Principal Investigator and Project Leader:
Robert E. Millstein
Phone (617) 245-9540

ARPA Order Number	ARPA 1554
Program Code Number	0D30
Contractor:	Applied Data Research, Inc.
Contract No.:	DAHC04 70 C 0023
Effective Date:	13 January 1970
Amount:	$916,712.50

Sponsored by
Advanced Research Projects Agency
ARPA Order No. 1554

APPLIED DATA RESEARCH, INC
LAKESIDE OFFICE PARK • WAKEFIELD, MASSACHUSETTS 01880 • (617) 245-9540

FOURTH SEMI-ANNUAL TECHNICAL REPORT
(14 July 1971 - 13 January 1972)
FOR THE PROJECT
COMPILER DESIGN FOR THE ILLIAC IV

VOLUME I

CA-7202-1111

Principal Investigator and Project Leader:
Robert E. Millstein Phone (617) 245-9540

TABLE OF CONTENTS

VOLUME I

INTRODUCTION

This progress report consists of two volumes: the first focuses on the theoretical foundations of our work in detecting and exploiting parallelism; the second amounts to a preliminary user's manual for the ILLIAC-IV FORTRAN system. Volume 1 also contains a description of the prototype of the eventual Parallelism-Analyzer.

During this period, we have completed the following accomplishments:

- Refinement and extension of algorithms for detection and exploitation of parallelism.
- Specification of the syntax and semantics of IVTRAN, the extension of FORTRAN-IV designed for ILLIAC-IV.
- Functional specification of a powerful linking loader.
- Development of a prototype parallel analyzer.
- Near completion of the parser portion of the compiler.
- Investigation of certain algorithms required to support optimization.
- Investigation of algorithms for efficient allocation of arrays within P.E. memory (the "knap-sack" problem).
- Rationalization of the ILLIAC-IV instruction set as required for code-selection.

This report concentrates on the extended FORTRAN (IVTRAN) and algorithms for deriving it from conventional FORTRAN. The next semi-annual report will focus on optimization, data allocation, and code selection algorithms.

Volume I contains five chapters. Chapter I discusses several methods for the solution of Laplace's equation on a parallel computer by relaxation methods. Chapter II presents parallelism-detection methods for asynchronous processors while Chapter III restricts these methods to adapt them to synchronous processors. Chapter IV then considers related practical considerations. Chapter V describes a prototype parallel analyzer.

Volume II contains three chapters. The first is an introduction to ILLIAC-IV FORTRAN entitled "Control Structures in ILLIAC-IV FORTRAN". Chapter II is a preliminary version of a language manual for IVTRAN. As such, it is essentially complete with the exception of identifiable appendices which will be available as the project progresses; e.g., as the entire list of diagnostics is determined. Chapter III presents the functional specification of the supporting Linkage Editor.

In defining the language and its link editor, we have surveyed systems facilities currently in use on large scale scientific systems such as IBM 360/370 and CDC 6000/7000. Successful implementation of a useable FORTRAN system on ILLIAC-IV means that run-time operations such as format conversion, computation of standard mathematical functions, and overlay handling must be supported. We have therefore identified the need for operating system support required by a FORTRAN system and assumed its existence in some form which will be suitable to the users of ILLIAC-IV FORTRAN. Obviously, we require complete specification of these features in order to complete the code selector and the linkage editor.

CHAPTER I

THE SOLUTION OF LAPLACE'S EQUATION ON A PARALLEL COMPUTER BY RELAXATION METHODS

THE SOLUTION OF LAPLACE'S EQUATION ON A PARALLEL COMPUTER BY RELAXATION METHODS

Relaxation

The computer solution of Laplace's equation* on a two-dimensional rectangular domain with prescribed boundary values involves finding a solution of the following system of linear equations:

(1) $A(I, J) = .25 * [A(I - 1, J) + A(I + 1, J) + A(I, J - 1) + A(I, J + 1)]$,

for $1 \leq I \leq M$, $1 \leq J \leq N$, with prescribed "boundary" values for $A(0, J)$, $A(M + 1, J)$, $A(I, 0)$, $A(I, N + 1)$.

Let D denote the set $\{ (I, J) : 1 \leq I \leq M, 1 \leq J \leq N \}$ of "interior points". The ordinary relaxation method solves this system of equations by the following process:

1. Choose any initial values for $A(I, J)$ $[(I, J) \in D]$

2. Execute Equation (1) as a FORTRAN statement for each $(I, J) \in D$, in some fixed order.**

3. Repeat Step 2 until convergence is obtained - i.e., until successively computed values of $A(I, J)$ are sufficiently close, for all $(I, J) \in D$.

We will call Step 2 a single relaxation step.

The usual algorithms for the relaxation step for a sequential computer is to enclose statement (1) in the following DO loop:

---

*For convenience, we only consider Laplace's equation. The generalization to Poisson's equation is trivial.

**Actually, in the chaotic methods discussed later, the order randomly changes.

(2) DO I = 1, M

DO J = 1, N .

This gives the Gauss-Seidel method.

The obvious ways of programming the relaxation step for an array computer like the ILLIAC-IV are: (a) the Jacobi method, which uses a

DO FOR ALL (I, J) / D

loop, and (b) the row method using a

DO I = 1, M

DO FOR ALL J/[1, 2... N]

loop. One would use the row method if MN greatly exceeds the number of processors (as is usually the case for the ILLIAC). However, we will show that there are better, less obvious parallel algorithms than the Jacobi and row methods.

The theoretical relative rates of convergence for these methods are shown in the first column of Table 1.* The rate of convergence is inversely proportional to the expected number of relaxation steps needed to obtain a certain accuracy of the solution. A small number of test cases have been run, and they agree quite well with these values.

Intuitively, the convergence rates can be understood by considering the number of new values of A - values already computed during the present relaxation step - which are used in computing A(I, J). They are:**

---

*They are obtained from the spectral radius of the iteration matrix for the particular method, as defined in [1]. The entries in Table 1 for the Jacobi and Gauss-Seidel methods can be found in [1].

**We are neglecting the points adjacent to the boundary.

Jacobi	- none
row	- one [A(I - 1, J)]
Gauss-Seidel	- two [A(I - 1, J), A(I, J - 1)] .

The more new values used, the faster a method converges. Thus, the Gauss-Seidel method converges in fewer relaxation steps than the parallel methods. However, the faster parallel computation of each step more than compensates for this. Hence, the Jacobi method is faster than the row method, which is faster than the Gauss-Seidel method (assuming a large enough number of processors).

	Ordinary Relaxation	Over-Relaxation
Jacobi Method	1	1
Row Method	4/3	2
Gauss-Seidel Method	2	$\frac{1.8\ MN}{\sqrt{M^2 + N^2}}$

TABLE 1

Approximate Relative Theoretical Rates of Convergence

(Compared to Jacobi Method)

Over-Relaxation

In over-relaxation methods, statement (1) is generalized to

$$(3) \qquad A(I, J) = \omega * .25*[A(I-1, J) + A(I+1, J) + A(I, J-1) + A(I, J+1)] + (1-\omega) * A(I, J) ,$$

where ω is a fixed parameter.* (If $\omega = 1$, (3) reduces to (1).) This yields no improvement in the Jacobi method, but can speed up the other two methods. The optimal choice of ω depends upon M and N. For M, N >> 1, it is slightly less than 4/3 for the row method and slightly less than 2 for the Gauss-Seidel method. The convergence rates given in Table 1 were obtained using this optimal choice of ω for each method. For the row and Gauss-Seidel methods, the faster convergence more than compensates for the extra computations in (3).

The enormous speed-up of the Gauss-Seidel method predicted by Table 1 does not occur in practice. The theoretical convergence rates are asymptotic limits as the number of steps goes to infinity. They hold only for computations requiring very great accuracy, and thus a very large number of relaxation steps. The theoretical convergence rate for the Gauss-Seidel over-relaxation method is probably valid only when the number of steps needed in the ordinary method is >> MN. In practice, this is never the case.

---

*In practice, the following "statements" appear in the relaxation loop:

$$\Delta = .25 * [A(I-1, J) + A(I+1, J) + A(I, J-1) + A(I, J+1)] - A(I, J)$$

$$A(I, J) = \omega * \Delta + A(I, J) ,$$

and Δ is then used in the convergence test. Here, Δ is an array variable (with subscripting) for the parallel methods, and is usually a scalar for the Gauss-Seidel method.

Hyperplane Methods

The methods developed in [2] show that the Gauss-Seidel loop (2) for both the ordinary and over-relaxation methods is equivalent to the following loop:

(4) DO K = 2, M + N

DO CONC FOR ALL (I, J) ϵ { (x, y) ϵ D : x + y = K } ,

which yields the hyperplane method. The "CONC" indicates that the DO FOR ALL can be executed by independent asynchronous processors. The DO FOR ALL set

$$D_K = \{ (x, y) \epsilon D : x + y = K \}$$

is shown in Figure 1 for various values of K.

The hyperplane method naturally has the same rate of convergence as the Gauss-Seidel method. Comparing rates of convergence and number of computation steps per relaxation step, it is seen that the row method is still faster than the hyperplane method for ordinary relaxation. For over-relaxation, the hyperplane method is superior.

The hyperplane method can be pipelined as follows. While a DO FOR ALL (I, J) ϵ D_K is being executed for a certain relaxation step, the DO FOR ALL (I, J) ϵ $D_{K'}$ can be executed at the same time for the next relaxation step, so long as K' < K - 1. Forgetting about start-up and shut-down difficulties, we can thus rewrite the hyperplane method as the pipelined hyperplane method with the following loop:

(5) DO K = 0, μ - 1

DO CONC FOR ALL (I, J) ϵ { (x, y) ϵ D : x + y $\equiv$ K mod μ }

for any $\mu > 1$.

Since it is the same as the hyperplane method except for start-up and shut-down, the (theoretical) rate of convergence of the pipelined hyperplane method equals that of the Gauss-Seidel method.* Moreover, it has the same optimal value of ω for over-relaxation.

*One might expect from this argument that start-up differences would make the Gauss-Seidel method converge somewhat faster in practice. Theoretical analysis has yielded no information about such an effect. The difference in actual convergence rates, if it exists at all, is probably quite small - especially if $u \gg 1$ (as is the case for the Z-stripe method described below).

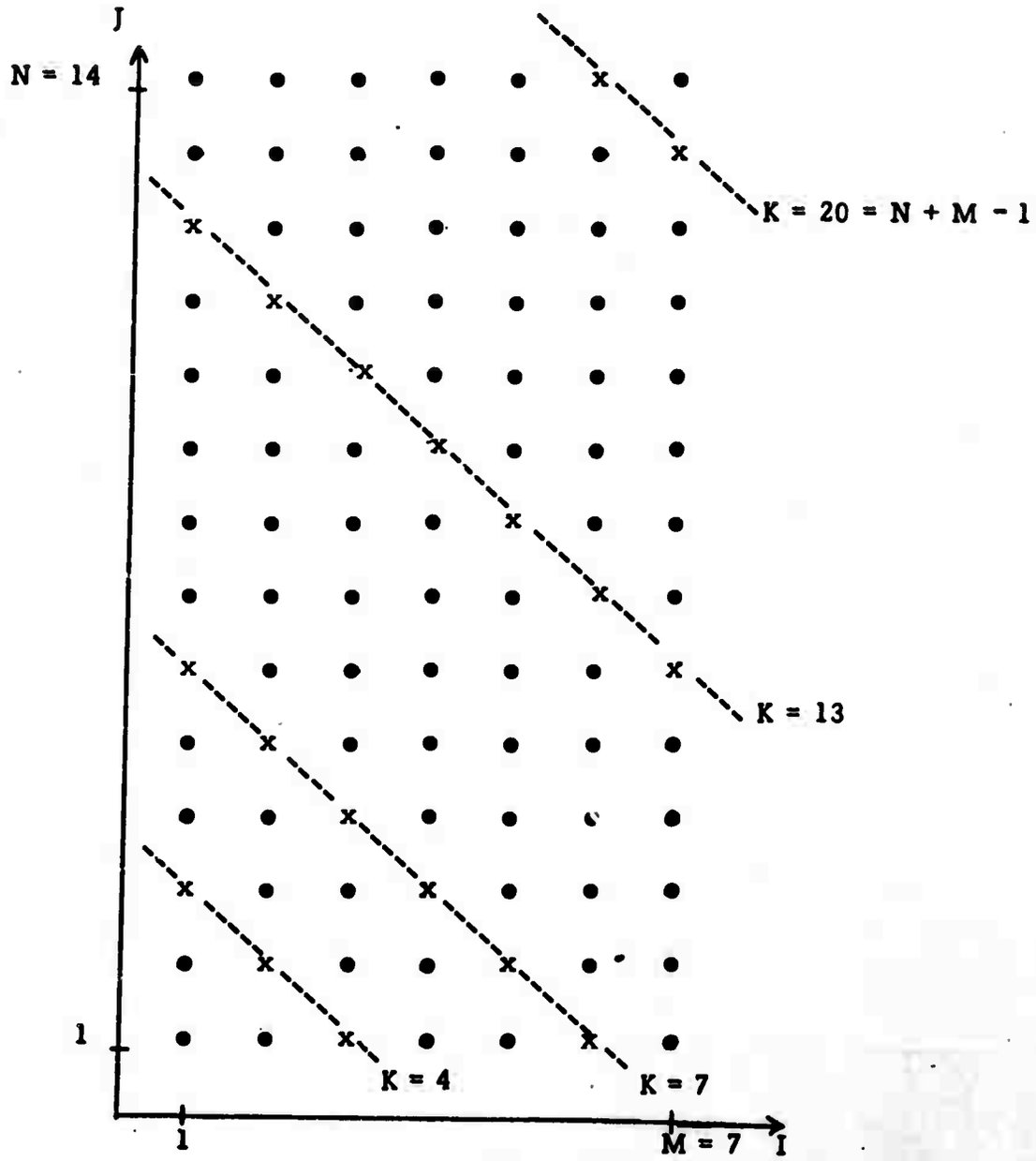


Figure 1 - The Sets D_K

The Z-Stripe Method

Setting $u = M$ in (5) gives the Z-stripe method. (We assume $N \geq M$.) Figure 2 shows the DO FOR ALL set:

$$Z_K = \{ (x, y) \in D : x + y \equiv K \bmod M \}$$

for $K = 4$.

The Z-stripe method requires the same number of sequential computations per relaxation step as the row method, but converges faster - by a factor of 1.5 for ordinary relaxation, and by a much greater factor for over-relaxation. Moreover, observe from Figure 2 that the sets Z_K always contain exactly one element from each row. The Z-stripe method is therefore well suited to the ILLIAC. The extra overhead in address calculation of the Z-stripe method compared to the row method is small. The only disadvantage of the Z-stripe method is that if $N \not\equiv 0 \bmod 64$, then there is wasted memory space in storing the array A. This space can often be used for storing other variables. Unless there is a critical shortage of memory space, the Z-stripe method should be used for the ILLIAC.

The Z-stripe method is probably the best method for the ILLIAC when $NM >> 64$, since

(i) M processors are used, and

(ii) Each computation of A(I, J) uses two new values.*

Obviously, (ii) cannot be improved upon. As for (i), any attempt to use more than M processors probably requires either a wasteful storage allocation scheme or an inefficient routing process. The most likely candidate for improving upon the Z-stripe method is the pipelined hyperplane method with $u = M/2$ when $M \leq 32$. The storage allocation problem for this method has not been studied.

---

*Except for the first M computed values, and the points adjacent to the boundary.

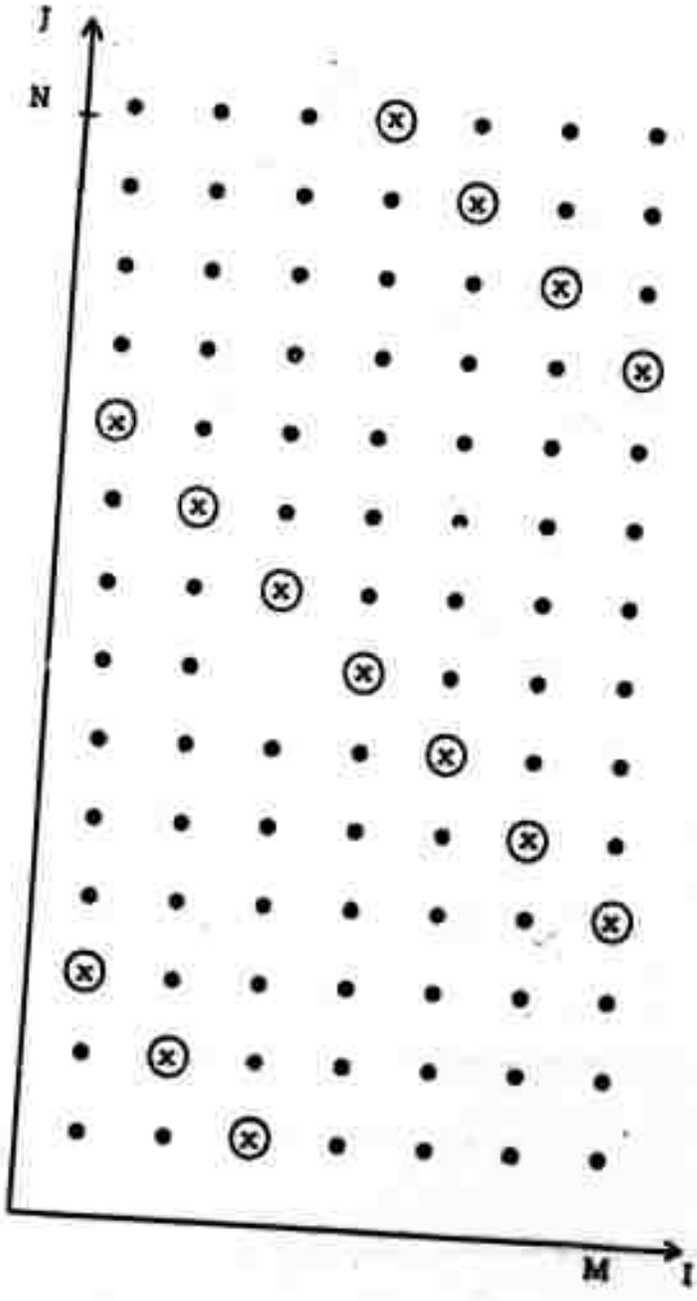


Figure 2 - The elements of Z_4 are indicated by ⊗

The Checkerboard Method

Letting $u = 2$ in the pipelined hyperplane method gives the checkerboard method. Imagine that the points of the set D are arranged in a rectangular array (as in Figures 1 and 2) and are colored alternately red and black in a checkerboard pattern. Then a single relaxation step consists of first computing A(I, J) simultaneously for all black points (I, J), then computing A(I,J) for all red points (I, J).*

Comparing the rates of convergence and number of sequential computations per relaxation step, we see that the checkerboard and Jacobi methods require the same number of computations for ordinary relaxation. However, the checkerboard method is far superior with over-relaxation. Moreover, it requires only half as many processors as the Jacobi method. Thus, the checkerboard over-relaxation method should be used on an array computer with MN/2 processors.**

Of greater practical significance, however, is the use of the checkerboard method for asynchronous processors. This is possible because of the DO CONC in (5). In general, the difficulty in using asynchronous processors for array computations is that they must be synchronized. This introduces inefficiency because some processors must be idle while waiting for others to finish. With the checkerboard method, only two synchronizations are needed per relaxation step. The inefficiency introduced is small if the number of processors is much smaller than MN/2.

Fewer synchronizations must introduce randomness into the computation. Such methods, called chaotic relaxation, have been tested by Rosenfeld [3]. However, the faster rate of convergence of the checkerboard method, especially with over-relaxation, seems to make it superior to chaotic methods despite the

---

*This assumes that the point (1,1) is colored black to conform with (5), but either coloring works equally well.

**For efficient coding, the computer probably needs $M \cdot \lceil (N+1)/2 \rceil$ processors, where $\lceil x \rceil$ denotes the smallest integer $\geq x$.

required synchronization.* Thus, the checkerboard method seems to be the best possible one for a computer with asynchronous processors - at least when the number of processors is $\leq MN/2$.

*It is not known whether there is any $\omega > 1$ for which the chaotic over-relaxation method always converges. Tests described in [3] indicate that over-relaxation does not improve the chaotic method nearly as much as it does the checkerboard method when the number of processors is $>> 1$.

BIBLIOGRAPHY

[1] Isaacson, E. and Keller, H., Analysis of Numerical Methods, John Wiley and Sons, New York, 1966.

[2] Applied Data Research, Inc., Third Semi-Annual Technical Report for the Project Compiler Design for the ILLIAC IV (13 January 1971 to 13 July 1971), CA-7110-0511, September, 1971.

[3] Rosenfeld, J., A Case Study in Programming for Parallel-Processors, Comm. ACM 12, 12 (December 1969).

CHAPTER II

ASYNCHRONOUS PROCESSORS

I. THE GIVEN LOOP

We will consider DO loops of the following form:

(1)
$$\begin{aligned} &\text{DO } \alpha\ I^1 = \ell^1, u^1 \\ &\qquad \vdots \\ &\text{DO } \alpha\ I^n = \ell^n, u^n \\ &\qquad \boxed{\text{loop body}} \\ &\alpha\ \text{CONTINUE} \end{aligned}$$

where the ℓ^i and u^i are positive integers,* and the loop body has no I/O statements, no subroutine or function calls which can modify data, and no transfer of control to any statement outside the loop. The extension to more general loops will be discussed later.

Let $\mathbb{Z}$ denote the set of all integers, and let $\mathbb{Z}^n$ denote the set of n-tuples of integers. For completeness, define $\mathbb{Z}^0 = \{0\}$.

The index set $\mathcal{J}$ of the loop (1) is defined to be the subset $\{ (i^1, \ldots, i^n) : \ell^j \leq i^j \leq u^j \}$ of $\mathbb{Z}^n$. Thus, for the loop

$$\begin{aligned} &\text{DO } 7\ I^1 = 1, 10 \\ &\text{DO } 7\ I^2 = 1, 20 \\ &\qquad \vdots \end{aligned}$$

---

*The use of superscripts and subscripts is in accord with the usual notation of tensor algebra.

$\mathcal{D} = \{ (x, y) : 1 \leq x \leq 10, x \leq y \leq 20 \}$.

An execution of the loop body for an element $(p^1, \ldots, p^n)$ of $\mathcal{D}$ is the process of setting $I^1 = p^1, \ldots, I^n = p^n$ and then executing the loop body in the usual fashion, stopping when statement α is reached. Executing the entire loop (1) then involves the execution of the loop body for each element of $\mathcal{D}$, in the order specified by the DO statements.

This suggests that we order the elements of $\mathbb{Z}^n$ lexicographically in the usual manner, with $(2, 9, 13) < (3, -1, 10) < (3, 0, 0)$. Then for any elements P and Q of $\mathcal{D}$, the loop body is executed for P before it is executed for Q if and only if $P < Q$. Thus, the relation $<$ on $\mathbb{Z}^n$ gives the appropriate temporal ordering of $\mathcal{D}$. In the preceding example, the loop body is executed for $(2, 11)$ before it is executed for $(3, 5)$, since $(2, 11) < (3, 5)$.

Define addition and subtraction of elements of $\mathbb{Z}^n$ by coordinate-wise addition and subtraction, as usual. Thus, $(3, -1, 0) + (2, 2, 4) = (5, 1, 4)$. Let $\vec{0}$ denote the element $(0, 0, \ldots, 0)$. It is easy to see that for any $P, Q \in \mathbb{Z}^n$, we have $P < Q$ if and only if $Q - P > \vec{0}$.

II. THE DO CONC STATEMENT

Our objective is to find a new temporal ordering of the executions of the loop body so that at any given time, the loop body is being executed in parallel for different elements of the index set by different processors. This new ordering must yield an algorithm which is equivalent to the one described by the original loop; i.e., one which computes the same values for all variables as the original loop.

Consider the loop

```
(2)     DO 10 I1 = 1, 3
        DO 10 I2 = 2, 7
        A(I1 + 3, I2) = 0
        10 CONTINUE
```

The loop body could be executed in parallel by three processors for the points (1, 6), (2, 5), and (3, 4) of $\mathcal{I}$. (In fact, it could be executed in parallel by 18 processors for all points in $\mathcal{I}$.)

In order to have a means of expressing parallel computation, we define the DO CONC (for CONCurrently) statement. Its form is

DO α CONC FOR ALL I ϵ $\mathcal{I}$

where $\mathcal{I}$ is a finite subset of $\mathbb{Z}$.* It has the following meaning: Let

*We remind the reader that a set is just an unordered collection of elements, so $\{1,2\} = \{2,1\} = \{1,2,1,1,2\}$. We will not bother to define a syntax for expressing sets. The usual FORTRAN DO syntax, which can only describe a restricted class of subsets of $\mathbb{Z}$, is probably the most convenient to implement.

$\mathcal{S} = \{i_1, \ldots, i_m\}$, where no two i_j are equal, and assume that we have m independent, completely asynchronous processors numbered 1 through m. Then each processor is to execute the statements following the DO CONC statement, through statement α, with processor number j setting $I = i_j$. The m processors are to run concurrently, independent of one another.

As an example, consider

```
DO 10 CONC FOR ALL J ε {x: 2 ≦ x ≦ 5}
10  A(J) = J ** 2
```

This sets A(2) = 4, A(3) = 9, A(4) = 16 and A(5) = 25.

For a DO CONC to give a well-defined algorithm, certain restrictions must be made on the statements in its range. Suppose the statement

```
9       B(J) = A(J + 1)
```

is inserted before statement 10 above. The resulting DO CONC loop does not give well-defined results. For example, the processor doing the computation for J = 3 sets B(3) to the value of A(4). But the value of A(4) it uses depends upon whether or not the processor for J = 4 has already executed statement 10. Since the processors are assumed to be asynchronous, the resulting value of B(3) is not well-defined.

We will not bother specifying the necessary restrictions on the DO CONC loop. The DO CONCs which will be written appear in loops which are equivalent to the original DO loop (1), and hence must give well-defined algorithms.

The DO CONC statement is generalized to the form

DO α CONC FOR ALL $(I^1, \ldots, I^k) \in \mathcal{S}$.

where $\mathcal{S}$ is a subset of $\mathbb{Z}^k$. The meaning should be clear: for each element $(p^1, \ldots, p^k) \in \mathcal{S}$, we have a processor performing the calculation for $I^1 = p^1, \ldots, I^k = p^k$.

III. REWRITING THE LOOP

Consider loop (2), with index set $\mathcal{I}$. Changing the order of execution of the loop body for the different elements of $\mathcal{I}$ obviously does not change the algorithm. The loop can, therefore, be rewritten as a single DO CONC, or in many different ways as a nested DO/DO CONC loop. Choosing one of these ways, we rewrite it as follows:

(3) DO 10 $J^1 = 3, 10$

DO 10 CONC FOR ALL $J^2 \in \{y: 2 \leq y \leq 7 \text{ and } J^1 - 3 \leq y \leq J^1 - 1\}$

$A(J^1 - J^2 + 3, J^2) = 0$

10 CONTINUE

The choice is arbitrary and unnatural, but instructive.

To actually construct loop (3), we first defined the one-to-one mapping $J: \mathbb{Z}^2 \to \mathbb{Z}^2$ by

$$J[(I^1, I^2)] = (I^1 + I^2, I^2) = (J^1, J^2),$$

as illustrated in Figure 1. We next defined the index set $\mathcal{J}$ to be the set $J(\mathcal{I}) = \{J(p): p \in \mathcal{I}\}$. Then $\mathcal{J} = \{(J^1, J^2): 3 \leq J^1 \leq 10, 2 \leq J^2 \leq 7$ and $J^1 - 3 \leq J^2 \leq J^1 - 1\}$, and we filled in the limits of the DO and DO CONC statements to give this index set. Finally, we rewrote the loop body in such a way that executing the body of loop (3) for the point $J(P) \in \mathcal{J}$ is equivalent to executing the body of loop (2) for the point $P \in \mathcal{I}$. In other words, $A(J^1 - J^2 + 3, J^2)$ references the same array element as $A(I^1 + 3, I^2)$ when $(J^1, J^2) = J[(I^1, I^2)]$.

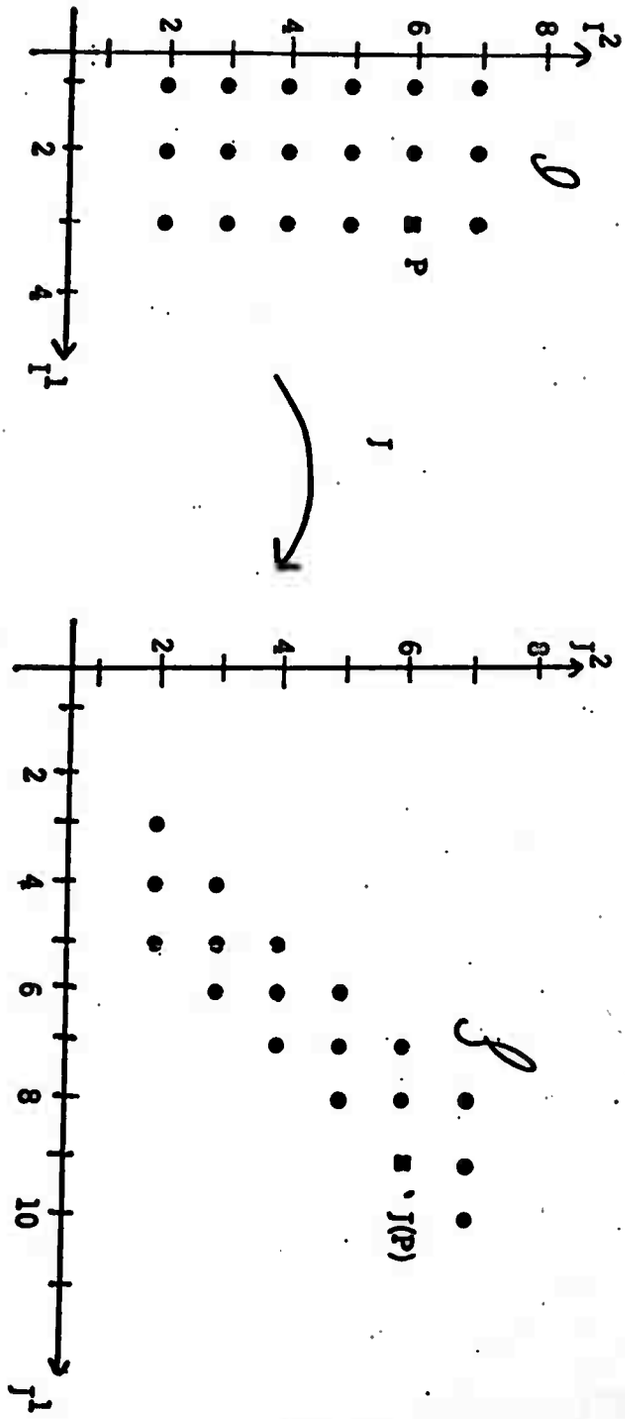


Figure 1

We can consider loop (3) to be the same as loop (2), except with a different order of execution of the body for the elements of $\mathcal{D}$. This order of execution is illustrated in Figure 2. The loop body is executed concurrently for all points in $\mathcal{D}$ lying on a straight line J^1 = constant. The execution for those points of $\mathcal{D}$ with $J^1 = 3$ precedes the execution for the points with $J^1 = 4$, which in turn precedes the execution for the points with $J^1 = 5$, etc.

This suggests that we define the mapping $\pi: \mathbb{Z}^2 \to \mathbb{Z}$ by

$$\pi [(I^1, I^2)] = J^1 = I^1 + I^2$$

Then the execution for $P \in \mathcal{D}$ precedes the execution for $Q \in \mathcal{D}$ if and only if $\pi(P) < \pi(Q)$. If $\pi(P) = \pi(Q)$, then the two executions of the loop body are concurrent.

The generalization of this rewriting procedure is straightforward. Loop (1) will be rewritten in the form

(4)
```
      DO α J^1 = λ^1, μ^1
                  .
                  .
                  .
      DO α J^k = λ^k, μ^k
      DO α CONC FOR ALL (J^(k+1), ..., J^n) ∈ S_(J^1, ..., J^k)
          loop body
α     CONTINUE
```

where $\mathcal{S}_{J^1, \ldots, J^k}$ is a subset of $\mathbb{Z}^{n-k}$ which may depend upon the values of $J^1, \ldots J^k$. Here, λ^i and μ^i need not be integers, but may be integer

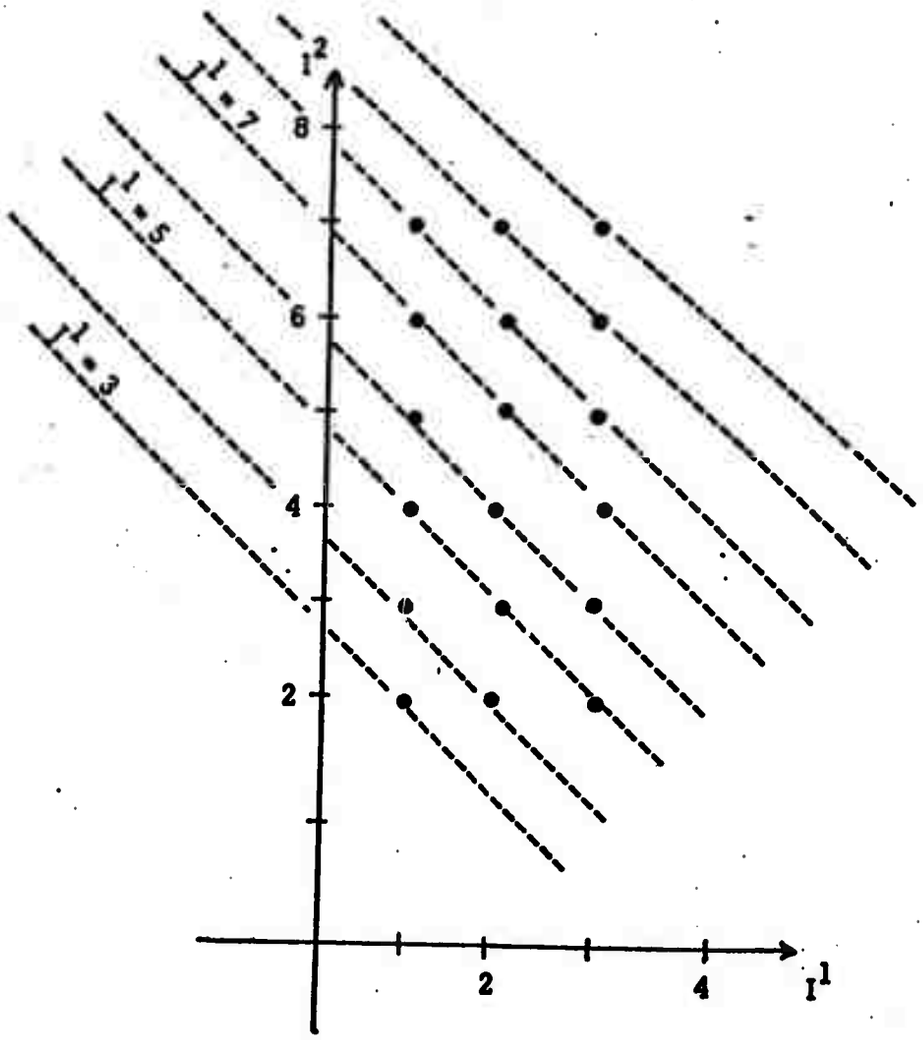


Figure 2

valued expressions whose values depend upon $J^1, \ldots, J^{i-1}$.

To perform this rewriting, we will construct a one-to-one mapping $J: \mathbb{Z}^n \to \mathbb{Z}^n$ of the form

$$(5) \quad J[(I^1, \ldots, I^n)] = (\sum_{j=1}^{n} a_j^1 I^j, \ldots, \sum_{j=1}^{n} a_j^n I^j) = (J^1, \ldots, J^n)$$

for integers a_j^i.* We then choose the λ^i, u^i and $\mathcal{S}_{J^1, \ldots, J^k}$ so that the index set $\mathcal{S}$ of the loop (4) equals $J(\mathcal{D})$, and write the body of loop (4) so that its execution for the point $J(P) \in \mathcal{S}$ is equivalent to the execution of the body of loop (1) for $P \in \mathcal{D}$.

Define the mapping $\pi: \mathbb{Z}^n \to \mathbb{Z}^k$ by

$$\pi[(I^1, \ldots, I^n)] = (J^1, \ldots, J^k) ,$$

so $\pi(P)$ consists of the first k coordinates of $J(P)$. It is then clear that for any points $J(P), J(Q) \in \mathcal{S}$, the execution of the body of loop (4) for $J(P)$ precedes the execution for $J(Q)$ if and only if $\pi(P) < \pi(Q)$. If we consider loop (4) to be a reordering of the execution of loop (1), this statement is equivalent to the following:

(E) For any $P, Q \in \mathcal{D}$, the execution of the loop body for P precedes that for Q, in the new ordering of executions, if and only if $\pi(P) < \pi(Q)$.

---

* J is one-to-one if and only if (5) can be solved to write the I^j as linear expressions in the J^i with integer coefficients.

The loop body is executed concurrently for all elements of $\mathcal{I}$ lying on a set of the form $\{P: \pi(P) = \text{constant} \in \mathbb{Z}^k\}$. Since J is assumed to be a one-to-one linear mapping, these sets are parallel (n-k)-dimensional planes in $\mathbb{Z}^n$.* We thus have concurrent execution of the loop body along (n-k)-dimensional planes through the index set.

Naturally, we cannot use any arbitrary mapping J. We must find one for which loop (4) gives an algorithm equivalent to that of loop (1). This is the goal of the following analysis.

Observe that rewriting loop (1) so all executions are concurrent; i.e., with a

DO α CONC FOR ALL $(I^1, \ldots, I^n) \in \mathcal{I}$

statement, involves setting J equal to the identity mapping, $k = 0$, and $\pi: \mathbb{Z}^n \to \mathbb{Z}^0$ the mapping defined by $\pi(P) = 0$ for all $P \in \mathbb{Z}^n$.

---

*We consider $\mathbb{Z}^n$ to be a subset of ordinary Euclidean n-space, as we did in drawing Figures 2 and 3.

IV. THE BASIC RULE

We first introduce some terminology to aid the discussion. Consider the variable VAR defined by the statement

DIMENSION VAR (10, 20) .

The range of VAR is the set $\mathcal{R}_{VAR} = \{ (x,y): 1 \leq x \leq 10, 1 \leq y \leq 20\}$, which is a subset of $\mathbb{Z}^2$. Thus, $\mathcal{R}_{VAR}$ is the set of all $(x,y) \in \mathbb{Z}^2$ such that VAR (x,y) is defined.*

An occurrence of VAR is any appearance of it in the loop body. If the occurrence appears as

VAR (-,-) = ...,

then it is called a generation; otherwise it is called a use. I.e., generations modify the values of elements of the array VAR, and uses do not.

Let f denote an occurrence of VAR in loop (1) of the form VAR ($I^1 + 3, I^3$), and assume n = 3. During execution of the loop body for the element $(3, 4, 5) \in \mathcal{S}$, this occurrence becomes VAR (6, 5). We say that f references the point $(6, 5) \in \mathcal{R}_{VAR}$ for (3, 4, 5).

This defines an occurrence mapping $T_f: \mathcal{S} \to \mathcal{R}_{VAR}$ by

---

*For a scalar variable x, we set $\mathcal{R}_x = \mathbb{Z}^0$.

letting $T_f(P)$ be the point of $\mathcal{R}_{VAR}$ referenced by f for $P \in \mathcal{I}$. In this case, T_f is given by

$$T_f[\,(p^1, p^2, p^3)\,] = (p^1 + 3, p^3)\,.$$

We will assume that all variable occurrences only have the loop variables $I^1, \ldots, I^n$ and integer constants in their subscript expressions. Then for any variable occurrence g, the occurrence mapping $T_g: \mathbb{Z}^n \to \mathbb{Z}^m$ is well-defined, where m is the dimension (number of subscript positions) of the variable.

Now consider the loop

```
(6)     DO  23  I1 = 2, 10
        DO  23  I2 = 3, 17
21      A (I1, I2) = C (I1)
          a1       c1
22      B(I1, I2) = A (I1 -1, I2 + 1) + B (I1, I2 )
          b1          a2                  b2
23      CONTINUE.
```

We have introduced the convention of writing the name of an occurrence in a circle beneath it. For the point $(4, 7) \in \mathcal{I}$, the loop body is

```
21      A (4, 7) = C (4)
22      B (4, 7) = A (3, 8) + B (4, 7) .
```

The value A (3, 8) used in statement 22 is the one computed in statement 21 during execution of the loop body for the point (3,8). To ensure that

the execution for (4,7) computes the right value when we change the order of executions of the body, we must only require that it be preceded by the execution for (3,8). By statement E above, this means that π must satisfy $\pi[(3,8)] < \pi[(4,7)]$.

In general, let VAR be any variable. If a generation and a use of VAR both reference the same element in the range of VAR during execution of the loop, then the order of the references must be preserved. In other words, if f is a generation and g is a use of VAR, and $T_f(P) = T_g(Q)$ for some points $P, Q \in \mathcal{J}$, then:

(i) If $P < Q$, we must have $\pi(P) < \pi(Q)$,

(ii) If $Q < P$, we must have $\pi(Q) < \pi(P)$.

In the above example, $T_{a_1}[(3,8)] = T_{a_2}[(4,7)] = (3,8)$ and $(3,8) < (4,7)$, so we must have $\pi[(3,8)] < \pi[(4,7)]$. Note that if $P = Q$, then the order of execution of the references will automatically be preserved, since they happen during a single execution of the loop body. Thus, the fact that $T_{b_1}[(4,7)] = T_{b_2}[(4,7)]$ does not place any restriction on our choice of π.

The above rule should also apply to any two generations of a variable. This guarantees that the variable has the correct values after the loop is run. Together with the above rule, it also ensures that a use will always obtain the value assigned by the correct generation.

These remarks can be combined into the following basic rule:

(C1) For every variable, and every ordered pair of occurrences f,g of that variable, at least one of which is a generation: if $T_f(P) = T_g(Q)$ for $P, Q \in \mathcal{O}$ with $P < Q$, then π must satisfy the relation $\pi(P) < \pi(Q)$.

Notice that the case $Q < P$ is obtained by interchanging f and g.

Rule C1 ensures that the new ordering of executions of the loop body preserves all relevant orderings of variable references. The orderings not necessarily preserved are those between references to different array elements, and between two uses. Changing just these orderings cannot change the value of anything computed by the loop. The assumptions we have made about the loop body, especially the assumption that it contains no exits from the loop, therefore imply that rule C1 gives a sufficient condition for loop (2) to be equivalent to loop (1).*

*For most loops, C1 is also a necessary condition.

V. THE SETS <f, g>

The trouble with rule C1 is that it requires that we consider many pairs of points P, Q in $\mathcal{I}$. For the loop (6), there are 112 pairs of elements P, Q, $\epsilon \mathcal{I}$ with $T_{a_1}(P) = T_{a_2}(Q)$ and $P < Q$. However, $T_{a_1}(P) = T_{a_2}(Q)$ if and only if $Q = P + (1, -1)$. We would like to be able to work with the single point $(1, -1) \epsilon \mathbb{Z}^n$, rather than all the pairs P,Q.

This suggests the following definition. For any occurrences f, g of a variable in loop (1), define the subset <f, g> of $\mathbb{Z}^n$ by

$$\langle f, g\rangle = \{ X : T_f(P) = T_g(P+X) \text{ for some } P \epsilon \mathbb{Z}^n \} .$$

Thus, for loop (6) we have <a1, a2> = { (1, -1) }, and <b1, b2> = { (0,0) }. Observe that <f, g> is independent of the index set $\mathcal{I}$.

We now rewrite rule C1 in terms of the sets < f, g>. First, note that $\pi(P + X) = \pi(P) + \pi(X)$, since we have assumed π to be a linear mapping. (Recall the definition of π, and formula (5).) Also, remember that $A < A + B$ if and only if $B > \vec{0}$. Then just substituting P + X for Q in rule C1 yields:

(C1') For ... generation: if $T_f(P) = T_g(P + X)$ for P, $P + X \epsilon \mathcal{I}$ with $X > \vec{0}$, then π must satisfy the relation $\pi(X) > \vec{0}$.

Removing the clause "for P, $P + X \epsilon \mathcal{I}$ " from C1' gives a stronger

condition for π to satisfy. Doing this and using the definition of $<f,g>$ then gives the following more stringent rule:

(C2) For every variable, and every ordered pair of occurrences f,g of that variable, at least one of which is a generation: for every $X \in <f,g>$ with $X > \vec{0}$, π must satisfy $\pi(X) > \vec{0}$.

Any π satisfying C2 also satisfies C1. Hence, rule C2 gives a sufficient condition for loop (4) to be equivalent to loop (1). Moreover, C2 is independent of the index set $\mathcal{I}$.

Note here that C2 is satisfied by the zero mapping $\pi: \mathbb{Z}^n \to \mathbb{Z}^0$ if and only if it is vacuous; i.e., if and only if there are no elements $X > \vec{0}$ in any of the sets $<f,g>$ referred to in the rule. In this case, the loop body can be executed concurrently for all points in $\mathcal{I}$.

VI. COMPUTING THE SETS $<f,g>$

We will obtain results about the existence of mappings π satisfying rule C2. In order to do this, some restrictions must be made on the forms of the occurrences to permit a simple computation of the sets $<f,g>$. We assume that each occurrence of a variable VAR is of the form

$$(7) \qquad \text{VAR } (I^{j_1} + m^1, \ldots, I^{j_r} + m^r) ,$$

where the m^k are integer constants, and $j_1, \ldots, j_r$ are r distinct integers between 1 and n. Moreover, we assume that the j_k are the same for any two occurrences of VAR. Thus, if an occurrence A $(I^2 - 1, I^1, I^4 + 1)$ appears in the loop, then the occurrence A $(I^2 + 1, I^1 + 6, I^4)$ may appear. However, the occurrence A $(I^1 - 1, I^2, I^4)$ may not .

Now let f be the occurrence (7) and let g be the occurrence VAR $(I^{j_1} + n^1, \ldots, I^{j_r} + n^r)$. Then

$$T_f [(p^1, \ldots, p^n)] = (p^{j_1} + m_1, \ldots, p^{j_r} + m_r)$$

$$T_g [(p^1, \ldots, p^n)] = (p^{j_1} + n_1, \ldots, p^{j_r} + n_r)$$

It is easy to see from the definition that $<f,g>$ is the set of all elements of $\mathbb{Z}^n$ whose j_k^{th} coordinate is $m^k - n^k$, for $k = 1, \ldots, r$, and whose remaining $n - r$ coordinates are any integers.

As an example, suppose $n = 5$ and f, g are the occurrences VAR $(I^3 + 1, I^2 + 5, I^5)$, VAR $(I^3 + 1, I^2, I^5 + 1)$. Then $\langle f, g \rangle = \{ (x, 5, 0, y, -1) : x, y \in \mathbb{Z} \}$. We will denote this set by $(*, 5, 0, *, -1)$, so "*" means "any integer".

The index variable I^j is said to be <u>missing from</u> VAR if I^j is not one of the I^{j_k} in (7). It is clear that I^j is missing from VAR if and only if the set $\langle f, g \rangle$ has an * in the $j^{\underline{th}}$ coordinate, for any occurrences f, g of VAR.

VII. THE HYPERPLANE THEOREM

I^j is called a missing index variable if it is missing from some generated variable in the loop; i.e., if it is missing from some variable for which a generation appears in the loop body.

The following result is an important special case of a more general result which will be given later.*

Hyperplane Concurrency Theorem: Assume that none of the index variables $I^2, \ldots, I^n$ is a missing variable. Then loop (1) can be rewritten in the form of loop (4) for $k = 1$. Moreover, the mapping I used for the rewriting can be chosen to be independent of the index set $\mathcal{S}$.

Proof: First, a mapping $\pi: \mathbb{Z}^n \to \mathbb{Z}$ will be constructed which satisfies rule C2. Let $\mathcal{P}$ be the set consisting of all the elements $X > \vec{0}$ of all the sets $<f,g>$ referred to in C2. We must construct π so that $\pi(X) > 0$ for all $x \in \mathcal{P}$.

Let "+" denote any positive integer, so $(+, x^2, \ldots, x^n)$ is any element of $\mathbb{Z}^n$ of the form $(x, x^2, \ldots, x^n)$ with $x > 0$. Since I^1

---

*A weaker version of this result can be found in [2].

is the only index variable which may be missing, we can write $\mathcal{P} = \{X_1, \ldots, X_n\}$, where

$$X_r = \begin{cases} (x_r^1, \ldots, x_r^n) \\ \text{or} \\ (+, x_r^2, \ldots, x_r^n) \end{cases}$$

for some integers x_r^i.

The mapping π is defined by

$$(8) \qquad \pi[\,(I^1, \ldots, I^n)\,] = a_1 I^1 + \ldots + a_n I^n$$

for non-negative integers a_j, to be chosen below. Since $a_1 \geq 0$, $\pi[\,(1, x_r^2, \ldots, x_r^n)\,] > 0$ implies $\pi[\,(x, x_r^2, \ldots, x_r^n)\,] > 0$ for any integer $x > 0$. Therefore, each X_r of the form $(+, x_r^2, \ldots, x_r^n)$ can be replaced by $X_r = (1, x_r^2, \ldots, x_r^n)$ and it is sufficient to construct π such that $\pi(X_r) > 0$ for each $r = 1, \ldots, N$. Recall that each $X_r > 0$.

Define $\mathcal{P}_j = \{X_r : x_r^1 = \ldots = x_r^{j-1} = 0, x_r^j \neq 0\}$, so $\mathcal{P}_j$ is the set of all X_r whose $j^{\underline{th}}$ coordinate is the left-most non-zero one. Then each X_r is an element of some $\mathcal{P}_j$.

Now construct the a_j sequentially for $j = n, n-1, \ldots, 1$ as follows. Let a_j be the smallest non-negative integer such that

$$a_j x_r^j + \ldots + a_n x_r^n > 0$$

for each $X_r = (0, \ldots, 0, x_r^j, \ldots, x_r^n) \in \mathcal{P}_j$. Since $X_r > \vec{0}$ and

$x_r^j \neq 0$ implies $x_r^j > 0$, this is possible.

Clearly, we have $\pi(X_r) > 0$ for all $X_r \in \mathcal{P}_j$. But each X_r is in some $\mathcal{P}_j$, so $\pi(X_r) > 0$ for each $r = 1, \ldots, n$. Thus, π satisfies rule C2. Observe that the first non-zero a_j that was chosen must equal 1, so 1 is the greatest common divisor of the a_j. (If all the a_j are zero, then $\mathcal{P}$ must be empty, so we can let $\pi[(I^1, \ldots, I^n)] = I^1$.) A classical number theoretic calculation, described on Page 31 of [3], and reproduced in Appendix A, then gives a one-to-one linear mapping $J: \mathbb{Z}^n \to \mathbb{Z}^n$ such that

$$J[(I^1, \ldots, I^n)] = (\pi[(I^1, \ldots, I^n)], \ldots)$$

Since the sets $<f,g>$ are independent of the index set $\mathcal{I}$, the construction of π and J given above is also independent of $\mathcal{I}$. This completes the proof. □

Loop (4) for $k = 1$ executes the loop body concurrently for all points in $\mathcal{I}$ lying along parallel $(n-1)$-dimensional hyperplanes, hence the name of the theorem.

Observe that the theorem is trivially true without the restriction that J be independent of $\mathcal{I}$, because given any set $\mathcal{I}$ we can construct a J for which the sets $\mathcal{I}_{J^2, \ldots, J^n}$ contain at most one element, and the order of execution of the loop body is unchanged. For example, if $\mathcal{I} = \{(x,y,z): 1 \leq x \geq 10, 1 \geq y \leq 5, 1 \leq z \leq 7\}$, let $J[(x,y,z)] = (35x + 7y + z, x, y)$. Such a J is clearly of no interest. However,

because the mapping J provided by the theorem depends only on the loop body, it will always give real concurrent execution for a large enough index set.

Condition C2 gives a set of constraints on the mapping $\pi: \mathbb{Z}^n \to \mathbb{Z}$. The Hyperplane Theorem proves the existence of a π satisfying those constraints. We now consider the problem of making an optimal choice of π.

It seems most reasonable to minimize the number of steps in the outer DO J^1 loop of loop (4). (Remember that $k = 1$.) If a sufficiently large number of processors are available, then this gives the maximum amount of concurrent computation. This means that we must minimize $\mu^1 - \lambda^1$ in loop (4). But λ^1 and μ^1 are just the upper and lower bounds $\{\pi(P): P \in \mathcal{P}\}$. Setting

$$M^i = u^i - \ell^i ,$$

it is easy to see that $u^1 - \lambda^1$ equals

$$(9) \qquad M^1 |a_1| + \ldots + M^n |a_n| ,$$

where the a_i are defined by (8). Finding an optimal π is thus reduced to the following integer programming problem: find integers $a_1, \ldots, a_n$ satisfying the constraint inequalities given by rule C2, which minimize the expression (9).

Observe that the greatest common divisor of the resulting a_i

must be 1. This follows because the constraints are of the form

$$x^1 a_1 + \ldots + x^n a_n > 0 \quad ,$$

so dividing the a_i by their g.c.d. gives new values of a_i satisfying the constraints, with a smaller value for (9). Hence, the method of [3] can be applied to find the mapping J.

Although the above integer programming problem is algorithmically solvable, we know of no practical method of finding a solution in the general case. However, the construction used in proving the Hyperplane Theorem should provide a good choice of π. In fact, for most reasonable loops it actually gives the optimal solution.

VIII. AN EXAMPLE

Now consider the following loop:

```
(10)    DO  16  I1 = 1, 25
        DO  16  I2 = 2, 19
        DO  16  I3 = 2, 29
        F (I2, I3) = (F (I2 + 1, I3) + F (I2, I3 + 1)
                      (f1)           (f2)          (f3)
        x          + F (I2 - 1, I3) + F (I2, I3 - 1) ) * .25
                     (f4)           (f5)
        16      CONTINUE .
```

It is a simplified version of a standard relaxation computation for a 20 by 30 array F, performed 25 times.

To apply the method of analysis, first perform the following calculations:

1. Compute the sets $\langle f,g \rangle$ referred to by rule C2.
2. Find all elements $X > \vec{0}$ in these sets.
3. Find the constraints on the a_i implied by $\pi(X) > \vec{0}$.

This is done in Table 1.

Next, choose a_1, a_2, a_3 consistent with these constraints,

Sets	Elements > $\vec{0}$	Constraints
<f1, f1> = (*, 0, 0)	(+, 0, 0)	$a_1 > 0$
<f1, f2> = (*, -1, 0)	(+, -1, 0)	$a_1 - a_2 > 0$
<f2, f1> = (*, 1, 0)	(+, 1, 0)	$a_1 + a_2 > 0$
	(0, 1, 0)	$a_2 > 0$
<f1, f3> = (*, 0, -1)	(+, 0, -1)	$a_1 - a_3 > 0$
<f3, f1> = (*, 0, 1)	(+, 0, 1)	$a_1 + a_3 > 0$
	(0, 0, 1)	$a_3 > 0$
<f1, f4> = (*, 1, 0)	same as <f2, f1>	
<f4, f1> = (*, -1, 0)	same as <f1, f2>	
<f1, f5> = (*, 0, 1)	same as <f3, f1>	
<f5, f1> = (*, 0, -1)	same as <f1, f3>	

Table 1

and minimizing

$$24 \mid a_1 \mid + 17 \mid a_2 \mid + 27 \mid a_3 \mid .$$

It is easy to see that the solution to this problem is $a_1 = 2$, $a_2 = 1$, $a_3 = 1$, so π is given by

$$\pi [(I^1, I^2, I^3)] = 2 I^1 + I^2 + I^3 .$$

Note that this is the π computed by the algorithm used in the proof of the Hyperplane Theorem.

Application of the algorithm described in the appendix gives the following mapping J:

$$J [(I^1, I^2, I^3)] = (J^1, J^2, J^3) = (2 I^1 + I^2 + I^3, I^1, I^3).*$$

Using this, and the inverse relation

$$(I^1, I^2, I^3) = (J^2, J^1 - 2J^2 - J^3, J^3) ,$$

the above loop is rewritten as follows:

```
      DO  16  J1 = 6, 98
      DO  16  CONC FOR ALL (J2, J3) ε { (x,y):
     X          1 ≤ x ≤ 25, 2 ≤ y ≤ 29 and J1 - 19 ≤ 2x + y ≤ J1 - 2 }
      F (J1 - 2*J2 - J3, J3) = (F (J1 - 2 * J2 - J3 + 1, J3) +
     X          F (J1 - 2 * J2 - J3, J3 + 1) + F (J1 - 2 * J2 - J3 - 1, J3)
```

---

*It is also easy to obtain this from the following fact: the mapping $J: \mathbb{Z}^n \to \mathbb{Z}^n$ defined by (5) is one-to-one if and only if the determinant of the a^i_j is ± 1 .

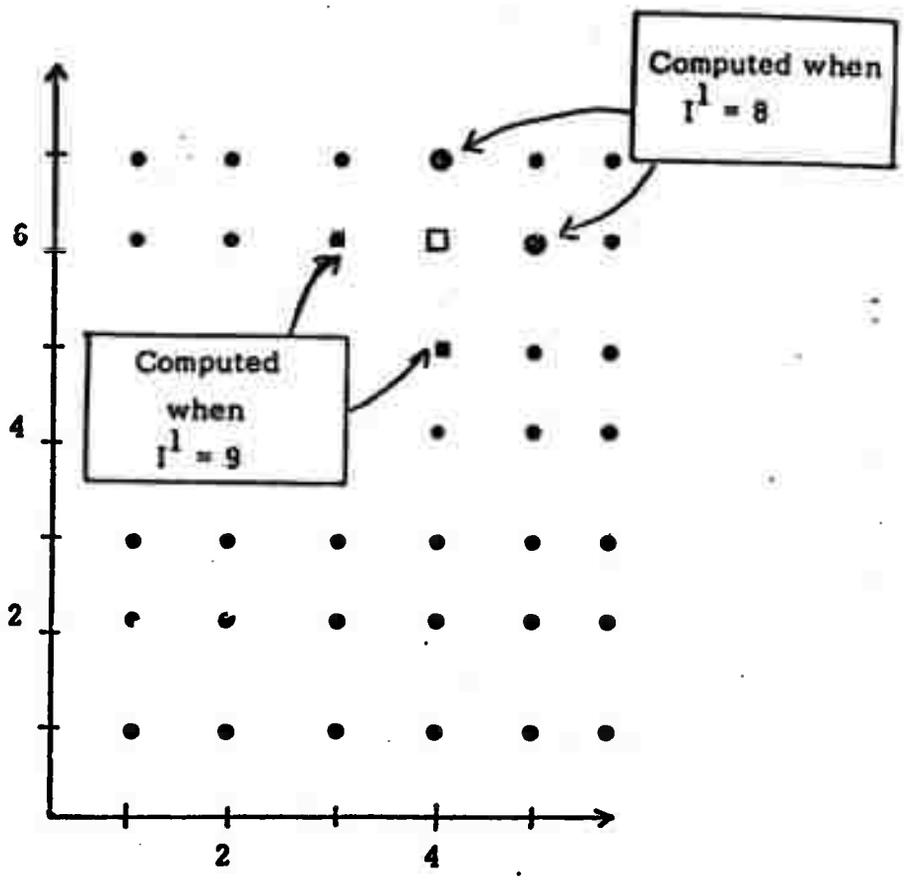
```
     X                +F (J1 - 2 * J2 - J3, J3 - 1) )* .25
16   CONTINUE
```

The set expression in the DO CONC statement was obtained by writing the relations $1 \le I^1 \le 25$, $2 \le I^2 \le 19$ and $2 \le I^3 \le 29$ in terms of J^1, J^2, J^3.

To understand why the rewritten loop gives the same results, consider the computation of F (4, 6) in the execution of the original loop body for the element (9, 4, 6) ϵ $\mathcal{I}$. It is set equal to the average of its four neighboring array elements: F (5, 6), F (4, 7), F (3, 6), F (4, 5). The values of F (5, 6) and F (4, 7) were calculated during the execution of the loop body for (8, 5, 6) and (8, 4, 7), respectively; i.e., during the previous execution of the DO I^1 loop, with $I^1 = 8$. The values of F (3, 6) and F (4, 5) were calculated during the current execution of the outer DO loop, with $I^1 = 9$. This is shown in Figure 3.

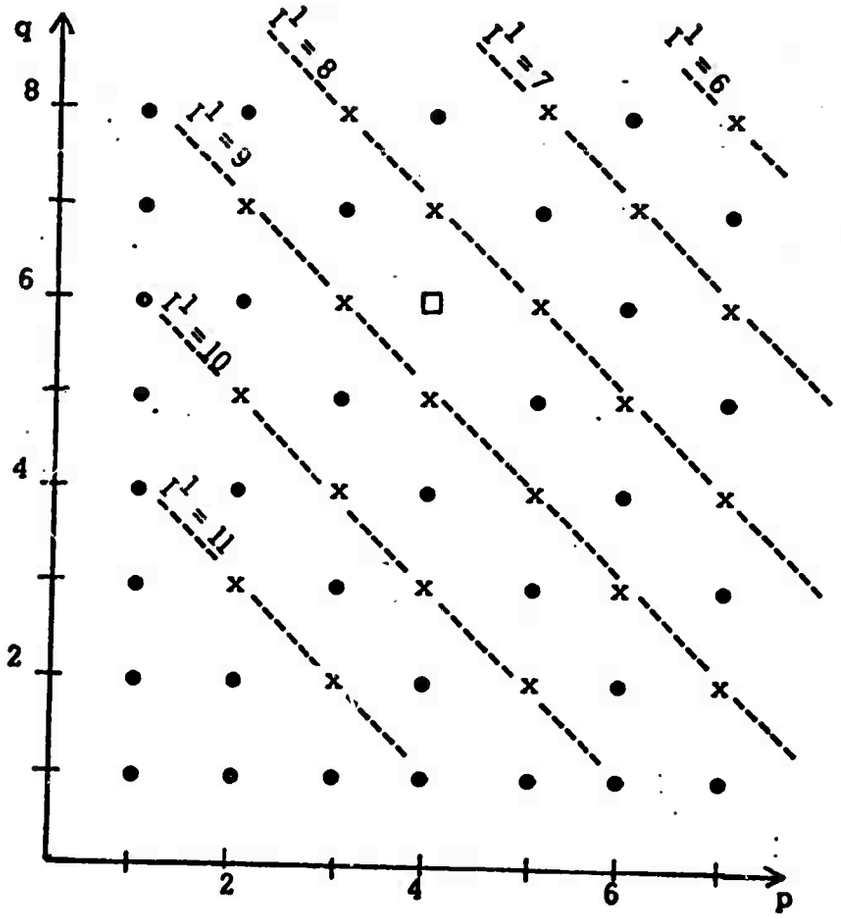
Now consider the rewritten loop. At any time during its execution, F (p,q) is being computed concurrently for up to half the elements (p,q) in the range set $\mathcal{R}_F$ of F. These computations are for different values of I^1. Figure 4 illustrates the execution of the DO CONC for $J^1 = 27$. The points (p,q) ϵ $\mathcal{R}_F$ for which F (p,q) is being computed are marked with "x"s, and the value of I^1 for the computation is indicated. Figure 5 shows the same thing for $J^1 = 28$.

Note how the values being used in the computation of F (4, 6)



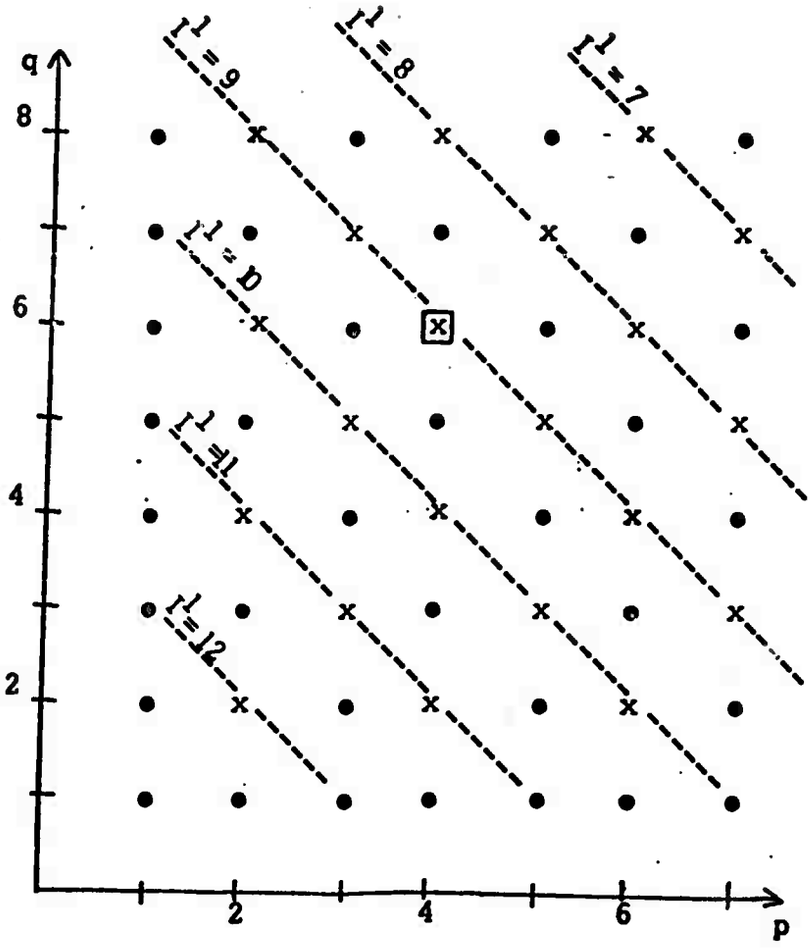
Computation of F(4, 6) for $I^1 = 9$

Figure 3



Execution for $J^1 = 27$

Figure 4



Execution for $J^1 = 28$

Figure 5

in Figure 5 were computed in Figure 4. A comparison with Figure 3 illustrates why this method of concurrent execution is equivalent to the original algorithm.

The saving in execution time achieved by the rewriting will depend upon the amount of overhead in the implementation of the DO CONC, as well as the actual number of processors available. (The sets in the DO CONC statement contain up to 252 elements.) However, the value of this approach is indicated by the fact that the number of sequential iterations has been reduced by a factor of over 135. (However, we must point out that a real program would probably include a convergence test within the outer DO I^1 loop, so the analysis could only be applied to the inner DO I^2/DO I^3 loop.)

IX. THE GENERAL PLANE THEOREM

We now generalize the Hyperplane Theorem to cover the case when some of the index variables $I^2, \ldots, I^n$ are missing. Concurrent execution is then possible for the points in $\mathcal{I}$ lying along parallel planes. Each missing variable may lower the dimension of the planes by one.

Plane Concurrency Theorem: Assume that at most $k-1$ of the index variables $I^2, \ldots, I^n$ are missing. Then loop (1) can be rewritten in the form of loop (4). Moreover, the mapping Π used for the rewriting can be chosen to be independent of the index set $\mathcal{I}$.

Proof: The proof is a generalization of the proof of the Hyperplane Theorem. Let $I^{j_2}, \ldots, I^{j_k}$ be the possibly missing variables among $I^2, \ldots, I^n$. Set $j_1 = 1$, $j_{k+1} = n+1$, and assume $j_1 < j_2 < \ldots < j_k < j_{k+1}$.

Let $\mathcal{P}$ be the set of all elements $X > 0$ of all the sets $\langle f, g \rangle$ referred to by rule C2. We must construct π so that $\pi(X) > 0$ for all $x \in \mathcal{P}$. Let $\mathcal{P}_j = \{(0, \ldots, 0, x^j, \ldots, x^n) \in \mathcal{P} : x^j > 0\}$, so $\mathcal{P}_j$ is the set of all elements of $\mathcal{P}$ whose $j^{\underline{th}}$ coordinate is the left-most non-zero one. Then every element of $\mathcal{P}$ is in one of the $\mathcal{P}_j$.

The mapping $\pi : \mathbb{Z}^n \to \mathbb{Z}^k$ will be constructed with

$$\pi(P) = (\pi^1(P), \ldots, \pi^k(P)),$$

where each $\pi^i : \mathbb{Z}^n \to \mathbb{Z}$ will be defined by

$$\pi^i [(I^1,\ldots,I^n)] = a^i_1 I^1 + \ldots + a^i_n I^n$$

for non-negative integers a^i_j. Moreover, we will have $a^i_j = 0$ if $j < j_i$ or $j \geq j_{i+1}$. This implies that is $X \in P_j$ and $j > J_{i+1}$. Then $\pi^i(X) = 0$. It therefore suffices to construct π^i so that for each j with $j_i \leq j < j_{i+1}$, and $X \in P_j$: $\pi^i(X) > 0$ - for we then have

$$\pi(X) = (0,\ldots,0, \pi^i(X),\ldots,\pi^k(X)) > 0.$$

Recall that for the sets <f,g>, an * can appear only in the $j_1,\ldots,j_k$ coordinates. Thus any element of any of the sets P_j with $j_i < j < j_{i+1}$ can be represented in the form

$$(0,\ldots,0, x_r^{j_i},\ldots,x_r^{j_{i+1}-1},\ldots) \text{ , or}$$

$$(0,\ldots,0, +, x_r^{j_{i+1}},\ldots, x_r^{j_{i+1}-1},\ldots)$$

for a finite collection of integers x^j_r, $j_i \leq j < j_{i+1}$. By the same argument used in the proof of the Hyperplane Theorem, we can replace "+" by $x_r^{j_i} = 1$, and choose $a^i_j \geq 0$, $j_i \leq j < j_{i+1}$ such that

$$a^i_{j_i} x_r^{j_i} + \ldots + a^i_{j_{i+1}-1} x^{j_{i+1}-1} > 0$$

for each r. Choosing $a^i_j = 0$ for $j < j_i$ and $j > j_{i+1}$ completes the construction of the required π^i.

The construction given in Appendix A is then applied to give invertable relations of the form

$$J^{j_i} = a^i_{j_i} I^{j_i} + \ldots + a^i_{j_{i+1}-1} I^{j_{i+1}-1}$$

$$J^j = \sum_{r=j_i}^{j_{i+1}-1} b^j_r I^r \qquad \text{for } j_i < j < j_{i+1}.$$

Combining these and reording the J^j gives the required mapping J. ∎

As in the hyperplane case, to get an optimal solution we want to minimize the number of iterations of the outer DO loops. This means minimizing $(\mu^1 - \lambda^1 + 1) \ldots (\mu^k - \lambda^k + 1)$. Using the notation of (5), it is easy to verify that this number is equal to

(11) $(M^1 \mid a_1^1 \mid + \ldots + M^n \mid a_n^1 \mid + 1) \ldots (M^1 \mid a_1^k \mid + \ldots + M^n \mid a_n^k \mid + 1)$,

where $M^i = \mu^i - \ell^i$.

Finding the a_j^i is now an integer programming problem. Note that a solution with $a_1^i = \ldots = a_n^i = 0$ for some i gives a solution to the rewriting problem with k replaced by k-1, since that π^i can be removed without affecting the constraint inequalities. The Plane Concurrency Theorem proves the existence of a π: $\mathbb{Z}^n \rightarrow \mathbb{Z}^k$ satisfying C2, for a particular value of k. It may be possible to find such a π for a smaller k.

For completeness, we will state a sufficient condition for the loop body to be concurrently executable for all points in $\mathcal{L}$. This is the case when the zero mapping ($\pi(P) = 0$ for all $P \in \mathbb{Z}^n$) satisfies C2. Since $\langle g, f \rangle = \{ -X : X \in \langle f, g \rangle \}$, it is clear that this is true if and only if all the sets $\langle f, g \rangle$ are equal to $\{0\}$. Finally, the rules for computing the sets $\langle f, g \rangle$ give the following rather obvious result:

<u>If none of the index variables are missing, and for any generated variable, all occurrences of that variable are identical, then loop (1) can be rewritten as a</u>

DO α CONC FOR ALL $(I^1, \ldots, I^n) \in \mathcal{I}$

<u>loop.</u>

The hypothesis means that in the expression (7) for any generated variable VAR, $r = n$ and the m^i are the same for all occurrences of VAR.

CHAPTER III

SYNCHRONOUS PROCESSORS

1. The DO SIM Statement

We now consider the case of completely synchronous processors, the primary example being the ILLIAC-IV. To accomodate it, let us introduce the DO SIM (for SIMultaneously) statement, having the following form:

DO α SIM FOR ALL $(I^1,\ldots,I^k) \in \mathcal{J}$,

where $\mathcal{J}$ is a subset of $\mathbb{Z}^k$. Its meaning is similar to that of the DO CONC statement, except that the computation is performed synchronously by the individual processors. Each point of $\mathcal{J}$ is assigned to a separate processor, and each statement in the range of the DO SIM is, in turn, simultaneously executed by all the processors. An assignment statement is executed by first computing the right-hand side, then simultaneously performing the assignment.

As an example, consider

```
      DO 15 SIM FOR ALL I ε { x : 2 ≤ x ≤ 10 }
14    A(I) = A(I-1) + B(I)
15    B(I) = A(I) ** 2
```

The right-hand side of statement 14 is executed for all I before the assignment of A(I) is made, and before statement 15 is executed. Therefore, if initially A(4) = 5 and B(5) = 2, then executing the loop sets A(5) = 7 and B(5) = 49.

Because of the simultaneity of execution of the body for the various points of $\mathcal{J}$, we cannot allow any conditional transfer of control in the loop body which depends upon the index variables. E. g., the statement

```
IF (A(I)) 3, 4, 5
```

may not appear in a "DO SIM FOR ALL I" loop.

For simplicity, assume that there is no transfer of control within the body of the loop, so the statements are always executed sequentially in the order in which they appear.

We will allow conditional assignment statements such as

```
IF (A(I).EQ. 0)  B(I) = 3
```

They are easily implemented on the ILLIAC-IV because of its ability to turn off individual processors.

The only other restriction to be made on the body of a DO SIM loop is that a generation may not reference the same array element for two different points in its index set $\mathcal{S}$. I.e., an assignment statement may not have two different processors simultaneously storing values into a single memory location. We do allow them to simultaneously load a value from a single memory location, so this restriction is not made for uses of a variable. *

---

* Simultaneous loads from a single memory location are implemented in the ILLIAC-IV by the ability of the central control unit to broadcast a value to all processors.

II. Rewriting the Loop

Now consider the problem of rewriting the given loop (1) in the form

(12) DO α $J^1 = \lambda^1, \mu^1$

$\vdots$

DO α $J^k = \lambda^k, \mu^k$

DO α SIM FOR ALL $(J^{k+1}, \ldots, J^n) \in \mathcal{J}_{J^1, \ldots, J^k}$

loop body

α CONTINUE

This is the same as loop (4), except the DO CONC is replaced by a DO SIM. We assume that the body of loop (1) contains no control transfering statements - i.e., no GO TO or numerical IF statements.

Define the mappings J and π as before. Any DO CONC statement can be executed as a DO SIM, since it must give the same result if the asynchronous processors happen to be synchronized. Thus, the rewriting could be done just as before by finding a π which satisfies C2. However, the synchrony of the computation will allow us to weaken the condition C2.

Recall that rule C1 was made so that the rewriting will preserve the order in which two different references are made to the same array element. For references made during two different executions of the loop body, the asynchrony of the processors requires that the order of those executions be preserved. However, with synchronous processors, we can allow the

two loop body executions to be done simultaneously if the references will then be made in the correct order. The order of these two references is determined by the positions within the loop body of the occurrences which do the referencing.

For two occurrences f and g, let $f \ll g$ denote that the execution of f precedes the execution of g within the loop body. This means that either the statement containing f precedes the statement containing g, or else that f is a use and g a generation in the same statement. The above observation allows us to change rule C1 to the following weaker condition on π:

For ... generation: if $T_f(P) = T_q(Q)$

for $P, Q \in \mathcal{I}$ with $P < Q$, then we must have either

(i) $\pi(P) < \pi(Q)$, or

(ii) $\pi(P) = \pi(Q)$ and $f \ll g$.

In this rule, either (i) or (ii) is sufficient to insure that occurrence f references $T_f(P)$ for the point $P \in \mathcal{I}$ before g references the same element for $Q \in \mathcal{I}$. The conditions can be rewritten in the following equivalent form:

(i) $\pi(P) \leq \pi(Q)$ and (ii) if $\pi(P) = \pi(2)$ then $f \ll g$.

In the same way c2 was obtained from c1, the above rule gives the following:

(S1) For every variable and every ordered pair of occurrences f, g of that variable, at least one of which is a generation: for every $\chi \in \langle f, g \rangle$ with $\chi > \vec{0}$, we must have:

(i) $\pi(X) \geq \vec{0}$, and

(ii) if $\pi(X) = \vec{0}$, than $f << g$.

If π satisfies rule S1, then it satisfies the preceding rule, so the rewritten loop (12) is equivalent to the original loop (1).

We have been assuming that in rewriting the loop body, the order of execution of the occurrences was not changed. I.e., the I^j were replaced by expressions involving $J^1, \ldots, J^n$, but nothing else was done to the loop body. Now let us consider changing the order of execution of the occurrences.* That is, we may change the position of occurrences within the loop body. For example, we may reorder the statements.

Let $f<<g$ mean that f is executed before g in the rewritten loop body (12). Then rule S1 guarantees that the correct temporal ordering of references is maintained when the references were made in the original loop during different executions of the loop body. Having changed the positions of occurrences in rewriting the loop body, we now have to make sure that any two references to the same array element made during a single execution of the loop body are still made in the correct order. The following analogue of rule C1 handles this:

For ... generation: if $T_f(P) = T_g(P)$ for some $P \in \mathcal{Y}'$, and f precedes g in the original loop body. Then $f << g$.

---

* Remember that there was no point in doing this before, since it couldn't help for asynchronous processors.

Rewriting this in terms of the sets $\langle f,g \rangle$ gives the following rule.

(S2) For every variable, and every ordered pair of occurrences f, g of that variable, at least one of which is a generation: if $\vec{0} \in \langle f, g \rangle$ and f precedes g in the original loop body, then $f \ll g$.

Rules S1 and S2 guarantee that the rewritten loop (12) is equivalent to the original loop (1). Note that rule S2 does not involve π.

III. The Coordinate Method

We could now try to solve the following problem: find a rewriting of the loop body (and the resulting << relation between occurrences) and a mapping π which satisfy rules S1 and S2, and which minimize the expression (11). This would give a rewriting of the loop which is optimal in the sense that the outer DO J^1 /.../ DO J^k loop has the fewest iterations. However, the optimality of such a rewriting is illusory, for reasons which we will now discuss.

The ILLIAC-IV has 64 processors. The feasibility of a machine with so many processors is achieved by having all processors operate synchronously with a single control unit, and by allowing each processor to access only its own separate portion of memory. If processor 12 wants to load a data word contained in processor 5's part of memory, then the following sequence of instructions is executed simultaneously by each processor number i, for i = 0 to 63:

(1) load

(2) transmit data word to processor i + 7 (mod 64).

This means that the method of storing arrays must depend upon how they are to be accessed. For example, consider the occurrence $F(J^1 - 2 * J^2 - J^3, J^3)$ inside the DO CONC FOR ALL (J^2, J^3), which we generated before with the Hyperplane Theorem. It necessitates a complicated, space-wasting format for storing the array F. The array would probably

have to be reformated before and after execution of the outer DO J^1 loop. *

It appears that the best results are obtained by choosing a mapping J which gives a loop with simple subscripting and a reasonable amount of simultaneous computation. An obvious way of choosing such a J is to let $J^1, \ldots, J^n$ be a permutation of the original index variables $I^1, \ldots, I^n$. More precisely, the mapping π: $\mathbb{Z}^n \to \mathbb{Z}^k$ is taken to be a coordinate projection - that is, a mapping for which $\pi[(a^1, \ldots, a^n)]$ is obtained by deleting n-k coordinates from $(a^1, \ldots, a^n)$.

For example, for n = 5 we may want to rewrite loop (1) as

DO α $I^3 = \ell^3, u^3$

DO α $I^4 = \ell^4, u^4$

DO α SIM FOR ALL $(I^1, I^2, I^5) \in \{(x, y, z)$:

$\ell^1 \leq x \leq u^1$, $\ell^2 \leq y \leq u^2$, and $\ell^5 \leq z \leq u^5\}$

:

Then $\pi[(I^1, I^2, I^3, I^4, I^5)] = (I^3, I^4)$ and $J[(I^1, \ldots, I^5)] = (I^3, I^4, I^1, I^2, I^5)$. Notice that if π is a corrdinate projection, then the sets $\mathcal{S}_{J^1, \ldots, J^k}$ of loop (11) are easy to compute.

The coordinate method consists of first choosing a coordinate projection π, and then trying to find a rewriting of the loop body for which S1 and S2 are satisfied. Since rewriting the loop body makes no difference

* A precise statement of the rules relating storage allocation and DO SIMs is contained in [4].

to condition (i) of S1, we must first require that it be satisfied for all relevant occurrences f, g. Next, we apply S1 and S2 to get certain ordering relations << between occurrences. We must then decide if it is possible to rewrite the loop body so that these relations are satisfied.

In order to make this decision, we need a trivial observation: a use in an assignment statement must precede the generation in that statement. This observation will be given the status of a rule.

(S3) For any use f and generation g in a single statement, we must have $f << g$.

Now we add the relations << given by S3 to those obtained from S1 and S2. Next, we add all relations implied by transitivity. I. e., whenever $f << g$ and $g << h$, we must add the relation $f << h$.* If the resulting ordering relations are consistent - that is, if we do not have $f << f$ for any occurrence f - then the loop body can be rewritten to satisfy the ordering relations. We will describe the method of rewriting the loop body by an example.

---

* An efficient algorithm for doing this is given by [5].

IV. An Example

Consider the following simple loop:

```
      DO 24 I1 = 2, 50
      DO 24 I2 = 1, 5
21    A(I1, I2) = B(I1, I2) + C(I1)
                  (a1)      (b1)    (c1)

22    C(I1) =  B(I1 -1, I2)
               (c2)      (b2)

23    B(I1, I2) = A(I1 + 1, I2) ** 2
                  (b3)        (a2)

24    CONTINUE
```

We want to rewrite it as a DO I^2/DO SIM FOR ALL ^{1} loop, so we apply the coordinate method with the coordinate projection π defined by $\pi[(I^1, I^2)] = I^2$. We proceed as follows. (The calculations for steps 1-3 are shown in Table 2.)

1. Compute the relevant sets $<f, g>$ for rules S1 and S2.
2. Check that S1 (i) is not violated.
3. Find the ordering relations given by S1(ii) and S2.
4. Apply S3 to get the following relations:

   statement 21: b1 $\ll$ a1
   c1 $\ll$ a1

   :

The Sets <f, g>	Is S1 (i) Violated ?	Ordering Relations S1 (ii)	S2
<a1, a1> = (0,0)	NO	-	-
<a1, a2> = (-1,0)	NO	-	-
<a2, a1> = (1, 0)	NO	a2<<a1	-
<b3, b3> = (0,0)	NO	-	-
<b1, b3> = (0,0)	NO	-	b1<<b3
<b3,b1> = (0,0)	NO	-	-
<b2,b3 > = (-1,0)	NO	-	-
<b3,b2> = (1,0)	NO	b3<<b2	-
<c1,c1> = (0,0)	NO	-	-
<c1, c2> = (0,*)	NO	-	c1<<c2
<c2, c1> = (0,*)	NO	-	-

Table 2

statement 22: $b2 \ll c2$

statement 23: $a2 \ll b3$

5. Find all relations implied by transitivity:

$b3 \ll c2$	[by $b3 \ll b2$ and $b2 \ll c2$]
$a2 \ll b2$	[by $a2 \ll b3$ and $b3 \ll b2$]
$b1 \ll b2$	[by $b1 \ll b3$ and $b3 \ll b2$]
$b1 \ll c2$	[by $b1 \ll b2$ and $b2 \ll c2$]
$a2 \ll c2$	[by $a2 \ll b3$ and $b3 \ll c2$]

6. Check that no relation of the form $f \ll f$ was found in 3 or 5.

7. Order the generations in any way which is consistent with the above relations - i.e., obeying $b3 \ll c2$. We let $a1 \ll b3 \ll c2$.

We then write

21 $A(I^1, I^2) =$

(a1)

23 $B(I^1, I^2) =$

(b3)

22 $C(I^1) =$

(c2)

8. Insert the uses in positions implied by the ordering relations (recall that $a2 \ll a1$):

$A(I^1 + 1, I^2)$

(a2)

21 $A(I^1, I^2) = B(I^1, I^2) + C(I^1)$

(a1) (b1) (c1)

23 $B(I^1, I^2) =$ **2

(b3)

22 $C(I^1) = B(I^1 - 1, I^2)$

(c2) (b2)

9. Add any extra variables necessitated by uses no longer appearing in their original statements:

$X(I^1 + 1, I^2) = A(I^1 + 1, I^2)$

(a2)

21 $A(I^1, I^2) = B(I^1, I^2) + C(I^1)$

(a1) (b1) (c1)

23 $B(I^1, I^2) = X(I^1 + 1, I^2)$ **2

(b3)

22 $C(I^1) = B(I^1 - 1, I^2)$

(c2) (b2)

This finally gives us the following rewriting of our original loop:

DO 24 $I^2 = 1, 5$

DO 24 SIM FOR ALL $I^1 \in \{x: 2 \leq x \leq 50\}$

$X(I^1 + 1, I^2) = A(I^1 + 1, I^2)$

21 $A(I^1, I^2) = B(I^1, I^2) + C(I^1)$

23 $B(I^1, I^2) = X(I^1 + 1, I^2)$ **2

22 $C(I^1) = B(I^1 - 1), I^2$

24 CONTINUE

V. Further Remarks

It is easy to deduce a general algorithm for the coordinate method from the preceeding example. The method can be extended to cover the case of an inconsistent ordering of the occurrences. In that case, the loop can be broken into a sequence of sub-loops. Every generation g for which the relation $g \ll g$ does not hold can be executed within a DO SIM loop. An algorithm for doing this is described in Chapter 4.

Observe that there are only 2^n-1 choices of a coordinate projection π for rewriting loop (1). It is easy to try them all, in decreasing order of the amount of parallel computation achieved, until one is found for which the rewriting is possible. Rule S1 should rapidly eliminate many choices.

It may happen that the rewriting cannot be done with any coordinate projection. In this case, a more general linear mapping π must be sought, using the approach developed before for DO CONC loops. For example, no coordinate projection works for the relaxation loop (10).

CHAPTER IV

PRACTICAL CONSIDERATIONS

I. Restrictions on the Loop

Now consider the application of these methods to the problem of compiling a FORTRAN program for execution on a multiprocessor computer. We immediately observe that the restrictions which have been placed on the loop (1) would eliminate most real Fortran DO loops from consideration. For example, the DO limits ℓ^1, μ^1 are usually not all constants known at compile time. Fortunately, most of the restrictions were made to simplify the exposition, and are not essential. We will now describe the restrictions which <u>are</u> essential to the analysis.

First, some terms must be defined. By a "loop constant", we mean an expression whose value does not change while the loop is executed - i.e., any expression not involving generated variables or loop index variables. A quantity is "known at compile time" if it has a constant value which can be determined by the FORTRAN compiler.

The analysis can be applied to the following loop:

(13)
$$\begin{aligned} &\text{DO } \alpha\, I^1 = \ell^1, \mu^1, d^1 \\ &\qquad \vdots \\ &\text{DO } \alpha\, I^n = \ell^n, \mu^n, d^n \end{aligned}$$

loop body

α CONTINUE

assuming that it satisfies the following conditions:

1. Each d^i is known at compile time.
2. The loop body contains no transfer of control to any statements outside it.
3. There is no I/O statement in the loop body.
4. For each subroutine or function call in the loop body, it is known which variable elements it can modify.
5. Each occurrence of a generated variable must be of the form VAR $(e^1, \ldots, e^m)$, with
$$e^i = a^i_1 * I^1 + \ldots + a^i_n * I^n + c^i ,$$
where c^i is a loop constant and each a^i_j is known at compile time.

For the coordinate method, the following additional assumptions are required.

6. There is no transfer of control within the loop body.
7. For every generated variable VAR, each occurrence of VAR within the loop body must be of the form
$$\text{VAR}\,(a^1 * I^{j_1} + c^1, \ldots, a^m * I^{j_m} + c^m),$$
where c^i is a loop constant, $a^i = \pm 1$ or 0, and the a^i and j_i are the same for all occurrences of VAR.

By weakening the restrictions, many complicated details are added to the process of rewriting the loop. However, the analysis remains largely unchanged. Some of these details are described in Chapter 4.

A significant change is introduced by allowing occurrence mappings of the form given in 5. It necessitates a complicated restating of the Hyperplane and Plane Concurrency Theorems, as well as changing the method

of choosing the mapping π. This will be discussed in a future paper.

Note that if the loop (13) satisfies these restrictions, then so does the inner DO I^k/.../ DO I^n loop, for any k. (The index variables I^j for $j < k$ are loop constants for the inner loop.)

II. Meeting the Restrictions

Even if a given loop does not satisfy the above restrictions, it may be possible to rewrite it so that it does. We will give some useful techniques for doing this.

It is easy to fulfill the requirement that the DO statements be tightly nested. The method is illustrated by the following example. The loop

```
      DO 77 I1 = 1, 10
16    A (I1, 1) = 0
      DO 77 I2 = 2, 20
        :
```

can be rewritten as the following tightly nested loop:

```
      DO 77 I1 = 1, 10
      DO 77 I2 = 2, 20
16    IF (I2 .EQ. 2) A(I1, I2-1) = 0
        :
```

This technique is referred to as quantifying statement 16. It may be possible to move the statement back outside the DO I^2 loop and unquantify it after the rewriting is performed.

Occurrence mappings can sometimes be rewritten by substituting for generated variables so that condition 5 is met. One trick is illustrated by the following example. Given

```
      K = N
      DO 6 I = 1, N
5     B(I) = A (K)
6     K = K - 1
```

We can rewrite it as

```
      DO 51 I = 1, N
5     B(I) = A (N + 1 - I)
51    CONTINUE
61    K = 1
```

This use of auxiliary variables to effect negative incrementing is fairly common in FORTRAN programs.

In condition 6, the real restriction is that there can be no possible loops inside the loop body. If this is the case, then transfer of control can easily be eliminated by quantifying assignment statements with logical IFs.

III. Scalar Variables

Even though the loop satisfies all the restrictions, it is clear that these methods can give no parallel computation if there are generated scalar variables. Any such variable must be eliminated.

A common situation is for the variable to be just a temporary storage word within a single execution of the loop body. The variable X in the following loop is an example

```
      DO 3 I = 1, 10
      X = SQRT (A(I))
      B(I) = X
3     C(I) = EXP (X)
```

In this loop, each occurrence of X can be replaced by XX(I), where XX is a new variable.

In general, we want to replace each occurrence of the scalar by $VAR(I^1, \ldots, I^n)$, for a new variable VAR.* A simple analysis of the loop body's flow path determines if this is possible.

Another common situation is when the variable X appears in the loop body only in the statement

$$X = X + \text{expression},$$

where the expression does not involve X. This statement just forms the sum of the expression for all points in the index set $\mathcal{I}$. We can replace it by

---

*After the rewriting, to save space we can lower the dimension of VAR by eliminating any subscript not containing a DO FOR ALL index variable.

the statement

$$VAR(I^1,\ldots,I^n) = \text{expression},$$

and add the following "statement" after the loop:

$$X = X + \sum_{(I^1,\ldots,I^n)\ \varepsilon\ \mathcal{L}} VAR(I^1,\ldots,I^n).$$

The sum can be executed in parallel with a special subroutine.

The same approach applies when the variable is used in a similar way to compute the maximum or minimum value of an expression for all points in $\mathcal{L}$.

IV. Conclusion

We have presented methods for obtaining parallel execution of a given DO loop. Many details and refinements were omitted for simplicity, but all the basic ideas have been included. Some of the methods are being implemented in the ILLIAC-IV FORTRAN compiler, as described in Chapter 4. Preliminary study indicates that they will yield parallel execution for a fairly large class of programs. This is true for other types of multiprocessor computers as well.

CHAPTER V

A PROTOTYPE PARALLEL ANALYZER

1. Introduction

1.1 Review

The previous semi-annual report described, in a somewhat rigorous fashion, techniques for the detection of parallelism in FORTRAN DO loops. It was indicated in that report that this procedure can be divided into three steps:

a) Rewriting part of the program as a sequence of DO loops in a form suitable for analysis (Setup step).

b) Detecting computations in the loops which can be efficiently performed in parallel by ILLIAC (Analysis step).

c) Rewriting the loops using the Extended FORTRAN DO FOR ALL statement to explicitly describe the parallelism (Synthesis step).

The intersection of the above Setup, Analysis, and Synthesis steps is not empty; i.e., there is overlap.

Two methods for accomplishing the above, i.e.,

a) Coordinate Method

b) Hyperplane Method

were described in the previous semi-annual report.

1.2 Summary

Discussions of the Parallel Analyzer contained in this report will be oriented toward implementation activities accomplished during the past half year and those planned for the forthcoming months.

A preliminary implementation of the basic Parallel Analyzer components has been completed and debugged. This is not a functional Parallel Analyzer since it is missing many parts. The most important omission is a Synthesis Step (see 1.2). Extant algorithms are oriented toward the Coordinate Method with minimal attention given to the Hyperplane Method.

Algorithms included in the preliminary system relate to:

a) The locating, tight nesting, and rewriting of DO nests.

b) The collecting and validating of all array references and the permutation of the subscripts.

c) The computing of upper and lower bounds for DO variables when limits are not known at compile time.

d) The computing of data dependency relations between array uses and occurrences (i.e., the <f,g> sets described in the last semi-annual report).

e) The collecting of all possible DO SIM variables sets and the elimination of those sets containing obviously bad DO SIM variables in any combination.

f) The backing up of the current statement macro position to reduce the number of DO variables in a nest, thus perhaps eliminating violations which occurred with a larger DO variable set.

g) General utility, including routines for character string manipulation, Boolean matrix manipulation, linear expression manipulation, intermediate language generation, and debugging assistance.

Section 2 describes the present environment and includes a discussion of (g). Section 3 describes presently implemented algorithms for the Setup Step including (a) and (f) above. Section 4 describes presently implemented algorithms for the Analysis Step, including (b), (c), (d), (e) above.

Section 5 describes projected implementation work for the months ahead. The next version of the Parallel Analyzer is scheduled for April 30, 1972, completion.

2. Environment

The preliminary Parallel Analyzer and its environmental routines function as a computer program which is independent of any other phase of the ILLIAC FORTRAN compiler. Ultimately the input to the Parallel Analyzer will be the output of the Parse.

Parts of this environment are temporary. However, many parts (including most of the utilities described below) are permanent.

2.1 Generating the Input

An intermediate language tables generator which produces the necessary input to the Parallel Analyzer is included. Intermediate language tables are produced from declarative and imperative statements expressed in a rather rudimentary language which looks like an Assembly language. All statement operators are permitted but only certain ones are meaningful in the present environment. Integer is the only permissible data type. However, extensive facilities for describing arrays, array references, and subscripts (the crux of Parallel Analysis) is available.

2.2 Debugging Aids

The present system contains facilities for obtaining a formatted dump of any significant Parallel Analyzer table via directives from the teletype.

The intermediate language tables can be collected together and dumped, also, upon request.

2.3 Manipulation of Linear Forms

A collection of utility routines for the manipulation of expressions of the form

$$c_o + \sum_{i=1}^{n} c_i x_i,$$

where c_o and the c_i are integers and the x_i are scalars, is included.

Such an expression can be:

a) substituted for any variable in any other expression of the same form,

b) compared for equality with other expressions of the same form,

c) added (subtracted) to (from) another expression of the same form, and

d) multiplied by an integer constant.

These linear manipulation functions are required in the computation of <f,g> sets and upper bounds and are utilized by the re-writing and re-formatting algorithms of both the Basic Coordinate and Hyperplane Methods.

2.4 Simple Intermediate Language Manipulation

Routines for deleting, duplicating, and creating symbols, constants, labels, expression level macros, statement level macros, and arbitrary operands are included. These basic functions are required for any inspection and alteration of computation trees.

2.5 Boolean Matrix Manipulation

A package for the manipulation of n x n bit matrices is included. In addition to the obvious simple functions, routines for computing the transpose and computing the transitive closure are included. Routines of this sort are required to record program flow, detect cycles, etc.

2.6 String Handling Package

Because of the lack of anything resembling a proper handling of characters within the FORTRAN language, a standard string manipulation package is included. Functions of the package include: scanning, decimal and octal conversion, concatenation, insertion, extraction, comparison, and substitution. This package contains the tools necessary for intelligent interaction with the user and for the generation of reports. These are functions the Parallel Analyzer is expected to ultimately perform.

3. Setup Step

This section discusses the currently implemented algorithms dealing with the 'Setup Step' of the overall Parallel Detection process. (The intermediate language generator will have produced intermediate language tables prior to entry to the Setup algorithms.) Beginning at the current position indicated by the statement macro pointer, the computation table is scanned for DO operators. When the first, if any, is encountered, a flag is set to reflect that DO body statements are being processed. Scanning continues until the level of DO's reaches zero (i.e., the number of DO labels equals the number of DO statements). This procedure suggests that as many DO variables as possible are attached to the nest. However, if subsequent analysis fails, backup procedures to reduce the number of DO variables are invoked. Analysis then proceeds over the same nest but with fewer DO variables.

The present implementation rejects any statement in the DO body statements which is not a substitution statement or a compound statement consisting of a Logical If followed by a substitution. This indicates that the present implementation permits no:

a) control statements (e.g., GOTO, Arithmetic IF),
b) subroutine CALL statements, nor
c) input/output statements.

Furthermore, function calls are outlawed. These restrictions will, of course, be relaxed in subsequent implementation versions.

Algorithms for tight nesting are invoked if all permissible non-DO statements encountered do not follow the innermost DO operator and precede the first DO label. This procedure is best described by an example. Let:

```
      DO 10 I = 3,NI
      A(I) = 2
      DO 20 J = 1,NJ
20    B(I,J) = C(I,J)
10    IF(D(I)) E(I) = F(I)
```

be the original statements. The tight nesting algorithms of the Setup Step would make the above statements appear as:

```
      DO 30 I = 3,NI
      DO 30 J = 1,NJ
      IF(J.EQ.1) A(I) = 2
      B(I,J) = C(I,J)
      IF(D(I).AND.J.EQ.NJ) E(I) = F(I)
30    CONTINUE
```

Present Setup algorithms do not look for scalars, sums, etc., nor completely validate the legality of the tightly nested code, for all cases. This was deliberately omitted, but subsequent implementations will be extended in this direction.

The DO statements themselves are 'analyzed' during the Setup phase as follows:

a) The DO variable increment must be a positive or negative integer.

b) The DO variable lower limit or upper limit must be expressible as follows:

$$k + \sum_{i=1}^{m} k_i x_i + \sum_{j=1}^{n} c_j v_j$$

where k, k_i, c_j are integers, the x_i are scalars and are not DO variables for the present DO body, and the v_j are DO variables but not the present DO variable nor any DO variable interior to it.

If successful, the Setup algorithms create (or partially create) certain tables, which, together with updated intermediate language information, serve as input to the Analysis Step.

One of the outputs is an n x n precedence matrix which records the original flow of the loop body where n is the number of statements in the loop body.

4. Analysis Step

This section discusses the algorithms (presently implemented) which deal with the 'Analysis Step' of the overall Parallel Detection process.

4.1 Background

The present Analysis (based on the Coordinate Method) generates candidate DO SIM sets and eliminates obvious bad sets. An exhaustive analysis of the DO SIM sets to determine the 'degrees of goodness' of the non-bad sets is made.

Activities in the Analysis Step (as well as the predecessor Setup step) occur on a local basis only. Global considerations, for the present implementation, are not considered. Global considerations relate to issues such as:

a) arrays in COMMON,
b) equivalenced arrays,
c) external subroutines and functions,
d) formal parameters (arrays) received and passed, and
e) the effect the rewriting of one loop within a program unit has upon the parallelization of another loop within the same program unit.

4.2.1 What Exists

The Setup Step will have located a DO nest, extracted the DO statements, tightly nested the DO body and recorded this information in Parallel Analyzer tables and in upgraded intermediate language tables.

The first step performed here is a tree walk of each statement in the loop body which searches for array references (generations and uses). A generation is an array reference to the left of the substitution operation; any other appearance is a use.

A table of arrays and array elements is built. Subscripts are permuted to conform to the order of the DO variables. Rejects occur if references to the same array have different permutations. Subscript expressions are checked to see that they do not violate the special linear form required by subsequent analysis.

The components of each DO statement (i.e., the lower limit, upper limit, increment) are then analyzed. If all these quantities are known at compile time, this analysis is simple. If not, upper bounds on the limits are computed as a function of the

a) non-integer limit expressions, and the
b) dimension information from the symbol table.

A convenient, ordered list of all possible DO SIM sets is then constructed. Because of previous analysis some of these sets can at this point be declared 'bad', i.e., totally unsuited for parallelism.

A routine is then invoked to compute data dependency information -- in effect the <f,g> sets described in the last semi-annual report. This information is recorded for subsequent analysis.

Portions of the basic Coordinate algorithm, operating over the data dependency <f,g> sets, is then invoked. Information is recorded in tables (in particular the candidate for DO SIM table) declaring a set as bad, good, or questionable. If good or questionable, data is accumulated for subsequent analysis and selection.

It should be pointed out that the analysis to date has been directed towards the Coordinate Method as opposed to the Hyperplane Method. However, all Hyperplane Method considerations were not eliminated when implementing the present version.

5. The Next Version

The next version of the Parallel Analyzer is scheduled for April 30, 1972, completion. This section discusses what will be appended to the current Parallel Analyzer to produce this next version.

5.1 General

Minor changes will be made to the tight nesting and some handling of scalars will be done (see Setup extensions below).

Exhaustive analysis to select proper DO SIM variables will be performed. Data and statistics leading to acceptance or rejection will be studied. (See Analysis below.) Selected loops will be rewritten using IVTRAN DO FOR ALL statements.

The Parallel Analyzer will continue to function as a computer program independent of the rest of the ILLIAC compiler. Output reports will be improved.

No new implementation work will be performed in the following areas:

a) Hyperplane Method
b) Inconsistent Orderings (cycles within DO body)
c) Transfer Outside Loop
d) Backward Transfer
e) Creation of DO loops from non-explicit DO statements
f) Forward Transfers
g) Subroutine Calls

5.2 Setup Step Extensions

Some enhancements will be made to the tight nesting procedure. This procedure will be interlaced with new procedures for handling generated scalars.

When resolving unwanted scalars, certain common special cases will be looked for first. In particular, is the scalar a summing variable? Arrays should be introduced to eliminate storing into scalars within the body of the loop.

5.3 Analysis Step Extensions

The candidate for DO SIM table is passed over and updated in the following fashion. Sets declared by previous analysis as bad are avoided. A candidate set, say Π, is selected for examination. An m x m precedence bit matrix P for Π based on the data dependency relation of the <f,g> sets and the original order of the statements is constructed where m is the number of array occurrences.

If any intra-statement dependencies are unveiled from inspection of P, Π is marked as bad and a new Π is selected. The transitive closure of P is computed and stored in Q. The number of cycles within Q is counted and recorded. If there exist any cycles, the inconsistent ordering algorithms must be invoked if this DO SIM set is ultimately selected as the best. The inconsistent ordering routines (as pointed out earlier) will not be available in the next version.

Simple and nice (indeed perhaps the most common in real programs) situations are looked for and special attention is given to them in an effort to further exploit the parallelism. Such situations include:

a) only one DO variable in the DO SIM set,
b) no data dependency sets, i.e., no <f,g> sets,
c) <f,g> = null for all sets,
d) <f,g> = 0 for all sets, etc.

A linear ordering for the statements is constructed. The number of uses which have to be moved is tallied and recorded. The above computations are saved in case this Π is the one ultimately chosen. The procedure then cycles and selects another Π, if any, from the candidate for DO SIM tables.

Presented with possibly n different DO SIM sets, algorithms will be needed to select the 'best' on the basis of previously recorded data. The data available for inspection at this point should serve as a heuristic in the creation of the final algorithm. Data available will ultimately concern:

a) cycles and structure,
b) number of moves of array uses,
c) frequency
d) user declared information ('control card' information, interactive acquisition),
e) allocation checks, etc.

A DO SIM set, say α is chosen. If α contains cycles, algorithms for handling inconsistent orderings should be invoked. If data computed above (i.e., precedence matrix P, transitive closure, linear ordering) for α has been dumped, it should be restored at this point in preparation for what is called the 'Synthesis Step'.

5.4 Synthesis

After selecting a suitable DO SIM set α, the final task is the rewriting of the loop. The task of rearranging, adding, regenerating, and rewriting statements has been simplified because of the rich collection of utility routines (2.3, 2.4) available.

Further analysis may have to be imbedded in the Synthesis algorithms: for example, some time measure of the number of new instructions added to the new loop, etc. It would be possible (with proper frequency information) to predict this in the Analysis proper section. A more accurate measure, however, may be desired.

If failure or rejection occurs anywhere along the line, the intermediate language tables can be restored to appear as if no Parallel Analysis attempts had occurred.

5.5 Conclusion

A preliminary implementation of the Parallel Analyzer has been completed. This version isolates DO loop ranges and performs early analysis. The next implementation will continue with further analysis and synthesis patterned after the Basic Coordinate Method. Activities will remain at a local level with global considerations deferred.