Programming Techniques
G. Manacher Editor

The Parallel Execution of DO Loops

Leslie Lamport
Massachusetts Computer Associates, Inc.

Methods are developed for the parallel execution of different iterations of a DO loop. Both asynchronous multiprocessor computers and array computers are considered. Practical application to the design of compilers for such computers is discussed.

Key Words and Phrases: parallel computing, multiprocessor computers, array computers, vector computers, loops

CR Categories: 4.12, 5.24

Introduction

Any program using a significant amount of computer time spends most of that time executing one or more loops. For a large class of programs, these loops can be represented as FORTRAN *DO* loops. We consider methods of executing these loops on a multiprocessor computer, in which different processors independently execute different iterations of the loop at the same time.

This approach was inspired by the ILLIAC IV since it is the only type of parallel computation which that computer can perform [1]. However, even for a computer with independent processors, it is inherently more efficient than the usual approach of having the processors work together on a single iteration of the loop. This is because it requires much less communication between individual processors.

The methods presented are, of course, independent of the syntax of FORTRAN. The basic feature of the FORTRAN *DO* loop which is used is that the range of values assumed by the index variable is known upon entry to the loop. Thus, most but not all ALGOL *for* loops can be handled.

The analysis is performed from the standpoint of a compiler for a multiprocessor computer. Two general methods are described. The *hyperplane method* is applicable to both multiple instruction stream computers and single instruction stream computers such as the ILLIAC IV, the CDC STAR-100 and the Texas Instruments ASC. The *coordinate method* is applicable to single instruction stream computers. Both methods translate a nest of *DO* loops into a form explicitly indicating the parallel execution. The *DO* loops may be of a fairly general nature. The major restrictions are that the loop body contain no I/O and no transfer of control to any statement outside the loop.

These methods are basically quite simple, and can drastically reduce the execution time of the loop on a parallel computer. They are currently being implemented in the ILLIAC IV FORTRAN compiler. Preliminary results indicate that they will yield parallel execution for a fairly large class of programs.

The two methods are described separately in the following two sections. The final section discusses some practical considerations for their implementation.

I. The Hyperplane Method

Example. To illustrate the hyperplane method, we consider the following loop.

```
      DO 99 I = 1, L
      DO 99 J = 2, M
      DO 99 K = 2, N
      U(J,K) = (U(J+1,K) + U(J,K+1)
        (u1)       (u2)       (u3)
             + U(J-1,K) + U(J,K-1)) * .25
               (u4)       (u5)
   99 CONTINUE                              (1)
```

(For future reference, we have assigned a name to each occurrence of the variable U, and written it in a circle beneath the occurrence.) This is a simplified version of a standard relaxation computation.

The loop body is executed $L(M-1)(N-1)$ times—once for each point (I,J,K) in the *index set* $\mathcal{I} = \{(i,j,k) : 1 \leq i \leq L, 2 \leq j \leq M, 2 \leq k \leq N\}$. We want to speed up the computation by performing some of these executions concurrently, using multiple processors. Of course, this must be done in such a way as to produce the same results as the given loop.

The obvious approach is to expand the loop into the $L(M-1)(N-1)$ statements

```
U(2,2) = ···
U(2,3) = ···
   ⋮
```

and then apply the techniques described in [2]. This is at

This research was supported by the Advanced Research Projects Agency of the Department of Defense and was monitored by Army Research Office-Durham under Contract No. DAHC04-70-C-0023. Author's address: Massachusetts Computer Associates, Inc., Lakeside Office Park, Wakefield, MA 01880.

best a formidable task. It is impossible if L, M, and N are not all known at compile time.

Our approach is to try to execute the loop body concurrently for all points (I,J,K) in $\mathcal{I}$ lying along a plane. In particular, the hyperplane method will find that the body of loop (1) can be executed concurrently for all points (I,J,K) lying in the plane defined by $2I + J + K =$ constant. The constant is incremented after each execution, until the loop body has been executed for all points in $\mathcal{I}$.

To describe this more precisely, we need a means of expressing concurrent computation. We use the statement

DO 99 *CONC FOR ALL* $(\bar{J},\bar{K}) \in S$

where S is a finite set of pairs of integers.[1] It has the following meaning: Assign a separate processor to each element of S. For each $(j,k) \in S$, the processor assigned to (j,k) is to set $\bar{J} = j$, $\bar{K} = k$ and execute the statements following the *DO CONC* statement through statement 99. All processors are to run concurrently, completely independent of one another. No synchronization is assumed. Execution is complete when all processors have executed statement 99.

Given loop (1), the hyperplane method chooses new index variables $\bar{I}$, $\bar{J}$, $\bar{K}$ related to I, J, K by

$$\begin{aligned} \bar{I} &= 2I + J + K \\ \bar{J} &= I \\ \bar{K} &= K \end{aligned} \tag{2}$$

and the inverse relations

$$\begin{aligned} I &= \bar{J} \\ J &= \bar{I} - 2\bar{J} - \bar{K} \\ K &= \bar{K} \ . \end{aligned} \tag{2'}$$

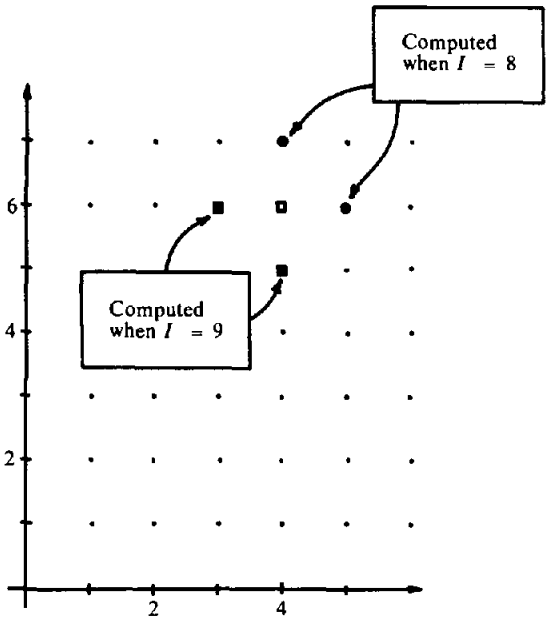
Loop (1) is then rewritten as

```
DO 99 Ī = 6, 2*L + M + N
DO 99 CONC FOR ALL (J̄,K̄) ∈ {(j, k) :
  1 ≤ j ≤ L, 2 ≤ Ī − 2j − k ≤ M and
    2 ≤ k ≤ N}
U(Ī−2*J̄−K̄,K̄) = (U(Ī−2*J̄−K̄+1,K̄)
  + U(Ī−2*J̄−K̄,K̄+1) + U(Ī−2*J̄
    −K̄−1,K̄) + U(Ī−2*J̄−K̄,K̄−1))
      * .25
99 CONTINUE                          (3)
```

Using relations (2) and (2'), the reader can check that loop (3) performs the same $L(M-1)(N-1)$ loop body executions as loop (1), except in a different order. To see why both loops give the same results, consider the computation of $U(4,6)$ in the execution of the original loop body for the element $(9,4,6) \in \mathcal{I}$. It is set equal to the average of its four neighboring array elements: $U(5,6)$, $U(4,7)$, $U(3,6)$, $U(4,5)$. The values of $U(5,6)$ and $U(4,7)$ were calculated during the execution of the loop body for (8,5,6) and (8,4,7), respectively,

[1] We remind the reader that a set is an *unordered* collection of elements. We will not bother to define a syntax for expressing sets, but will use the customary informal mathematical notation.

Fig. 1. Computation of U(4,6) for $\bar{I} = 9$.



i.e. during the previous execution of the *DO I* loop, with $I = 8$. The values of $U(3,6)$ and $U(4,5)$ were calculated during the current execution of the outer *DO I* loop, with $I = 9$. This is shown in Figure 1.

Now consider loop (3). At any time during its execution, $U(p,q)$ is being computed concurrently for up to half the elements of the array U. These computations involve many different values of I. Figure 2 illustrates the execution of the *DO CONC* for $\bar{I} = 27$. The points (p,q) for which $U(p,q)$ is being computed are marked with "x"s, and the value of I for the computation is indicated. Figure 3 shows the same thing for $\bar{I} = 28$.

Note how the values being used in the computation of $U(4,6)$ in Figure 3 were computed in Figure 2. A comparison with Figure 1 illustrates why this method of concurrent execution is equivalent to the algorithm specified by loop (1).

The rewriting has reduced the number of sequential iterations from $L(M-1)(N-1)$ to $2L + M + N - 5$. This gives the possibility of an enormous reduction in execution time. The actual saving in execution time will depend upon the overhead in executing the *DO CONC*, as well as the actual number of processors available. The *DO CONC* set contains up to $(M-1)(N-1)/2$ points. Since individual executions may be asynchronous, the *DO CONC* is easily implemented with fewer processors.

We must point out that a real program would probably have a loop terminated by a convergence test in place of the outer *DO I* loop. The hyperplane method

Fig. 2. Execution for $\bar{I} = 27$.

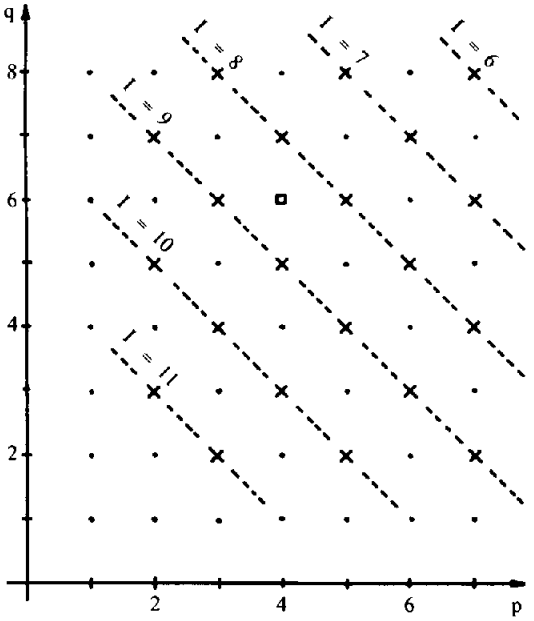
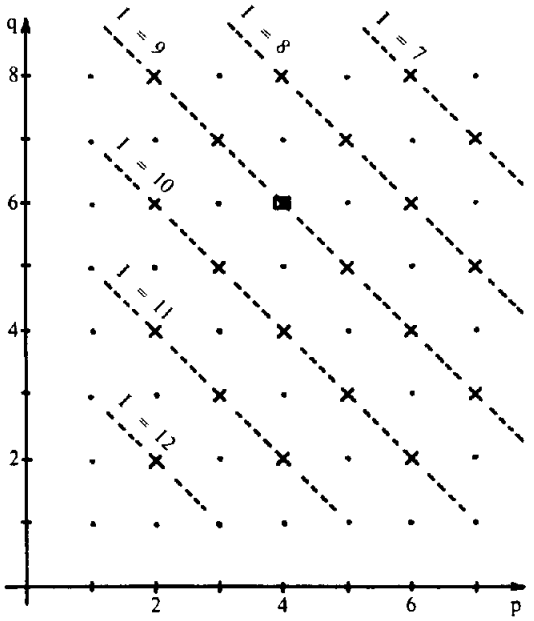


Fig. 3. Execution for $\bar{I} = 28$.



could then only be applied to the *DO J/DO K* loop. The reader can check that applying the general method described below to this loop reduces the number of sequential iterations from $(M-1)(N-1)$ to $M+N-3$ —still a significant reduction.

Notations and Assumptions

To describe the general methods, we introduce some notation. We consider loops of the following form:

$$\begin{array}{l} DO \quad I^1 = l^1, u^1 \\ \quad\quad \vdots \\ DO \quad I^n = l^n, u^n \\ \quad\quad \boxed{\text{loop body}} \\ CONTINUE \end{array} \qquad (4)$$

where l^i and u^i may be any integer-valued expressions, possibly involving $I^1, \ldots, I^{i-1}$. (Our use of superscripts and subscripts is in accord with the usual notation of tensor algebra.) We could allow arbitrary constant *DO* increments, but this would add many complicated details.

A variable which appears on the left-hand side of an assignment statement in the loop body is called a *generated* variable.

We make the following assumptions about the loop body.

(A1) It contains no I/O statement.

(A2) It contains no transfer of control to any statement outside the loop.

(A3) It contains no subroutine or function call which can modify data.

(A4) Any occurrence in the loop body of a generated variable VAR is of the form $VAR(e^1, \ldots, e^r)$, where each e^i is an expression not containing any generated variable.

Assumption (A3) could be replaced by the assumption that we know which data can be modified by a subroutine or function call. However, this would complicate the discussion. Assumption (A4) must be strengthened to assure that the hyperplane method will work. This will be done below.

We let $\mathbf{Z}$ denote the set of all integers, and $\mathbf{Z}^n$ denote the set of n-tuples of integers. For completeness, we define $\mathbf{Z}^0 = \{0\}$. The *index set* $\mathcal{I}$ of loop (4) is defined to be the subset of $\mathbf{Z}^n$ consisting of all values assumed by $(I^1, \ldots, I^n)$ during execution of the loop, so

$$\mathcal{I} = \{(i^1, \ldots, i^n) : l^1 \leq i^1 \leq u^1, \cdots\}.$$

Note that $\mathcal{I}$ may not be known at compile time. The element $(i^1, \ldots, i^n)$ of $\mathcal{I}$ represents the execution of the loop body for $I^1 = i^1, \ldots, I^n = i^n$.

We order the elements of $\mathbf{Z}^n$ lexicographically in the usual manner, with $(2,9,13) < (3,-1,10) < (3,0,0)$. For any elements P and Q of $\mathcal{I}$, the loop body is executed for P before it is executed for Q if and only if $P < Q$.

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

Rewriting the Loop

To generalize the rewriting procedure used in our example, loop (4) will be rewritten in the form

$$
\begin{array}{ll}
 & DO\ \alpha\ J^1 = \lambda^1, \mu^1 \\
 & \quad \vdots \\
 & DO\ \alpha\ J^k = \lambda^k, \mu^k \\
 & DO\ \alpha\ CONC\ FOR\ ALL \\
 & \quad (J^{k+1}, \ldots, J^n) \in S_{J^1,\ldots,J^k} \\
 & \qquad \boxed{\text{loop body}} \\
\alpha & CONTINUE
\end{array} \tag{5}
$$

where $S_{J^1,\ldots,J^k}$ is a subset of $\mathbf{Z}^{n-k}$ which may depend upon the values of $J^1, \ldots, J^k$.

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

$$
\begin{aligned}
J[(I^1, \ldots, I^n)] &= \left(\sum_{j=1}^{n} a_j^1 I^j, \cdots, \sum_{j=1}^{n} a_j^n I^j\right) \\
&= (J^1, \ldots, J^n)
\end{aligned} \tag{6}
$$

for integers a_j^i.[2] We then choose the λ^i, μ^i and $S_{J^1,\ldots,J^k}$ so that the index set $\mathcal{I}$ of loop (5) equals $J(\mathcal{I})$, and write the body of loop (5) so that its execution for the point $J(P) \in \mathcal{I}$ is equivalent to the execution of the body of loop (4) for $P \in \mathcal{I}$.

Define the mapping $\pi : \mathbf{Z}^n \to \mathbf{Z}^k$ 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 $J(P)$, $J(Q) \in \mathcal{I}$, the execution of the body of loop (5) for $J(P)$ precedes the execution for $J(Q)$ if and only if $\pi(P) < \pi(Q)$. If we consider loop (5) to be a reordering of the execution of loop (4), this statement is equivalent to the following.

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

The loop body is executed concurrently for all elements of $\mathcal{I}$ lying on a set of the form $\{P : \pi(P) = \text{constant} \in \mathbf{Z}^k\}$. Since J is assumed to be a one-to-one linear mapping, these sets are parallel $(n-k)$-dimensional planes in $\mathbf{Z}^n$.[3] We thus have concurrent execution of the loop body along $(n-k)$-dimensional planes through the index set. For $k = 1$, these are hyperplanes. We use the name "hyperplane method" to also include the case $k > 1$.

In our example, we had $n = 3$ and $k = 1$. The mapping $J : \mathbf{Z}^3 \to \mathbf{Z}^3$ is defined by $J[(i,j,k)] = (2i+j+k, i, k)$, and $\pi : \mathbf{Z}^3 \to \mathbf{Z}$ is defined by $\pi[(i,j,k)] = 2i + j + k$.

The general problem is to find a mapping J for which loop (5) gives an algorithm equivalent to that of loop (4). By requiring that J be a *linear* mapping, we have greatly restricted the class of mappings which are to be examined. It is this restriction which makes the analysis feasible.

[2] J is one-to-one if and only if (6) can be solved to write the I^j as linear expressions in the J^i with integer coefficients.

[3] We consider $\mathbf{Z}^n$ to be a subset of ordinary Euclidean n-space in the obvious way.

Basic Considerations

Let VAR be a program array variable. An *occurrence* of VAR is any appearance of it in the loop body. If it appears on the left-hand side of an assignment statement, the occurrence is called a *generation*; otherwise, it is called a *use*. Thus, generations modify the values of elements of the array, and uses do not.

Consider the use $u2$ of the variable U in loop (1). During execution of the loop body for $(i,j,k) \in \mathcal{I}$, it references the array element $U(j+1,k)$. We define the *occurrence mapping* $T_{u2} : \mathcal{I} \to \mathbf{Z}^2$ by $T_{u2}[(i,j,k)] = (j+1,k)$. Similarly, if f is an occurrence of an r-dimensional generated variable VAR in loop (4), then the occurrence mapping $T_f : \mathcal{I} \to \mathbf{Z}^r$ is defined so that f references the $T_f(P)$ element of VAR during execution of the loop body for $P \in \mathcal{I}$. Assumption (A4) guarantees that this is a reasonable definition.

We are looking for a condition to assure that the rewritten loop (5) is equivalent to the given loop (4). From our example, we can see that the significant consideration is the sequence of references to array elements. In loop (1), a value for $U(5,6)$ is generated by $u1$ during execution of the loop body for $(8,5,6) \in \mathcal{I}$. This value is used by $u2$ during the execution for (9,4,6). Therefore, when we change the order of executions in the rewriting, we must still have the execution for (8,5,6) precede the execution for (9,4,6). By statement (E) above, this means that π must satisfy $\pi[(8,5,6)] < \pi[(9,4,6)]$. Indeed, for our particular choice of π we have $\pi[(8,5,6)] = 27 < \pi[(9,4,6)] = 28$.

In general, let VAR be any variable. If a generation and a use of VAR both reference the same array element 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 $P, Q \in \mathcal{I}$, then: (i) if $P < Q$, we must have $\pi(P) < \pi(Q)$; and (ii) if $Q < P$, we must have $\pi(Q) < \pi(P)$. In the above example, $T_{u1}[(8,5,6)] = T_{u2}[(9,4,6)] = (5,6)$, and $(8,5,6) < (9,4,6)$, so we must have $\pi[(8,5,6)] < \pi[(9,4,6)]$. 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.

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. 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 $P, Q \in \mathcal{I}$ 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.

Table I.

Sets	Elements $> \mathbf{0}$	Constraints
$\langle u1,u1\rangle = (*,0,0)$	$(+,0,0)$	$a_1 > 0$
$\langle u1,u2\rangle = (*,-1,0)$	$(+,-1,0)$	$a_1 - a_2 > 0$
$\langle u2,u1\rangle = (*,1,0)$	$(+,1,0)$	$a_1 + a_2 > 0$
	$(0,1,0)$	$a_2 > 0$
$\langle u1,u3\rangle = (*,0,-1)$	$(+,0,-1)$	$a_1 - a_3 > 0$
$\langle u3,u1\rangle = (*,0,1)$	$(+,0,1)$	$a_1 + a_3 > 0$
	$(0,0,1)$	$a_3 > 0$
$\langle u1,u4\rangle = (*,1,0)$	same as $\langle u2,u1\rangle$	
$\langle u4,u1\rangle = (*,-1,0)$	same as $\langle u1,u2\rangle$	
$\langle u1,u5\rangle = (*,0,1)$	same as $\langle u3,u1\rangle$	
$\langle u5,u1\rangle = (*,0,-1)$	same as $\langle u1,u3\rangle$	

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 premature exit from the loop, therefore imply that rule (C1) gives a sufficient condition for loop (5) to be equivalent to loop (4). For most loops, (C1) is also a necessary condition.

The Sets $\langle f, g\rangle$

The trouble with rule (C1) is that it requires us to consider many pairs of points P,Q in $\mathcal{I}$. For the loop (1), there are $(L-1)\ (M-1)\ (N-1)$ pairs of elements $P,Q \in \mathcal{I}$ with $T_{u1}(P) = T_{u2}(Q)$ and $P < Q$. However, $T_{u1}(P) = T_{u2}(Q)$ only if $Q = P + (*,-1,0)$, where $*$ denotes any integer. We would like to be able to work with the single descriptor $(*,-1,\ 0)$ rather than all the pairs P,Q.

This suggests the following definition. For any occurrence f,g of a generated variable in loop (4), define the subset $\langle f, g\rangle$ of $\mathbf{Z}^n$ by $\langle f, g\rangle = \{X : T_f(P) = T_g(P+X)$ for some $P \in \mathbf{Z}^n\}$. Observe that $\langle f,g\rangle$ is independent of the index set $\mathcal{I}$. In our example, $\langle u1,\ u2\rangle = \{(x,-1,0) : x \in \mathbf{Z}\}$, and we denote this set by $(*,-1,0)$. The other sets $\langle f,g\rangle$ of loop (1) which we will use are listed in Table I.

We now rewrite rule (C1) in terms of the sets $\langle f,g\rangle$. 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 (6).) Also, remember that $A < A + B$ if and only if $B > \mathbf{0}$. Then just substituting $P + X$ for Q in rule (C1) yields this rule.

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

Removing the clause "for $P,P + X \in \mathcal{I}$" from (C1′) gives a stronger condition for π to satisfy. Doing this and using the definition of $\langle f,g\rangle$ 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 $X \in \langle f,g\rangle$ with $X > \mathbf{0}$, π must satisfy $\pi(X) > \mathbf{0}$.

Any π satisfying (C2) also satisfies (C1). Hence, rule (C2) gives a sufficient condition for loop (5) to be equivalent to loop (4). Moreover, (C2) is independent of the index set $\mathcal{I}$.

Each condition $\pi(X) > \mathbf{0}$ given by rule C2 is a constraint on our choice of π. If π satisfies all these constraints, then loop (5) is equivalent to loop (4). Table I lists the constraints on π for loop (1). In this case $\pi : \mathbf{Z}^3 \to \mathbf{Z}$ is of the form $\pi[(i,j,k)] = a_1 i + a_2 j + a_3 k$, and Table I gives the constraints which must be satisfied by a_1, a_2, and a_3. For example, the set of elements $> \mathbf{0}$ in $\langle u1,\ u2\rangle$ is $(+,-1,0) = \{(x,-1,0) : x > 0\}$. The requirement that $\pi[(x,-1,0)] > 0$ for each $x > 0$ yields the constraint $a_1 - a_2 > 0$. Our choice of $a_1 = 2$, $a_2 = a_3 = 1$ satisfies all these constraints. Therefore, loop (3) is equivalent to loop (1).

Computing the Sets $\langle f,g\rangle$

In order to guarantee that we find a mapping π which satisfies (C2), some further restriction must be made on the forms of variable occurrences allowed in the loop body. We make the following assumption.

(A5) Each occurrence of a generated variable VAR in the loop body is of the form

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

where the m^i are integer constants, and $j_1, \ldots, j_r$ are r distinct integers between 1 and n. Moreover, the j_i are the same for any two occurrences of VAR.

Thus, if a generation $A(I^2-1,I^1,I^4+1)$ appears in the loop body, then the occurrence $A(I^2+1,I^1+6,I^4)$ may also appear. However, the occurrence $A(I^1-1,I^2,I^4)$ may not.

It is possible to generalize our results to the case of occurrences of the form $VAR(e^1, \ldots, e^r)$ in which e^i is any linear function of $I^1, \ldots, I^n$. However, the results become weaker and much more complicated.

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 $T_f[(p^1, \ldots, p^n)] = (p^{j_1}+m^1, \ldots, p^{j_r}+m^r)$, and $T_g[(p^1, \ldots, p^n)] = (p^{j_1}+n^1, \ldots, p^{j_r}+n^r)$. It is easy to see from the definition that $\langle f,g\rangle$ is the set of all elements of $\mathbf{Z}^n$ whose j_kth 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 $\langle f,g\rangle$ is the set $\{(x,5,0,y,-1) : x, y \in \mathbf{Z}\}$, which we denote by $(*,5,0,*,-1)$.

The index variable I^j is said to be *missing from 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 jth coordinate, for any occurrences f,g of VAR. We call I^j a *missing* index variable if it is missing from some generated variable in the loop.

The Hyperplane Theorem

The following result is an important special case of a more general result which will be given later.[4] The proof contains an algorithm for constructing a mapping π which satisfies (C2). The reader can check that it gives the π which we used for loop (1). As we will see in loop (11) below, the algorithm sometimes works even if the hypothesis of the theorem is not satisfied.

HYPERPLANE CONCURRENCY THEOREM. *Assume that loop* (4) *satisfies* (A1)–(A5), *and that none of the index variables* $I^2, \ldots, I^n$ *is a missing variable. Then it can be rewritten in the form of loop* (5) *for* $k = 1$. *Moreover, the mapping* J *used for the rewriting can be chosen to be independent of the index set* $\mathcal{I}$.

PROOF. We will first construct a mapping $\pi : \mathbf{Z}^n \to \mathbf{Z}$ which satisfies rule (C2). Let $\mathcal{P}$ be the set consisting of all the elements $X > \mathbf{0}$ of all the sets $\langle f,g \rangle$ 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 $\mathbf{Z}^n$ of the form $(x, x^2, \ldots, x^n)$ with $x > 0$. Since I^1 is the only index variable which may be missing, we can write $\mathcal{P} = \{X_1, \ldots, X_N\}$, where $X_r = (x_r^1, \ldots, x_r^n)$, or $X_r = (+, x_r^2, \ldots, x_r^n)$ for some integers x_r^i.

The mapping π is defined by

$$\pi[(I^1, \ldots, I^n)] = a_1 I^1 + \cdots + a_n I^n \tag{8}$$

for nonnegative integers a_i, 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 $r = 1, \ldots, N$.

Define $\mathcal{P}_j = \{X_r : x_r^1 = \cdots = x_r^{j-1} = 0, x_r^j \neq 0\}$, so $\mathcal{P}_j$ is the set of all X_r whose jth coordinate is the leftmost nonzero 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 nonnegative integer such that $a_j x_r^j + \cdots + a_n x_r^n > 0$ for each $X_r = (0, \ldots, 0, x_r^j, \ldots, x_r^n) \in \mathcal{P}_j$. Since $X_r > \mathbf{0}$ and $x_r^j \neq 0$ imply $x_r^j > 0$, 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 nonzero 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 in [4, p. 31], then gives a one-to-one linear mapping $J : \mathbf{Z}^n \to \mathbf{Z}^n$ such that $J[(I^1, \ldots, I^n)] = (\pi[(I^1, \ldots, I^n)], \ldots)$.[5]

Since the sets $\langle f,g \rangle$ 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. □

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 $S_{J^2}, \ldots, S_{J^n}$ contain at most one element, and the order of execution of the loop body is unchanged. For example, if $\mathcal{I} = \{(x,y,z) : 1 \leq x \leq 10, 1 \leq y \leq 5, 1 \leq z \leq 7\}$, 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 $\pi : \mathbf{Z}^n \to \mathbf{Z}$. 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 (5). (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 (5). But λ^1 and μ^1 are just the upper and lower bounds of $\{\pi(P) : P \in \mathcal{I}\}$. Setting $M^i = u^i - l^i$, it is easy to see that $\mu^1 - \lambda^1$ equals

$$M^1 \mid a_1 \mid + \cdots + M^n \mid a_n \mid, \tag{9}$$

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 $x^1 a_1 + \cdots + x^n a_n > 0$, 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 [4] can be applied to finding 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 such as loop (1), it actually gives the optimal solution.

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. The following theorem may be viewed as a generalization of a result stated in [5, p. 584].

[4] A weaker version of this result can be found in [3].

[5] If $a^j = 1$, then we can define J as follows: for each $k \geq 2$, let J^k equal some distinct I^{l_k} with $l_k \neq j$.

PLANE CONCURRENCY THEOREM. *Assume that loop* (4) *satisfies* (A1)–(A5) *and that at most* $k-1$ *of the index variables* $I^2, \ldots, I^n$ *are missing. Then loop* (4) *can be rewritten in the form of loop* (5). *Moreover, the mapping* J *used for the rewriting can be chosen to be independent of the index set* $\mathcal{S}$.

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 $j_1 < j_2 < \cdots < j_k < j_{k+1}$.

Let $\mathcal{P}$ be the set of all elements $X > \mathbf{0}$ of all sets $\langle f,g\rangle$ referred to by rule (C2). We must construct π so that $\pi(X) > \mathbf{0}$ 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 jth coordinate is the left-most nonzero one. Then every element of $\mathcal{P}$ is in one of the $\mathcal{P}_j$.

The mapping $\pi : \mathbf{Z}^n \to \mathbf{Z}^k$ will be constructed with $\pi(P) = (\pi^1(P), \ldots, \pi^k(P))$, where each $\pi^i : \mathbf{Z}^n \to \mathbf{Z}$ is defined by $\pi^i[(I^1, \ldots, I^n)] = a_1{}^i I^1 + \cdots + a_n{}^i I^n$ for nonnegative integers $a_j{}^i$. Moreover, we will have $a_j{}^i = 0$ if $j < j_i$ or $j \geq j_{i+1}$. This implies that if $X \in \mathcal{P}_j$ and and $j \geq j_{i+1}$, 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 each $X \in \mathcal{P}_j : \pi^i(X) > 0$—for we then have $\pi(X) = (0, \ldots, 0, \pi^i(X), \ldots, \pi^k(X)) > \mathbf{0}$.

Recall that for the sets $\langle f,g\rangle$, an $*$ can appear only in the $j_1, \ldots, j_k$ coordinates. Thus any element of any of the sets $\mathcal{P}_j$ with $j_i \leq j < j_{i+1}$ can be represented in the form $(0, \ldots, 0, x_r^{j_i}, \ldots, x_r^{j_{i+1}-1}, \ldots)$, or $(0, \ldots, 0, +, x_r^{j_i+1}, \ldots, x_r^{j_{i+1}-1}, \ldots)$ for a finite collection of integers x_r^j, $j_i \leq j < j_{i+1}$. By the same argument used in the proof of the Hyperplane Theorem, we can replace "+" by $x_r^{j_i} = 1$, and choose $a_j{}^i \geq 0$, $j_i \leq j < j_{i+1}$ such that $a_{j_i}^i x_r^{j_i} + \cdots + a_{j_{i+1}-1}^i x_r^{j_{i+1}-1} > 0$ for each r. Choosing $a_j{}^i = 0$ for $j < j_i$ and $j \geq j_{i+1}$ completes the construction of the required π^i.

The construction of [4] is then applied to give invertable relations of the form $J^{j_i} = a_{j_i}^i \cdot I^{j_i} + \cdots + a_{j_{i+1}-1}^i \cdot I^{j_{i+1}-1}$, and $J_r^j = \sum_{r=j_i}^{j_{i+1}-1} b_r^j \cdot I^r$, for $j_i < j < i_{i+1}$. Combining these and reordering 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 $(\mu^1 - \lambda^1 + 1) \cdots (\mu^k - \lambda^k + 1)$. It is easy to verify that if none of the expressions l^i, u^i involve any index variable, then this number is equal to $(M^1 \mid a_1{}^1 \mid + \cdots + M^n \mid a_n{}^1 \mid + 1) \cdots (M^1 \mid a_1{}^k \mid + \cdots + M^n \mid a_n{}^k \mid + 1)$, where $M^i = u^i - l^i$, and the $a_j{}^i$ are defined by (6).

Finding the best $a_j{}^i$ is now an integer programming problem. Note that a solution with $a_1{}^i = \cdots = a_n^i = 0$ 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 $\pi : \mathbf{Z}^n \to \mathbf{Z}^k$ satisfying (C2), for a particular value of k. It may be possible to find such a π for a smaller k.

For completeness, we state a sufficient condition for the loop body to be concurrently executable for all points in $\mathcal{S}$—i.e. to be able to rewrite loop (4) with a

DO α *CONC FOR ALL* $(I^1, \ldots, I^n) \in S$

statement. This involves setting J equal to the identity mapping, $k = 0$, and $\pi : \mathbf{Z}^n \to \mathbf{Z}^0$ the mapping defined by $\pi(P) = 0$ for all $P \in \mathbf{Z}^n$. Since $\langle g,f\rangle = \{-X : X \in \langle f,g\rangle\}$, it is clear that this π satisfies (C2) if and only if all the sets $\langle f,g\rangle$ are equal to $\{\mathbf{0}\}$. The method of computing these sets then gives the following rather obvious result.

> If loop (4) satisfies (A1)–(A5), none of the index variables are missing, and all occurrences of any generated variable are identical, then the loop can be rewritten as:
>
> *DO* α *CONC FOR ALL* $(I^1, \ldots, I^n) \in S$

The hypothesis means that in the expression (6) for each generated variable *VAR*, $r = n$ and the m^i are the same for all occurrences of *VAR*.

II. The Coordinate Method

Example. We illustrate the coordinate method with the following loop.

```
      DO 24 I = 2, M
      DO 24 J = 1, N
21    A(I, J) = B(I, J) + C(I)
          (a1)      (b1)     (c1)
22    C(I)    = B(I - 1, J)
          (c2)      (b2)
23    B(I, J) = A(I + 1, J) ** 2
          (b3)      (a2)
24    CONTINUE                          (11)
```

The hyperplane method would rewrite this as a *DO* $\bar{I}$/*DO CONC* $\bar{J}$ loop with $\bar{I} = I + J$, and $\bar{J} = J$. (Although J is a missing variable, so the hypothesis of the hyperplane theorem is not satisfied, the algorithm used in the proof still gives a π satisfying (C2).) The rewritten loop has $M + N - 2$ sequential iterations.

For a synchronous, single instruction stream computer like the ILLIAC IV, we can do better than this by using the coordinate method. To express synchronous parallel execution, we introduce the *DO SIM* (for *SIM*ultaneously) statement having the following form.

DO α *SIM FOR ALL* $\in S$,

where S is a finite set of integers. Its meaning is similar to that of the *DO CONC* statement, except that the computation is performed synchronously by the individual processors. Each element of S 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.

The coordinate method does not introduce new index variables. It will rewrite loop (11) as

```
      DO 24 J = 1, N
      DO 24 SIM FOR ALL I ∈ {i : 2 ≤ i ≤ M}
      TEMP(I) = A(I + 1, J)
                (a2)
21    A(I, J)  = B(I, J) + C(I)
      (a1)       (b1)      (c1)
23    B(I, J)  = TEMP(I) ** 2
      (b3)
22    C(I)     = B(I − 1, J)
      (c2)       (b2)
24    CONTINUE                                  (12)
```

Observe that the *DO SIM* must be executed by synchronous processors. Processor i must generate the value for $B(i, j)$ in statement 23 before processor $i + 1$ uses it in statement 22. We also see from this that it was necessary to rearrange the loop body in writing loop (12) in order to obtain a loop equivalent to the original one.

Loop (12) requires only N sequential iterations, instead of the $M + N - 2$ required by the hyperplane method. Moreover, the change of index variables in the hyperplane method produces more complicated subscript expressions, significantly increasing the time needed for a single execution of the loop body. By using the original index variables, the coordinate method eliminates this source of inefficiency. However, there are some loops, such as loop (1), which cannot be rewritten with the coordinate method. These loops require the hyperplane method.

Assumptions and Notation

In general, we consider a loop of the form

```
      DO α I^1 = l^1, u^1, d^1
               ⋮
      DO α I^n = l^n, u^n, d^n
        [loop body]
α     CONTINUE                                  (13)
```

We assume that the loop body satisfies assumptions (A1)–(A4). In addition, we make the following assumptions:

(A6) Each d^i is an integer constant.

(A7) There is no conditional transfer of control within the loop body.

Assumption (A7) prohibits a statement such as

```
IF (A(I^1).GT.0) GO TO 9
```

in the loop body. Such a statement would be meaningless inside a *DO SIM* I^1 loop, since all processors must execute the same statement. However, we do allow a conditional assignment statement such as

```
IF (A(I^1).GT.0) B(I^1) = A(I^1)
```

It is easily implemented on the ILLIAC IV by turning off individual processors. The real assumption in (A7) is that there are no loops within the loop body. In that case, conditional branches can be removed by adding *IF* clauses.

We are not making assumption(A5). Some restrictions on subscript expressions must be made by a real compiler to permit computation of the $\langle f,g \rangle$ sets. We will not consider this problem.

To simplify the discussion, we assume that each d^i equals 1, and that the expressions l^i and u^i do not contain any of the index variables I^j. The modifications necessary for the general case are described later.

The coordinate method will rewrite loop (13) as

```
      DO α I^j1 = l^j1, u^j1
                ⋮
      DO α I^jk = l^jk, u^jk
      DO α SIM FOR ALL (I^jk+1, ..., I^jn) ∈ S
        [loop body]
α     CONTINUE                                  (14)
```

where $j_1 < \cdots < j_k$ and S is the set $\{(x^{k+1}, \ldots, x^n) : l^{j_i} \leq x^i \leq u^{j_i}\}$.

The mapping $\pi : \mathbf{Z}^n \rightarrow \mathbf{Z}^k$ is defined as before. However, now it is the simple mapping $\pi[(i^1, \ldots, i^n)] = (i^{j_1}, \ldots, i^{j_k})$. In other words, π just deletes the $j_{k+1}, \ldots, j_n$ coordinates. In our example, π was defined by $\pi[(i, j)] = j$.

Basic Considerations

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 trying to find a π which satisfies (C2). However, the synchrony of the computation allows 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.

First, assume that we do not change the loop body. For two occurrences f and g, let $f \rightarrow g$ denote that the execution of f precedes the execution of g within the loop body. This means either that the statement containing f precedes the statement containing g, or that f is a use and g a generation in the same statement. The

Table II.

The sets $\langle f,g\rangle$	Is (S1 (i)) violated?	Ordering relations S1 (ii)	S2
$\langle a1,a1\rangle = (0,0)$	NO	—	—
$\langle a1,a2\rangle = (-1,0)$	NO	—	—
$\langle a2,a1\rangle = (1,0)$	NO	$a2 \rightarrow a1$	—
$\langle b3,b3\rangle = (0,0)$	NO	—	—
$\langle b1,b3\rangle = (0,0)$	NO	—	$b1 \rightarrow b3$
$\langle b3,b1\rangle = (0,0)$	NO	—	—
$\langle b2,b3\rangle = (-1,0)$	NO	—	—
$\langle b3,b2\rangle = (1,0)$	NO	$b3 \rightarrow b2$	—
$\langle c1,c1\rangle = (0,*)$	NO	—	—
$\langle c1,c2\rangle = (0,*)$	NO	—	$c1 \rightarrow c2$
$\langle c2,c1\rangle = (0,*)$	NO	—	—

above observation allows us to change rule (C1) to the following weaker condition on π.

> For . . . generation: if $T_f(P) = T_g(P)$ 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* $f \rightarrow g$.

In this rule, either (i) or (ii) is sufficient to ensure that occurrence f references the array element $T_f(P)$ for the point $P \in \mathcal{I}$ before g references the same array 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* $\pi(P) = \pi(Q)$ *then* $f \rightarrow g$.

In the same way that (C2) was obtained from (C1), the above rule gives the following rule.

(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 $X \in \langle f,g\rangle$ with $X > \mathbf{0}$, we must have
(i) $\pi(X) \geq \mathbf{0}$, *and*
(ii) *if* $\pi(X) = \mathbf{0}$, *then* $f \rightarrow g$.

If π satisfies rule (S1), then it satisfies the preceding rule, so the rewritten loop (14) is equivalent to the original loop (13).

So far, our discussion has assumed that we have not changed 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, as we did in writing loop (12). (There was no point in doing this for asynchronous processors since it couldn't help.)

Let $f \rightarrow g$ mean that f is executed before g in the *rewritten* loop body. 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 $P \in \mathcal{I}$ and f precedes g in the original loop body, then $f \rightarrow g$.

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 $\mathbf{0} \in \langle f,g\rangle$ and f precedes g in the original loop body, then $f \rightarrow g$.

Rules (S1) and (S2) guarantee that the rewritten loop (14) is equivalent to the original loop (13). Note that rule (S2) does not involve π.

The Coordinate Algorithm

(S1) and (S2) together give a sufficient condition for a particular rewriting to be equivalent to the original loop. Rule (S1(i)) gives a condition which must be satisfied by π. Rules (S1(ii)) and (S2) specify ordering relations among the occurrences in the rewritten loop body. However, they do not indicate whether it is possible to rewrite the loop body so that these relations are satisfied. We now give a method for deciding if such a rewriting exists.

First, we make a trivial observation: a use in an assignment statement must precede the generation in that statement. This observation is given the status of a rule.

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

Now let $\rightarrow$ denote the relations given by rules (S1)–(S3). Add all relations implied by transitivity. That is, whenever $f \rightarrow g$ and $g \rightarrow h$, add the relation $f \rightarrow h$. (An efficient algorithm for doing this is given by [6].) If the resulting ordering relations are consistent—that is, if we do not have $f \rightarrow f$ for any occurrence f—then the loop body can be rewritten to satisfy the ordering relations.

To show how the rewriting is actually done, we describe the application of the coordinate method to loop (11). The calculations for Steps 1, 3, and 4 are shown in Table II.

Step 1. Compute the relevant sets $\langle f,g\rangle$ for rules (S1) and (S2).
Step 2. Choose the *DO SIM* variables. We wish to rewrite loop (11) as a *DO J/DO SIM I* loop, so the mapping π is defined by $\pi[(i,j)] = j$.
Step 3. Check that (S1(i)) is not violated.
Step 4. Find the ordering relations given by (S1(ii)) and (S2).

Step 5. Apply (S3) to get the following relations:

statement 21: $b1 \rightarrow a1$
$c1 \rightarrow a1$
statement 22: $b2 \rightarrow c2$
statement 23: $a2 \rightarrow c3$

Step 6. Find all relations implied by transitivity:

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

Step 7. Check that no relation of the form $f \rightarrow f$ was found in Step 4 or Step 6.

Step 8. Order the generations in any way which is consistent with the above relations—i.e. obeying $b3 \rightarrow c2$. We let $a1 \rightarrow b3 \rightarrow c2$. We then write:

```
21  A(I, J) =
      (a1)
23  B(I, J) =
      (b3)
22  C(I) =
      (c2)
```

Step 9. Insert the uses in positions implied by the ordering relations (recall that $a2 \rightarrow a1$):

```
          A(I + 1, J)
              (a2)
21  A(I, J) = B(I, J) + C(I)
      (a1)      (b1)     (c1)
23  B(I, J) =   **2
      (b3)
22  C(I) = B(I - 1, J)
      (c2)     (b2)
```

Step 10. Add any extra variables necessitated by uses no longer appearing in their original statements:

```
          TEMP(I) = A(I + 1, J)
                        (a2)
21  A(I, J)   = B(I, J) + C(I)
      (a1)      (b2)     (c1)
23  B(I, J)   = TEMP(I) ** 2
      (b3)
22  C(I)      = B(I - 1, J)
      (c2)      (b2)
```

Step 11. Insert the *DO* and *DO SIM* statements, to get loop (12).

Further Remarks

It is easy to deduce a general algorithm for the coordinate method from the preceding 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 $g \rightarrow g$ does *not* hold can be executed within a *DO SIM* loop. An algorithm for doing this is described in [7].

In general, there are $2^n - 1$ choices for the *DO SIM* variables in Step 2. Steps 3–11 are repeated for different choices until a suitable one is found. Rule (S1) should quickly eliminate many possibilities. In our example, the choice of a *DO I/DO SIM J* rewriting is eliminated by the relation $c1 \rightarrow c1$ given by (S1(ii)). One can also show that loop (13) can be rewritten with a *DO SIM* $(I^{j_k}, \ldots, I^{j_n})$ only if it can be rewritten with a *DO SIM* $(I^{j_{k+1}}, \ldots, I^{j_n})$, where $j_k < \cdots < j_n$. Thus, eliminating *DO I/DO SIM J* for loop (11) also eliminates the possibility of a *DO SIM (I,J)* rewriting. If no choice of *DO SIM* variables works, then the hyperplane method must be tried.

To handle arbitrary *DO* increments d^i, one need only generalize the definition of the set $\langle f,g \rangle$ as follows: $\langle f,g \rangle = \{(x^1, \ldots, x^n) \in \mathbf{Z}^n : T_f(P) = T_g[P + (d^1x^1, \ldots, d^nx^n)]$, for some $P \in \mathbf{Z}^n\}$. The rules (S1)–(S3) and the algorithm described above remain the same.

For arbitrary *DO* limits l^i, u^i we proceed as follows. For each i: if the expression l^i contains some I^j, then replace I^i by the new index variable $\bar{I}^i = I^i - l^i$. Steps 1–10 are then executed as before. In Step 11, a more complicated procedure is needed to find the *DO* limits and *DO SIM* set for the rewritten loop.

III. Practical Considerations

Satisfying the Assumptions

Our analysis required several assumptions about the given loop. If a loop does not satisfy these assumptions, then it may still be possible to rewrite it so that it does. We have already indicated that assumption (A7) can be met by replacing conditional transfers with *IF* clauses. We now describe some other useful techniques.

Our first assumption was that the *DO*s are *tightly nested*, as in loop (4); i.e. we did not allow loops such as

```
      DO 99 I = 1, M
21    A(I, 1) = 0
      DO 99 J = 2, N
            ⋮
```

It is easy to rewrite this as the following tightly nested loop:

```
      DO 99 I = 1, M
      DO 99 J = 2, N
21    IF (J .EQ. 2) A(I,J-1) = 0
            ⋮
```

This method works in general. It may be possible later to move statement 21 back outside the J loop and remove the *IF* clause. A future paper will describe methods of handling nontightly nested loops without using this artifice [8].

Assumption (A4) can sometimes be satisfied by substituting for generated variables. One technique is illus-

trated 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
```

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

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.[6]

Often, the variable simply acts as 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. (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.) A simple analysis of the loop body's flow path determines if this is possible.

Another common situation is for the variable X to appear in the loop body only in the statement $X = X +$ 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 the statement $VAR(I^1, \ldots, I^n) =$ expression, and add the following "statement" after the loop: $X = X + \sum_{(I^1,\ldots,I^n)\in\mathcal{I}} VAR(I^1, \ldots, I^n)$. 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 $\mathcal{I}$.

Practical Restrictions

The methods we have described yield parallelism in the form of *DO CONC* or *DO SIM* loops. In order for them to be of use in a real compiler, the particular target computer must be capable of efficiently executing these loops in parallel. The structure of the computer will place additional restrictions on the loops which the compiler can handle.

[6] In our formalism, a scalar is a zero-dimensional array. Each $\langle f,g\rangle$ set for a scalar variable equals all of $\mathbf{Z}^n$.

Consider the loop

```
      DO 2 I = 1, N
      A(2*I-1) = B(I)
    2 A(2*I) = C(I)
```

The coordinate method can rewrite this as a *DO SIM I* loop. However, to execute this *DO SIM* loop in parallel on the ILLIAC IV requires a peculiar method of storing the arrays. This storage scheme would probably be incompatible with the requirements of the rest of the program.

In general, the computer's data accessing mechanism will limit the forms of variable occurrences which may appear in the loop. It may also limit the utility of the hyperplane method. For example, implementation of the hyperplane method on the STAR-100 requires dynamic reformating of the arrays.

We have allowed a conditional assignment statement such as

```
IF (A(I).GT.0)B(I) = A(I)
```

inside a *DO SIM I* loop. This is easily implemented on the ILLIAC IV and with vector operations on the STAR-100. However, it cannot be implemented with the ASC vector operations.

Other computer designs will require different restrictions on the loops. However, our methods seem sufficiently general to be applicable to any parallel computer to be built in the near future.

Conclusion

We have presented methods for obtaining parallel execution of a *DO* loop nest. A number of details and refinements were omitted for simplicity. Some of these are described in [7]. However, all the basic ideas necessary for their implementation have been included. Preliminary experience with the ILLIAC IV FORTRAN compiler indicates that these methods can be used to obtain parallel execution for a fairly large class of sequential programs.

Received February 1972; revised January 1973

References

1. McIntyre, David. An introduction to the ILLIAC-IV computer. *Datamation 16*, 4 (Apr. 1970), 60–67.
2. Ramamoorthy, C.V., and Gonzalez, M.J. A survey of techniques for recognizing parallel processable streams in computer programs. Proc. AFIPS 1969 FJCC, Vol. 35. AFIPS Press, Montvale, N. J. pp. 1–15.
3. Muroaka, Yoishi. Parallelism exposure and exploitation in programs. Ph.D. Th., U. of Illinois, Urbana, Ill., 1971.
4. Mordell, L.J. *Diophantine Equations.* Academic Press, New York, 1969.
5. Karp, R.M., Miller, R.E., and Winograd, S. The Organization of computations for uniform recurrence equations. *J. ACM 14*, 3 (July 1967), 563–590.
6. Warshall, Stephen. A theorem on Boolean matrices. *J. ACM 9*, 1 (Jan. 1962), 11–12.
7. Lamport, Leslie, and Presberg, David. The parallel execution of FORTRAN DO loops. Mass. Computer Associates, Inc., AD 742-279. Wakefield, Mass. 1971.
8. Lamport, Leslie. The coordinate method for the parallel execution of DO loops. To appear in Proc. 1973 Sagamore Comput. Conf.