SRI International

The Coordinate Method for the Parallel Execution of Iterative Loops

by

Leslie Lamport

August 2, 1976
CA-7608-0221
Minor Revisions: 31 May 1979 and
21 October 1981

This work was done while the author was at Massachusetts Computer Associates, and was supported by the Advanced Research Projects Agency of the Department of Defense and Rome Air Development Center. It was monitored by Rome Air Development Center under contract number F 30602-76-C0094.

SRI International®
333 Ravenswood Ave. • Menlo Park, CA 94025
(415) 859-6200 • TWX: 910-373-2046 • Telex: 334 486

Abstract

A method is described for compiling a sequential program for execution on an array or vector computer. It transforms sequential loops so that the different iterations are executed in parallel, generalizing previous techniques for doing this. A method of analyzing and representing a program is developed which can be applied to other optimization problems.

Key Words and Phrases: parallel computing, array computers, vector computers, nested loops, compiler optimization, multiprocessor computers.

CR Categories: 4.12, 5.24

Table of Contents

Introduction 1

The Programming Language 3

The Essential Aspects of the Program . 8

The Tree Representations 19

Equivalent Tree Representations. . . . 23

The Coordinate Algorithm 26

The Coordinate Method 34

The Hyperplane Method 37

Conclusion 39

References 42

Figures 44

Tables 49

Introduction

There are currently several array and vector computers in operation. The need for increased computation power and the decreasing cost of hardware will lead to the wider availability of such parallel computers. We consider the problem of compiling a sequential program for parallel execution by these computers. Different types of parallel computers will differ in the details of what they can execute in parallel, and writing efficient programs for them will not be easy. The ability to compile a sequential program into a parallel version tailored to the particular computer will simplify the programming task and make it easier to transfer a program from one computer to another.

The coordinate method described here basically attempts to transform iterative loops so that all the iterations are performed in parallel. This seems to be the most practical approach for current array and vector computers. However, it may not be possible to execute an entire loop in this way. We will illustrate the coordinate method by applying it to the loop of Figure 1. This loop is written in a simple programming language which will be precisely defined later, but its meaning should be clear. (The circled items are for future reference, and should be ignored now.) The SIM FOR ALL K statement specifies simultaneous execution of its DO clause for all the indicated values of K, and might have been generated by a previous step in the coordinate method. The loop does not compute anything of interest, but is contrived to provide a compact illustration of the coordinate method's capability. In attempting to execute all iterations of this FOR I loop in parallel, the coordinate method can transform this loop into the program shown in Figure 2. To obtain the maximum amount of parallel execution, the FOR I loop is split into three parts: the parallel SIM FOR ALL $I^{(1)}$ and SIM FOR ALL $I^{(3)}$ statements, and the sequential FOR $I^{(2)}$ loop. Note the radical restructuring that is performed. The FOR $I^{(2)}$ and SIM FOR $ALL^{(3)}$ statements have been moved inside the FOR J loop, and the single IF statement has been split so that the ELSE

clause is in the FOR $I^{(2)}$ loop and the THEN clause is in the SIM FOR ALL $I^{(3)}$ statement. In general, the coordinate method can be applied to any program containing nested sequential loops. This should include most programs suitable for execution by an array or vector computer.

Much work has already been done on the problem of sequential to parallel conversion [3, 4, 5, 7, 9, 11, 12]. The method described in this paper generalizes the coordinate method of [4], as well as generalizing several other approaches [9, 10, 11]. A way of generalizing the hyperplane method of [4] is also sketched. Preliminary versions of some of the results described here originally appeared in [5].

Perhaps more important than the actual sequential to parallel conversion techniques themselves is the method of analyzing and representing a program upon which they are based. Translating a sequential program to an equivalent parallel version may require changing the order in which different parts of the program are executed. Such a program transformation is usually described in terms of moving code from one part of the program to another. We take a different approach. The program is first translated into a new form, called a tree representation, which eliminates the unnecessary execution orderings that were specified when it was written in an ordinary programming language. Sequential to parallel conversion is performed by transforming the tree representation. Code can then be generated directly from the new representation. We believe that any optimization technique which exploits the program's loop structure can best be performed by such a transformation of the tree representation.

The major part of the paper is devoted to the development of the tree representation. Although the ideas are basically simple, quite a bit of notation must be introduced. We feel that the elegance and simplicity of the results justifies the effort of coping with this unfamiliar notation. Moreover, the notation itself can be useful in implementing the coordinate method.

The Programming Language

We begin by describing a simple programming language for the source program, which allows the explicit specification of parallel execution. The language contains no input/output or subroutine calling statements, so it can only be used to represent part of an actual source program. The only data structures are k-dimensional array variables, for any k, and integer valued indices. (A scalar is a zero-dimensional array.)

Note: 1. The indices in Figure 1 are I, J, K and L. They are usually called index variables, but we reserve the term"variable" for a variable whose value may be changed by an assignment statement. End note.

An expression is defined in the usual way. The components of an expression are constants, variable occurrences and indices. A variable occurrence is an expression of the form V(subscripts), where V is a k-dimensional array variable and subscripts is a k-tuple of integer valued expressions. We will not bother to specify any declaration statements, and will not concern ourselves with the range of values which an element of an array variable may assume.

A program consists of a single statement. A statement consists of a list of elementary statements separated by semicolons. There are four types of elementary statements, illustrated in Figures 1 and 2: assignment, IF, FOR and SIM statements. For convenience, we assume that no two FOR or SIM statements have the same index.

To describe the semantics of this programming language precisely, we first introduce some notation. For any subscripted symbol or expression X_i and any integer $m \geq 0$, we let $\vec{X}_m$ denote the m-tuple $(X_1, \ldots, X_m)$. In particular, $\vec{X}_0$ denotes the 0-tuple $()$. We define $\underset{\text{MM}}{Z}^m$ to be the set of all m-tuples of integers. Addition and subtraction in $\underset{\text{MM}}{Z}^m$ are the usual vector operations defined by $\vec{i}_m \pm \vec{j}_m = \overrightarrow{i_m \pm j_m}$, and the zero element $\underset{\text{MM}}{0} \in \underset{\text{MM}}{Z}^m$ equals $(0, \ldots, 0)$. We let $\underset{\text{MM}}{<}$ [boldface $<$] denote the lexicographical ordering on $\underset{\text{MM}}{Z}^m$ defined inductively by $\vec{i}_m \underset{\text{MM}}{<} \vec{j}_m$ iff (if and only if) either (a) $\vec{i}_{m-1} \underset{\text{MM}}{<} \vec{j}_{m-1}$ or (b) $\vec{i}_{m-1} = \vec{j}_{m-1}$ and $i_m < j_m$.

Each program statement and expression S has associated with it a <u>free index list</u> of the form $\vec{I}_m ; \vec{J}_n$, where the I_k and J_ℓ are indices. We call $\vec{I}_m$ and $\vec{J}_n$ the FOR multi-index and the SIM multi-index of S, respectively. The FOR multi-index $\vec{I}_m$ indicates that S is in the DO clause of a FOR I_m statement, which is in the DO clause of a FOR I_{m-1} statement, etc. Similarly, the SIM multi-index indicates the nesting of SIM statements which contain S. For any $(\vec{i}_m ; \vec{j}_n) \in \underset{\text{MM}}{Z}^m \times \underset{\text{MM}}{Z}^n$, define $S(\vec{i}_m ; \vec{j}_n)$ to be the statement or expression obtained from S by substituting i_k for I_k, $k = 1, \ldots, m$, and j_ℓ for J_ℓ, $\ell = 1, \ldots, n$. The entire program is a single statement whose free index list is $() ; ()$.

A program statement or expression S with free index list $\vec{I}_m; \vec{J}_n$ is executed with a context of the form $\vec{i}_m; C$, where $\vec{i}_m \in Z^m$ and $C \subset Z^n$. Executing S with the context $\vec{i}_m; C$ means executing $S(\vec{i}_m; \vec{j}_n)$ simultaneously for all $\vec{j}_n \in C$, as defined below.

The entire program is executed with the context $(); Z^0$. The semantics of our programming language are then defined inductively by the following description of how a statement S with free index list $\vec{I}_m; \vec{J}_n$ is executed with the context $\vec{i}_m; C$. There are five cases to be considered.

S1. S is the list of elementary statements $S_1; \ldots ; S_p$. Each S_i is executed in turn with the context $\vec{i}_m; C$.

S2. S is the assignment statement

V[subscripts] := exp .

The expressions subscripts and exp are first executed by evaluating subscripts$(\vec{i}_m; \vec{j}_n)$ and exp$(\vec{i}_m; \vec{j}_n)$ for all $\vec{j}_n \in C$. The occurrence V[subscripts] is then executed by setting each array element V[subscripts$(\vec{i}_m; \vec{j}_n)$] equal to exp$(\vec{i}_m; \vec{j}_n)$ for all $\vec{j}_n \in C$. We require that if $\vec{j}_n, \vec{j'_n} \in C$ and $\vec{j}_n \neq \vec{j'_n}$, then subscripts$(\vec{i}_m; \vec{j}_n) \neq$ subscripts$(\vec{i}_m; \vec{j'_n})$. Hence, no array element is set twice by this execution.

S3. S is the statement

IF exp THEN S_1 ELSE S_2 FI .

The boolean valued expression exp is first executed by evaluating $\underline{exp}(\vec{i}_m; \vec{j}_n)$ for all $\vec{j}_n \in C$. Next, S_1 is executed with the context $\vec{i}_m; \{\vec{j}_n \in C : \underline{exp}(\vec{i}_m; \vec{j}_n) = \text{true}\}$, and then S_2 is executed with the context $\vec{i}_m; \{\vec{j}_n \in C : \underline{exp}(\vec{i}_m, \vec{j}_n) = \text{false}\}$.

S4. S is the statement

FOR I_{m+1} := lower.limit TO upper.limit DO S' OD . For simplicity, we only consider the case in which the integer valued expressions lower.limit and upper.limit do not contain any of the SIM indices J_ℓ . These expressions are first executed by evaluating $\underline{lower.limit}(\vec{i}_m)$ and $\underline{upper.limit}(\vec{i}_m)$, and S' is then executed successively with the contexts $\vec{i}_{m+1}; C$ for $i_{m+1} = \underline{lower.limit}(\vec{i}_m), \underline{lower.limit}(\vec{i}_m) + 1, \ldots, \underline{upper.limit}(\vec{i}_m)$. Of course, S' is not executed at all if $\underline{lower.limit}(\vec{i}_m) > \underline{upper.limit}(\vec{i}_m)$.

S5. S is the statement

SIM FOR ALL $J_{n+1} \in \mathcal{S}$ DO S' OD. .

The expression $\mathcal{S}$, whose value is a set of integers, is first executed by evaluating $\mathcal{S}(\vec{i}_m; \vec{j}_n)$ for all $\vec{j}_n \in C$. The statement S' is then executed with the context $\vec{i}_m; \{\vec{j}_{n+1} : \vec{j}_n \in C \text{ and } j_{n+1} \in \mathcal{S}(\vec{i}_m; \vec{j}_n)\}$.

Notes: 2. We have chosen the IF statement as our conditional statement for simplicity. Any conditional construction can be used, including conditional gotos. We need only require that the gotos neither transfer control into or out of a DO clause, nor form a loop.

3. For simplicity, we only define a FOR loop in which the index is incremented by one. The generalization of our method to an arbitrary constant increment is described later.

4. The assignment, IF and FOR statements have their usual meanings when they are not inside any SIM statement and are executed with the context $\vec{i}_m ; Z^0_{MM}$. In this case, one clause of the IF statement is executed with the context $\vec{i}_m ; Z^0_{MM}$ and the other clause with the context $\vec{i}_m ; \phi$, where ϕ denotes the empty set. A statement does nothing if it is executed with the context $\vec{i}_m ; \phi$.

5. The set C in the context $\vec{i}_m ; C$ is called the SIM context. The SIM and IF statements are used to change the SIM context under which their component clauses are executed. They can be eliminated by introducing a statement to explicitly set the SIM context. In fact, the SIM and IF statements are removed from a program in just this way when compiling it for execution on an array or vector computer. (If the computer has separate scalar operations, then an IF statement whose conditional expression contains no SIM indices is eliminated by using conditional gotos.)

6. Suppose a statement or expression S is executed once with the context $\vec{i}_m ; C$ and another time with the context $\vec{i'}_m ; C'$. Then $\vec{i}_m \neq \vec{i'}_m$ and the execution with context $\vec{i}_m ; C$ is specified by S1-S5 to precede the execution with context $\vec{i'}_m ; C'$ iff $\vec{i}_m < \vec{i'}_m$. The generalization of S4 to allow

lower.limit and upper.limit to contain SIM indices is made so that this remains true. We will not bother with the details of the general definition.

7. Suppose the statement

SIM FOR ALL $I \in \mathcal{S}$ DO S' OD

is to be executed on an array computer having fewer processors than there are elements in $\mathcal{S}$. Then S' cannot actually be executed simultaneously for all elements of $\mathcal{S}$. To minimize the amount of index computation, one would then like to "strip mine" the SIM statement by executing it as follows:

FOR $J := 1$ TO n DO SIM FOR ALL $I \in \mathcal{S}_J$ DO S' OD OD ,

where $\mathcal{S} = \mathcal{S}_1 \cup \ldots \cup \mathcal{S}_n$ and if $x \in \mathcal{S}_i$, $y \in \mathcal{S}_j$, and $i < j$ then $x < y$. This is not always possible, since it changes the execution order specified by S1-S5. The problem of strip mining a SIM statement is discussed in [6].

8. Many additional restrictions must be placed on a program to permit efficient execution by a particular computer. For example, some restriction on the subscript expression of a variable occurrence is needed to insure that the parallel fetching or storing of values implied by the SIM context can actually be performed in parallel by the computer. These restrictions will not concern us. End notes.

The Essential Aspects of the Program

The programming language provides a convenient method for describing programs. However, it is not well suited to our purpose because

it places unnecessary constraints on the order in which expressions are executed. We will extract from the program just those constraints which are necessary. This will lead to the tree representation of the program.

Executing a program involves the following three types of actions:

(1) Fetching the value of an array variable element

(2) Storing a value in an array variable element.

(3) Evaluating a function of the fetched values and the index values to determine:

  (i) the values of the indices for which a fetch or store is to be executed, or

  (ii) the particular array element to be fetched or stored, or

  (iii) the value to be stored.

We will concern ourselves with actions of type(1) and (2) and largely ignore type(3) actions. A complete representation of the program must include a specification of the functions to be evaluated. Defining this specification is not difficult, but it involves a number of uninteresting details which we wish to avoid. Instead, we will just consider the execution of the program to consist of the fetches and stores to variable elements. Our first task is to determine what ordering relations on these fetches and stores are needed to preserve the meaning of the program.

We will represent the program's nested FOR statement structure by a

tree, so we need some terminology for discussing trees. We let Ω denote the root of a tree, and use the customary father/son relation to describe the tree structure. Hence, Ω is an ancestor of all nodes other than itself. A node with no sons is called a leaf. Two distinct nodes are said to be cousins if neither one is an ancestor of the other.

For any node ν, we define $|\nu|$ to be the number of ancestors of ν, so $|\Omega| = 0$. We define the ancestor-tuple of ν to be $\vec{\mu}_{|\nu|-1}$, where $\Omega, \mu_1, \cdots, \mu_{|\nu|-1}, \nu$ is the unique path from the root to ν.

For any pair of leaves u, v we define $u \cap v$ to be the node ν such that (i) ν is an ancestor of both u and v, and (ii) if μ is any other ancestor of both u and v then μ is an ancestor of ν. Hence, $u \cap u$ is the father of u.

We now define the program tree of a program to be the tree whose root is Ω, whose leaves are the program's variable occurrences, and whose remaining nodes are the program's FOR indices. The expected tree structure is defined by requiring that the ancestor-tuple of each occurrence equals its FOR multi-index. Each variable occurrence in the programs of Figures 1 and 2 has been given a name, which appears in a circle beneath the occurrence. The program tree for the program of Figure 1 is given in Figure 3.

The semantics of our programming language defines a partial ordering relation $\rightarrow$ among the variable occurrences, where $u \rightarrow v$ means that the oc-

currence u precedes the occurrence v in the execution sequence. More precisely, if u and v are occurrences in a statement S , then $u \rightarrow v$ in the following cases.

L1. S is the statement $S_1; \ldots ; S_p$, u appears in S_i , v appears in S_j and $i < j$.

L2. S is the statement

v := exp

and u appears in the expression exp .

L3. S is the statement

IF exp THEN S_1 ELSE S_2 FI

and either (i) u appears in exp and v appears in S_1 or S_2 , or (ii) u appears in S_1 and v appears in S_2 .

L4. S is the statement

FOR I := lower.limit TO upper.limit DO S' OD ,

u appears in lower.limit or upper.limit and v appears in S' .

L5. S is the statement

SIM FOR ALL $J \in \mathcal{S}$ DO S' OD ,

u appears in the expression $\mathcal{S}$ and v appears in S' .

L6. The occurrence v is of the form V[subscripts] and u appears in the expression subscripts .

Let v be an occurrence of the form V[subscripts] with free index list $\vec{i}_m ; \vec{j}_n$. We call v a generation if it appears as the left-hand side of an assignment statement, otherwise it is called a use. A generation is executed by setting the value of an array element, and a use is executed by fetching its value. We define $\mathcal{Z}(v)$ to equal $\underset{\text{MM}}{Z^m} \times \underset{\text{MM}}{Z^n}$, and define the index set of v , denoted by $\mathcal{I}(v)$, to be the subset of $\mathcal{Z}(v)$ consisting of all elements $(\vec{i}_m ; \vec{j}_n)$ such that v is executed under a context $\vec{i}_m ; C$ with $\vec{j}_n \in C$. During the execution of the program, the occurrence v is executed once for each element $(\vec{i}_m ; \vec{j}_n)$ in $\mathcal{I}(v)$. This execution of v is said to reference the the array element V[subscripts$(\vec{i}_m ; \vec{j}_n)$] , fetching its value if v is a use and setting its value if v is a generation. For any element $(\vec{i}_m ; \vec{j}_n) \in \mathcal{Z}(v)$, we define $(\vec{i}_m ; \vec{j}_n)|_{I_k}$ to equal $\vec{i}_k \in \underset{\text{MM}}{Z^k}$, $k = 1 , \ldots , m$. We also define $(\vec{i}_m ; \vec{j}_n)|_{\Omega}$ to equal $() \in \underset{\text{MM}}{Z^0}$.

In the program of Figure 1, we have $\mathcal{Z}(c2) = \underset{\text{MM}}{Z^2} \times \underset{\text{MM}}{Z^1}$ and $\mathcal{I}(c2) = \{(i, \ell ; k) : 1 \le i \le R[\,] , 3 \le \ell \le 9 , 0 \le k \le 63$ and $F[\, i, k \,] > 0\}$. The index set thus depends upon the initial values of variables. Executing c2 for the element $(i, \ell ; k) \in \mathcal{I}(c2)$ references C[i-1 , k+1 , ℓ-3] by fetching its value. For any $(i, \ell ; k) \in \mathcal{Z}(c2)$ we have $(i, \ell ; k)|_L = (i, \ell)$ and $(i, \ell ; k)|_I = (i)$.

We now ask what ordering of the individual executions of occurrences is specified by the program. Remembering Note 6, it is a simple exercise in understanding our notation to verify that the answer is given by the following condition.

PE. Let u, v be a pair of occurrences, let $\underset{\text{MM}}{p} \in \mathcal{J}(u)$ and $\underset{\text{MM}}{q} \in \mathcal{J}(v)$. The execution of u for $\underset{\text{MM}}{p}$ precedes the execution of v for $\underset{\text{MM}}{q}$ if either

(i) $\underset{\text{MM}}{p}|_{u \cap v} \leq \underset{\text{MM}}{q}|_{u \cap v}$, or

(ii) $\underset{\text{MM}}{p}|_{u \cap v} = \underset{\text{MM}}{q}|_{u \cap v}$ and $u \rightarrow v$.

If the order of two executions is not specified by PE, then those executions may be performed in either order, or simultaneously. Observe that PE applies when $u = v$ as well as when $u \neq v$.

Condition PE specifies many unnecessary orderings among executions because the relation $\rightarrow$ is stronger than it need be. For example, consider the statement

X[] := 0 ; Y[] := 0 .

Because of L1, condition PE specifies that the setting of X[] must precede the setting of Y[], even though the order of these operations is obviously irrelevant. We will therefore redefine the relation $\rightarrow$ to eliminate the unnecessary orderings specified by L1-L6.

In L2, L3(i), L4, L5 and L6, the value fetched by executing the use u for some $\underset{\text{MM}}{p} \in \mathcal{J}(u)$ is used either to execute v for some $\underset{\text{MM}}{q} \in \mathcal{J}(v)$ [L2, L6], or else to determine if $\underset{\text{MM}}{q}$ is an element of $\mathcal{J}(v)$ [L3(i), L4, L5]. The structure of the program implies that in all of these cases $\underset{\text{MM}}{p}|_{u \cap v} = \underset{\text{MM}}{q}|_{u \cap v}$, so the relation $u \to v$ is needed to preserve the logical structure of program. For later use, we restate these conditions more abstractly as follows.

R1. If u is a use and v any occurrence, and for some $\underset{\text{MM}}{p} \in \mathcal{J}(u)$, $\underset{\text{MM}}{q} \in Z(v)$ the value fetched when u is executed for $\underset{\text{MM}}{p}$ is used either:

(i) to determine if $\underset{\text{MM}}{q} \in \mathcal{J}(v)$, or

(ii) to evaluate the subscript expression of v when it is executed for $\underset{\text{MM}}{q} \in \mathcal{J}(v)$, or

(iii) to evaluate the expression which determines the value stored by executing v for $\underset{\text{MM}}{q} \in \mathcal{J}(v)$, v a generation;

then $\underset{\text{MM}}{p}|_{u \cap v} = \underset{\text{MM}}{q}|_{u \cap v}$ and $u \to v$.

Note: 9. Condition (i) is a restatement of L3(i), L4 and L5. It means that the value fetched by executing u for $\underset{\text{MM}}{p}$ is needed to decide if v should be executed for $\underset{\text{MM}}{q}$. The relation $u \to v$ insures that this value is available before the execution of v for $\underset{\text{MM}}{q}$. This is clearly necessary if v is a generation. It is not necessary if v is a use, since executing a fetch has no effect if the value fetched is never used. In practise, however, fetching the value might cause an out of bounds memory reference, and evaluating an expression with it might cause an overflow error. Thus, these theoretically unnecessary relations $\to$ usually must be included, even though they will prevent cer-

tain optimizations. (In a parallel or pipelined computer, it is often faster unconditionally to evaluate an expression even if its value might not subsequently be used.) This suggests a possible area for improvement in computer design. End note.

We now consider which of the relations $\rightarrow$ determined by the remaining conditions L1 and L3(ii) are necessary. Condition R1 insures that when executing the program, the arguments of functions are fetched before the functions need to be evaluated. We are thus free to regard the program execution as consisting only of fetch and store operations. We now ask which of the orderings among those operations are necessary to preserve the meaning of the program. The order of operations to different array elements, or of fetches to the same element, is clearly irrelevant. It is easy to see that the following condition suffices to preserve the meaning of the program.

Q1. If two occurrence executions reference the same array element, and at least one of them is a store, then the order of those executions must be preserved.

As an example, consider the following statement.

```
SIM FOR ALL I ∈ {i : 1 ≤ i ≤ 99}
        DO IF S[ I ] > 0 THEN T[ I ] := T[ I-1 ]
                         ELSE T[ I ] := T[ I ] + 1 FI OD
```

Condition L3(ii) implies that all the fetches of T[I-1] in the THEN clause precede all the stores of T[I] in the ELSE clause. Reversing the order of these fetches and stores may change the effect of executing the statement, thus altering the meaning of the program. It does not matter whether or not the fetches in the THEN clause are performed before the fetches in the ELSE clause. It also does not matter in which order the stores are performed, since they reference different array elements.

We define a relevant occurrence pair u, v to be a pair of occurrences of the same variable, at least one of which is a generation. Using condition PE, we may restate Q1 as follows.

Q2. For every relevant occurrence pair u, v : if there exist $\underset{\text{MM}}{p} \in \mathcal{J}(u)$ and $\underset{\text{MM}}{q} \in \mathcal{J}(v)$ such that

(a) the executions of u for $\underset{\text{MM}}{p}$ and of v for $\underset{\text{MM}}{q}$ reference same array element,

(b) $\underset{\text{MM}}{p}|_{u \cap v} = \underset{\text{MM}}{q}|_{u \cap v}$, and

(c) if $u = v$ then $\underset{\text{MM}}{p} \neq \underset{\text{MM}}{q}$,

then either $u \rightarrow v$ or $v \rightarrow u$.

Which of the two possible relations $u \rightarrow v$ or $v \rightarrow u$ holds is determined by L1-L6. Since u and/or v is a generation, L1-L6 imply that one of these relations holds when $u \neq v$. When $u = v$, the assumption made in S2 implies that (a) and (b) of Q2 cannot hold for $\underset{\text{MM}}{p} \neq \underset{\text{MM}}{q}$. Hence, that assumption is implied by Q2 and the fact that $v \not\rightarrow v$.

For $\underset{\sim}{p} \in \mathcal{J}(u)$ and $\underset{\sim}{q} \in \mathcal{J}(v)$, let $\underset{\sim}{x} = \underset{\sim}{q}|_{u \cap v} - \underset{\sim}{p}|_{u \cap v} \in \underset{\sim}{Z}^{|u \cap v|}$.

Then $\underset{\sim}{p}|_{u \cap v} \underset{\sim}{<} \underset{\sim}{q}|_{u \cap v}$ iff $\underset{\sim}{x} \underset{\sim}{>} \underset{\sim}{0}$, and $\underset{\sim}{p}|_{u \cap v} = \underset{\sim}{q}|_{u \cap v}$ iff $\underset{\sim}{x} = \underset{\sim}{0}$. In Q2, it is more convenient to work with the single element $\underset{\sim}{x}$ rather than the pair of elements $\underset{\sim}{p}, \underset{\sim}{q}$. We therefore make the following definition.

Definition: For any relevant occurrence pair u, v , the set $\langle\langle u, v \rangle\rangle \subset \underset{\sim}{Z}^{|u \cap v|}$ is defined to be the set of all $\underset{\sim}{x}$ for which there exist $\underset{\sim}{p} \in \mathcal{J}(u)$ and $\underset{\sim}{q} \in \mathcal{J}(v)$ such that:

(a) $\underset{\sim}{x} = \underset{\sim}{q}|_{u \cap v} - \underset{\sim}{p}|_{u \cap v}$,

(b) the execution of u for $\underset{\sim}{p}$ and of v for $\underset{\sim}{q}$ reference the same array element, and

(c) if $u = v$ then $\underset{\sim}{p} \neq \underset{\sim}{q}$.

Using this definition, Q2 can be restated more compactly as follows.

Q3. For any relevant occurrence pair u, v : if $\underset{\sim}{0} \in \langle\langle u, v \rangle\rangle$ then either $u \rightarrow v$ or $v \rightarrow u$.

As an example, consider the relevant occurrence pair b1, b3 in the program of Figure 1. We see from Figure 3 that $b1 \cap b3 = I$. Execution of b1 for $I = i$ references the same element as execution of b3 for $I = i-1$. Hence, $\langle\langle b1, b3 \rangle\rangle = \{(-1)\}$ if there is some i, j, k, ℓ such that $(i, j;) \in \mathcal{J}(b1)$ and $(i-1, \ell; k) \in \mathcal{J}(b3)$; otherwise, $\langle\langle b1, b3 \rangle\rangle = \phi$. Similarly, $\langle\langle b1, b2 \rangle\rangle$ is the set of all elements of the form $(0, y)$ such that there exist i, j with $(i, j;) \in \mathcal{J}(b1)$ and $(i, j+y;) \in \mathcal{J}(b2)$.

The set $\langle\langle u, v\rangle\rangle$ may depend upon the initial values of variables, and so may not be known to the compiler. We therefore assume that for each relevant occurrence pair u, v, we are given a set $\langle u, v\rangle \subset \underset{\text{MM}}{Z}^{|u \cap v|}$ such that $\langle\langle u, v\rangle\rangle \subset \langle u, v\rangle$. The set $\langle u, v\rangle$ is the smallest convenient upper bound which the compiler can place on $\langle\langle u, v\rangle\rangle$. For the program of Figure 1, we could have $\langle b1, b3\rangle = \{(-1)\}$ and $\langle b1, b2\rangle = \{(0, y) : -99 \leq y \leq 99\}$.

Table I contains the sets $\langle u, v\rangle$ which a compiler might construct for the program of Figure 1. We have made very simple choices for these sets. E.g., we ignored the information about $\langle\langle b1, b2\rangle\rangle$ given by the limits of the FOR J statement. This is because the only thing a compiler needs to know about an element of $\langle u, v\rangle$ when applying the coordinate algorithm is which of its components are positive, which are negative, and which are zero.

Finding a set $\langle u, v\rangle$ which contains all possible useful information about $\langle\langle u, v\rangle\rangle$ is crucial to obtaining all possible parallel execution. The problem of computing these sets will not be considered. Some relevant discussion can be found in [4]. A compiler would probably make a good choice for $\langle u, v\rangle$ when the subscript expressions of u and v assume some standard form, and make a simple worst-case approximation otherwise. The sets given in Table I should indicate the type of computations which are needed.

Note: 10. It is clear that $\langle\langle v, u\rangle\rangle = \{-\underset{\text{MM}}{x} : \underset{\text{MM}}{x} \in \langle\langle u, v\rangle\rangle\}$. We may assume that the same relation holds between the sets $\langle u, v\rangle$ and $\langle v, u\rangle$. Thus, Table I specifies all the relevant sets $\langle u, v\rangle$ for the program of Figure 1. End note.

We replace Q3 with the following stronger condition. (It is stronger because it may imply more $\rightarrow$ relations.)

R2. For every relevant occurrence pair u, v : if $\underset{MM}{0} \in \langle u, v \rangle$ then either $u \rightarrow v$ or $v \rightarrow u$.

Condition R2 implies that $\underset{MM}{0} \in \langle v, v \rangle$ for any generation v . This means that the compiler must be able to verify that the assumption made in S2 is satisfied. The restrictions mentioned in Note 8 will usually make this a simple task.

Conditions R1 and R2 determine all the relations $\rightarrow$ which are needed to insure that any execution satisfying PE is a correct execution of the program. Using the $\langle u, v \rangle$ sets of Table I, R1 and R2 imply the relations $\rightarrow$ shown in Table II, where $u \rightarrow v$ is indicated by a $\rightarrow$ in the u-row, v-column. The other entries in Table II are explained later.

The Tree Representation

We have abstracted the following items from a program written in our programming language.

T1. A program tree of occurrences and FOR indices.

T2. For each occurrence v, a specification of the functions of fetched values and index values which determine:

(i) $\mathcal{J}(v)$.

(ii) the value of the subscript expression of v.

(iii) the value to be stored if v is a generation.

T3. For each relevant occurrence pair u, v : a set $\langle u, v\rangle$ which contains $\langle\langle u, v\rangle\rangle$.

T4. A relation $\rightarrow$ on the set of occurrences.

We call these four items a tree representation. Conditions R1 and R2 together with PE imply that the tree representation completely defines the result of executing the program.

Now suppose that we have constructed a tree representation in some arbitrary fashion, not necessarily starting from the programming language representation of a program. Does this tree representation define a program? The answer is yes, if it satisfies R1, R2 and one additional condition given below. However, the program might not be representable in our programming language without adding additional variables.

Part T2 of the tree representation specifies how to determine (i) what fetches and stores are to be executed, (ii) the array elements to be referenced, and (iii) the values to be stored. Conditions R1 and R2 imply that executing

these fetches and stores in any order satisfying PE must produce the same result. All that remains is to require that there exist some execution order which satisfies PE. We know that such an order exists if the tree representation is derived from a program written in our programming language. However, its existence is not implied by R1 and R2 alone, since these conditions allow the possibility that the relations $u \rightarrow v$ and $v \rightarrow u$ both hold for some occurrences u, v .

We must obviously require that the relation $\rightarrow$ be cycle-free -- i.e., that there is no cycle $v_1 \rightarrow v_2 \rightarrow \ldots \rightarrow v_n \rightarrow v_1$. However, we will need a stronger condition. Suppose that in the program tree of Figure 3 we had $b1 \rightarrow b3$ and $b3 \rightarrow b2$. Even though there were no cycles, this would contradict the FOR statement structure implied by the tree. The program tree implies that the FOR J and FOR L statements are disjoint, but the relations $b1 \rightarrow b3 \rightarrow b2$ would imply that $b3$, which appears in the FOR L statement's DO clause, must also be in the FOR J statement's DO clause.

To be able to state the necessary condition precisely, we need some more terminology about trees. For any node ν , we define $\underline{\text{tribe}}(\nu)$ to equal the set of all leaves which are descendants of ν if ν is not a leaf, and to equal $\{\nu\}$ if ν is a leaf. A relation $\rightarrow$ on the set of nodes of a tree is said to be <u>tree complete</u> if it satisfies the following property:

> <u>If</u> μ and ν are cousins, $\mu' \in \underline{\text{tribe}}(\mu)$, $\nu' \in \underline{\text{tribe}}(\nu)$, and $\mu' \rightarrow \nu'$, <u>then</u> $\mu \rightarrow \nu$.

Given a relation $\rightarrow$ on the set of leaves of a tree, its tree completion is defined to be the smallest tree complete relation on the entire set of tree nodes which contains $\rightarrow$. A relation is said to be tree consistent if its tree completion is cycle-free.

An index node I of the program tree represents the DO clause of a FOR I statement. The tree completion of $\rightarrow$ consists of all precedence relations on the set of DO clauses and occurrences implied by the relations $\rightarrow$ between occurrences. This tree completion is cycle-free iff the tree representation defines a program which can be written in terms of the nested FOR statement structure specified by the program tree. We therefore add the following condition.

R3. The relation $\rightarrow$ is tree consistent.

Conditions R1 - R3 guarantee that an arbitrary tree representation defines a valid program.

Notes: 11. If the relation $\rightarrow$ is cycle-free but not tree consistent, then the tree representation still defines a meaningful program. However, this program could not be implemented with simple iterative loops, but would instead require a more complicated and inefficient control structure. We will therefore not consider such programs.

12. We have placed no condition on the SIM index structure corresponding to R3 for the FOR indices. This is because the program can be implemented using

statements for explicitly setting the SIM context, as mentioned in Note 5. These statements can be executed efficiently on most array and vector computers. End notes.

Equivalent Tree Representations

Two tree representations are said to be equivalent if they define programs which produce the same results. An optimization technique like the coordinate method involves transforming a program's tree representation into an equivalent new tree representation. We must therefore develop sufficient conditions for the new tree representation to be equivalent to the original one. We will use the symbols $Z(\ )$, $\mathcal{J}(\ )$, $<<\ >>$, $<\ >$, $\cap$, and $\rightarrow$ in regular type when they refer to the original representation, and in boldface type [denoted in manuscript by wavy underlining] when they refer to the new representation.

We only consider the case in which the two representations have the same set of occurrences. We will guarantee equivalence by requiring that the new program executes the same operations, using the same values, as the original one. This means that for each occurrence u and for each element $\underset{\sim}{p} \in \mathcal{J}(u)$ for which u is executed in the original program, there must be a corresponding element $\psi_u(\underset{\sim}{p}) \in \underset{\sim}{\mathcal{J}}(u)$ for which u is executed in the new program. Moreover, the functions specified by T2 of the new tree representation must be the same as for the original representation, except with each array reference for an element

$\underset{\sim}{p} \in \mathcal{J}(u)$ replaced by the corresponding reference for $\psi_u(\underset{\sim}{p}) \in \underset{\sim}{\mathcal{J}(u)}$. We cannot state this condition more precisely without giving a precise definition of the specification mentioned in T2, but its meaning should be clear. We are thus led to the following condition.

> E1. For each occurrence u , there is a 1-1 correspondence $\psi_u : Z(u) \rightarrow \underset{\sim}{Z(u)}$ such that the mappings ψ_u transform T2 of the original tree representation into T2 of the new tree representation.

As an illustration, consider the original program of Figure 4 and the equivalent new program of Figure 5 obtained by replacing the index K with the index $K' = J+K$. For each occurrence u inside the FOR K statement's DO clause, the mapping $\psi_u : \underset{\sim}{Z^2} \times \underset{\sim}{Z^0} \rightarrow \underset{\sim}{Z^2} \times \underset{\sim}{Z^0}$ is given by $\psi_u(j, k;) = (j, j+k;)$. Executing this occurrence in the original program for the element $(j, k;) \in \mathcal{J}(u)$ [i.e., with $J = j$ and $K = k$] references the same array element as executing it in the new program for the element $(j, j+k;) \in \underset{\sim}{\mathcal{J}(u)}$ [with $J = j$ and $K' = j+k$] . Observe how the limits of the FOR K' statement were chosen so that $\underset{\sim}{\mathcal{J}(u)} = \psi_u(\mathcal{J}(u))$. This example should clarify the meaning of E1.

Since both tree representations must satisfy R1 - R3, the only requirement other than E1 needed to guarantee their equivalence is that the relevant ordering of references to the same array element should be the same. In other words, Q1 must be satisfied -- except that the ordering must now be preserved when translating from the original tree representation to the new one, rather than from the programming language version to its tree representation as before.

The element $\underset{\sim}{x} = \underset{\sim}{q}|_{u \cap v} - \underset{\sim}{p}_{u \cap v} \in \langle\langle u, v\rangle\rangle$ has a corresponding element $\psi_{u, v}(\underset{\sim}{x}) = \psi_v(\underset{\sim}{q})|_{u \cap v} - \psi_u(\underset{\sim}{p})|_{u \cap v} \in \langle\langle\langle u, v\rangle\rangle\rangle$. This defines a mapping $\psi_{u, v}$ from $\langle\langle u, v\rangle\rangle$ onto $\langle\langle\langle u, v\rangle\rangle\rangle$. Since we could also have $\underset{\sim}{x} = \underset{\sim}{q}'|_{u \cap v} - \underset{\sim}{p}'|_{u \cap v}$ for a different pair $\underset{\sim}{p}'$, $\underset{\sim}{q}'$, the mapping $\psi_{u, v}$ may be multiple valued.

The same type of reasoning that led us before from Q1 to R2 now leads us from Q1 to the following condition.

E2. For any relevant occurrence pair, let $\psi_{u, v} : \underset{\sim}{Z}^{|u \cap v|} \to \underset{\sim}{Z}^{|u \cap v|}$ be the (possibly multiple valued) mapping defined by $\underset{\sim}{x}' = \psi_{u, v}(\underset{\sim}{x})$ iff there exist $\underset{\sim}{p} \in Z(u)$ and $\underset{\sim}{q} \in Z(v)$ such that $\underset{\sim}{x} = \underset{\sim}{q}|_{u \cap v} - \underset{\sim}{p}|_{u \cap v}$ and $\underset{\sim}{x}' = \psi_v(\underset{\sim}{q})|_{u \cap v} - \psi_u(\underset{\sim}{p})|_{u \cap v}$. Then:

(a) $\langle u, v\rangle = \psi_{u, v}(\langle u, v\rangle)$, and

(b) if $\underset{\sim}{x} \in \langle u, v\rangle$ and either

(i) $\underset{\sim}{x} > \underset{\sim}{0}$, or

(ii) $\underset{\sim}{x} = \underset{\sim}{0}$ and $u \to v$;

then for each $\psi_{u, v}(\underset{\sim}{x})$ either

(i) $\psi_{u, v}(\underset{\sim}{x}) > \underset{\sim}{0}$, or

(ii) $\psi_{u, v}(\underset{\sim}{x}) = \underset{\sim}{0}$ and $u \to v$.

Conditions E1 and E2 are sufficient to guarantee that the new tree representation is equivalent to the original one.

The Coordinate Algorithm

The heart of the coordinate method is the coordinate algorithm, which essentially attempts to convert a single FOR I statement into a SIM FOR ALL I statement. More precisely, given a tree representation with an index node I , the coordinate algorithm attempts to transform it into an equivalent representation in which I becomes a SIM index and is thus removed from the program tree. If this is not possible, then I may be transformed into one or more different nodes $I^{(k)}$. Each occurrence which was originally a descendant of I will either become a descendant of some $I^{(k)}$ or else will be executed with I as a SIM index. Hence, the original FOR I statement is split into disjoint FOR $I^{(k)}$ statements and one or more SIM statements.

Figure 6 shows such a transformed version for the tree of Figure 3, in which I has been transformed into a new node $I^{(2)}$. The occurrences which were originally descendants of I but are not descendants of $I^{(2)}$ in the new tree will be executed in parallel for all values of I . Observe that it was necessary to make $I^{(2)}$ a descendant of J in order to preserve the tree structure, even though J was originally a descendant of I . In Figure 2, we have translated the new tree representation into our programming language. Observe that it was necessary to introduce a new variable T and an extra occurrence of R . The new tree representation cannot be obtained directly from any program written in our programming language.

For the transformation generated by the coordinate algorithm, it is clear what the mappings ϕ_u are. For any $\underset{\sim}{x} \in Z^m$, we define $\tilde{\underset{\sim}{x}}$ to be the element of

$\underset{\sim}{Z}^m$ obtained by moving the $|I|^{\underline{th}}$ coordinate of $\underset{\sim}{x}$ to the right $|I^{(k)}| - |I|$ positions, and we define $\hat{\underset{\sim}{x}}$ to be the element of $\underset{\sim}{Z}^{m-1}$ obtained by deleting the $|I|^{\underline{th}}$ coordinate of $\underset{\sim}{x}$. We can then define the mapping ψ_u as follows.

<u>if</u> u is not a descendant of I in the original tree

<u>then</u> $\psi_u(\underset{\sim}{p}_1 ; \underset{\sim}{p}_2) = (\underset{\sim}{p}_1 ; \underset{\sim}{p}_2)$

<u>else if</u> u is a descendant of some $I^{(k)}$ in the new tree

<u>then</u> $\psi_u(\underset{\sim}{p}_1 ; \underset{\sim}{p}_2) = (\tilde{\underset{\sim}{p}}_1 ; \underset{\sim}{p}_2)$

<u>else</u> $\psi_u(\underset{\sim}{p}_1 ; \underset{\sim}{p}_2) = (\hat{\underset{\sim}{p}}_1 ; \underset{\sim}{p}'_2)$,

where $\underset{\sim}{p}'_2$ is formed by taking $\underset{\sim}{p}_2$ and adding an extra coordinate equal to the $|I|^{\underline{th}}$ coordinate of $\underset{\sim}{p}_1$.

This definition of ψ_u implies that $\psi_{u,v} : \underset{\sim}{Z}^{|u \cap v|} \to \underset{\sim}{Z}^{|u \cap v|}$ is the single valued mapping defined as follows.

<u>if</u> u, v are not both descendants of I in the original tree

<u>then</u> $\psi_{u,v}(\underset{\sim}{x}) = \underset{\sim}{x}$

<u>else if</u> u, v are both descendants of some single $I^{(k)}$

<u>then</u> $\psi_{u,v}(\underset{\sim}{x}) = \tilde{\underset{\sim}{x}}$

<u>else</u> $\psi_{u,v}(\underset{\sim}{x}) = \hat{\underset{\sim}{x}}$

The first step of the coordinate algorithm is to examine all relevant occurrence pairs u, v to determine what E2 implies about the new tree repre-

sentation. E2 might imply that if u and v are not both descendants of a single $I^{(k)}$ in the new tree, then $u \underset{\mathbf{M}}{\rightarrow} v$ must hold. This is noted by writing $u \dashrightarrow v$. Condition E2 might instead imply that u and v must both be descendants of some single $I^{(k)}$. In this case, there will be some descendant J of I which is an ancestor of both u and v such that J must be a descendant of $I^{(k)}$ in the new tree. This node J will be called a blocking node.

Combining the definition of $\psi_{u, v}$ with E2 gives us the following algorithm step for constructing the relation $\dashrightarrow$ and the set of blocking nodes.

C1. For every relevant occurrence pair u, v such that u and v are both descendants of I , and for every $\underset{\mathbf{M}}{x} \in \langle u, v \rangle$ such that $\underset{\mathbf{M}}{x} > \underset{\mathbf{M}}{0}$:

(i) if $\hat{\underset{\mathbf{M}}{x}} = \underset{\mathbf{M}}{0}$ then add the relation $u \dashrightarrow v$;

(ii) if $\hat{\underset{\mathbf{M}}{x}} < \underset{\mathbf{M}}{0}$ then the descendant J of I such that

(a) J is an ancestor of both u and v , and

(b) the $|J|^{\underline{th}}$ coordinate of $\underset{\mathbf{M}}{x}$ is its left-most negative coordinate

is designated a blocking node.

Applying C1 to the program tree of Figure 6, using the $\langle u, v \rangle$ sets of Table I, gives the relations $\dashrightarrow$ indicated in Table II. There are no blocking nodes. Observe that the elements $\underset{\mathbf{M}}{x} \in \langle u, v \rangle$ with $\underset{\mathbf{M}}{x} < \underset{\mathbf{M}}{0}$ are considered when C1 is applied to the pair v, u . (See Note 10.)

Note: 13. Suppose that the SIM FOR ALL K statement in Figure 1 were replaced by a FOR K statement. We would then have $\langle c1, c2 \rangle = \{(1, -1, 3)\}$, and appli-

cation of C1 to the pair c1, c2 would designate K as a blocking node.

End note.

The second step in the algorithm is the crucial one which determines the structure of the new program tree. Before stating it, we need some final notation. A forest is a collection of disjoint trees. The definition of tree consistency applies equally well to a forest. A terminal forest of a tree $\mathcal{T}$ is a non-empty set F of nodes of $\mathcal{T}$, not containing the root, such that (i) if $\nu \in F$ then every descendant of ν is in F, and (ii) if $\nu \notin F$ then F cannot contain both a cousin and a descendant of ν. A pruning of a tree $\mathcal{T}$ consists of a collection $\rho = \{F_1, \ldots, F_s\}$ of pairwise disjoint terminal forests of $\mathcal{T}$ whose union contains all the leaves of $\mathcal{T}$. The pruning ρ defines the pruned tree $\mathcal{T}_\rho$ whose leaves are $F_1, \ldots F_s$ and whose non-leaf nodes are the nodes of $\mathcal{T}$ which are not in any F_k. The pruned tree has the obvious tree structure, where a node is an ancestor of F_k iff it is not in F_k and is an ancestor of some node in F_k. A relation $\Rightarrow$ on the leaves of $\mathcal{T}$ induces a relation on the leaves of $\mathcal{T}_\rho$ defined by $F_k \Rightarrow F_\ell$ iff $k \neq \ell$ and $u \Rightarrow v$ for some $u \in F_k$, $v \in F_\ell$.

The next step is to use the information gathered by step C1 to determine the new program tree so that E2 and R3 are satisfied. This is done by constructing a pruning $\{F_1, \ldots, F_s\}$ of the subtree rooted by I, and designating some of the F_k to be sequential. A node will become a descendant of $I^{(k)}$ in the new tree iff it is in the sequential element F_k. The nodes in a non-sequential F_k will appear inside a SIM statement. The precise statement of

the second step of the coordinate algorithm is as follows.

C2. Let $\Rightarrow$ denote the union of the relations $\rightarrow$ and $\dashrightarrow$, and let $\mathcal{T}$ be the tree consisting of I and its descendants.

(1) Construct a pruning $\rho = \{F_1, \ldots, F_s\}$ of $\mathcal{T}$ such that:

(i) every blocking node is in some F_k, and

(ii) the relation induced by $\Rightarrow$ on $\mathcal{T}_\rho$ is tree consistent.

(2) Designate certain elements of ρ to be sequential, where F_k is sequential if (but not necessarily only if):

(i) it contains a blocking node, or

(ii) the relation $\Rightarrow$ on the forest F_k is not tree consistent.

Such pruning ρ always exists: namely, the trivial one consisting of the single element containing all descendants of I. However, if this element is sequential, then the new program tree will be the same as the old one, and no parallel execution is achieved. The choice of the best pruning will be discussed later. Figure 7 shows a pruned tree constructed by C2 for the program tree of Figure 3 and the relations of Table II. The only sequential element is {a1, a2 a3, g}. This pruning yields the new program tree of Figure 6.

The final step of the coordinate algorithm constructs the new tree representation in the obvious way.

C3. The new tree representation is constructed as follows.

(1) The new program tree consists of the same nodes as the original tree except without I and with a new index node $I^{(k)}$ for every sequential element F_k in ρ. A node μ is an ancestor of a node ν in the new tree iff either

(a) $\mu = I^{(k)}$ and $\nu \in F_k$, or

(b) μ is not one of the $I^{(k)}$ and either

(i) μ is an ancestor of ν in the original tree, or

(ii) $\nu = I^{(k)}$ and μ is either an ancestor of I in the old tree, or an ancestor of F_k in $\mathcal{T}_\rho$.

(2) Part T2 is the transformation of T2 of the original representation obtained from the mappings ψ_u defined above.

(3) For each relevant occurrence pair u, v : $\underset{MM}{\langle u, v \rangle} = \psi_{u, v}(\langle u, v \rangle)$.

(4) The relation $\underset{MM}{\rightarrow}$ is defined by $u \underset{MM}{\rightarrow} v$ iff

(a) $u \rightarrow v$, or

(b) $u \dashrightarrow v$ and u and v are either

(i) in different elements of ρ, or

(ii) in the same non-sequential element of ρ.

Applying step C3 to the pruning shown in Figure 7 yields the tree representation used to construct the program of Figure 2. Observe that each non-sequential F_k becomes the set of occurrences in the DO clause of a SIM $I^{(k)}$ statement.

To demonstrate the correctness of the coordinate algorithm, we must first show that the new representation satisfies R1 - R3, so it defines a valid program, and then show that E1 and E2 are satisfied.

Proof of R1: The definition of the mappings ψ_u imply that if $\underset{\sim}{p}|_{u \cap v} = \underset{\sim}{q}|_{u \cap v}$ then $\psi_u(\underset{\sim}{p})|_{u \cap v} = \psi_v(\underset{\sim}{q})|_{u \cap v}$ for any $\underset{\sim}{p} \in Z(u)$, $\underset{\sim}{q} \in Z(v)$. Condition R1 for the new representation then follows easily from C3(2), C3(4)(a), and the fact that the original representation satisfies R1.

Proof of R2: Let u, v be a relevant occurrence pair. The definition of $\psi_{u, v}$ shows that $\psi_{u, v}(\underset{\sim}{0}) = \underset{\sim}{0}$. By C3(4) and the fact that R2 holds for the original representation, we need only show that if $\underset{\sim}{0} \in \langle u, v \rangle$ and $\underset{\sim}{0} \notin \langle u, v \rangle$ then $u \underset{\sim}{\rightarrow} v$ or $v \underset{\sim}{\rightarrow} u$. Recalling Note 10, it thus suffices to prove that if $\underset{\sim}{x} \in \langle u, v \rangle$, $\underset{\sim}{x} > \underset{\sim}{0}$ and $\psi_{u, v}(\underset{\sim}{x}) = \underset{\sim}{0}$, then $u \underset{\sim}{\rightarrow} v$. The definition of $\psi_{u, v}$ shows that if $\underset{\sim}{x} > \underset{\sim}{0}$ and $\psi_{u, v}(\underset{\sim}{x}) = \underset{\sim}{0}$, then u and v are not both descendants of a single $I^{(k)}$ and $\psi_{u, v}(\underset{\sim}{x}) = \hat{\underset{\sim}{x}}$. By C1 and C3(4), this implies that $u \underset{\sim}{\rightarrow} v$.

Proof of R3: Consider the terminal forest of the new program tree consisting of the $I^{(k)}$ and all the former descendants of I , and let $\mathcal{T}'$ be the tree formed by adding I as the root. Since $u \underset{\sim}{\rightarrow} v$ and $u \not\underset{\sim}{\rightarrow} v$ only if u and v are both descendants of I in the original tree, it suffices to show that the relation $\underset{\sim}{\rightarrow}$ is tree consistent on $\mathcal{T}'$. Let ρ' be any pruning of $\mathcal{T}'$. It is easy to show that $\underset{\sim}{\rightarrow}$ is tree consistent on $\mathcal{T}'$ iff (a) it is tree consistent on each element of ρ' [remember that the elements of ρ' are terminal subforests of $\mathcal{T}'$], and (b) it induces a tree consistent relation on $\mathcal{T}'_{\rho'}$. Let $\rho' = \{F'_1, \ldots, F'_s\}$, where

$F'_k = F_k \cup \{I^{(k)}\}$ if F_k is sequential, and $F'_k = F_k$ otherwise. Then $\mathcal{T}'_{\mathcal{P}'} = \mathcal{T}_{\mathcal{P}}$, and tree consistency follows easily from C2, C3(4) and the fact that the relation $\rightarrow$ is tree consistent on the original program tree, and hence on each F'_k.

Proof of E1: Condition E1 follows trivially from the definition of ψ_u and step C3(2).

Proof of E2: Let u, v be a relevant occurrence pair and $\underset{\sim}{x} \in \langle u, v \rangle$. We showed in the proof of R2 that (i) if $\underset{\sim}{x} = \underset{\sim}{0}$ and $u \rightarrow v$ then $\psi_{u,v}(\underset{\sim}{x}) = \underset{\sim}{0}$ and $u \underset{\sim}{\rightarrow} v$, and (ii) if $\underset{\sim}{x} > \underset{\sim}{0}$ and $\psi_{u,v}(\underset{\sim}{x}) = \underset{\sim}{0}$ then $u \underset{\sim}{\rightarrow} v$. All that is left for us to show is that it is impossible to have $\underset{\sim}{x} > \underset{\sim}{0}$ and $\psi_{u,v}(\underset{\sim}{x}) < \underset{\sim}{0}$. The definition of $\psi_{u,v}$ implies that if $\underset{\sim}{x} > \underset{\sim}{0}$ and $\psi_{u,v}(\underset{\sim}{x}) < \underset{\sim}{0}$ then $\hat{\underset{\sim}{x}} < \underset{\sim}{0}$. Let J be the blocking node obtained by C1(ii) for $\underset{\sim}{x}$. Steps C2 and C3(1) imply that J must be a descendant of some $I^{(k)}$, and the definition of $\psi_{u,v}$ then implies that $\psi_{u,v}(\underset{\sim}{x}) > \underset{\sim}{0}$. This completes the correctness proof of the coordinate algorithm.

Notes: 14. Each non-sequential F_k in the pruning can form the DO clause of a SIM FOR ALL $I^{(k)}$ statement. One can show that this statement can be strip mined by writing it in the form

$$\underset{\sim}{\text{FOR}}\ J := 1\ \underset{\sim}{\text{TO}}\ n\ \underset{\sim}{\text{DO}}\ \underset{\sim}{\text{SIM FOR ALL}}\ I^{(k)} \in \mathcal{S}_J \ \ldots\ \underset{\sim}{\text{OD}}$$

as discussed in Note 7.

15. Had we permitted FOR statements with arbitrary constant increments, rather than just the increment 1, then PE would no longer hold. However, all of our results would remain valid by changing the definition of $\langle\langle u, v \rangle\rangle$ as follows. Let $\vec{I}_m$ be the FOR multi-index of u, let $u \cap v = I_k$, and let d_j denote the increment of the FOR I_j statement, $j = 1, \ldots, m$. Then we need only change part (a) of the definition of $\langle\langle u, v \rangle\rangle$ to the following:

(a') $\underset{\sim}{q}|_{u \cap v} - \underset{\sim}{p}|_{u \cap v} = \overrightarrow{d_k x_k}$ and $\underset{\sim}{x} = \vec{x}_k$. End notes.

The Coordinate Method

The coordinate method consists of applying the coordinate algorithm successively to different index nodes of the tree representation. We could easily generalize the coordinate algorithm so it can be applied to several indices at once, rather than to just a single index. However, this general algorithm would yield the same results as repeated application of the single index algorithm. This is obvious if the index nodes are all cousins. Suppose K is a descendant of I. Any relation $\dashrightarrow$ or blocking node generated by applying the coordinate algorithm to K will also be generated by applying the generalized algorithm to I and K together. It follows from this that any tree representation obtained by applying the generalized algorithm to I and K can be obtained by first applying the coordinate algorithm to K and then applying it to I.

In general, the coordinate algorithm should be applied to an index before

it is applied to any of its ancestors. This order of application is important. The program of Figure 1 could have been obtained by applying the coordinate algorithm to a purely sequential program with a FOR K node. The final version of Figure 2 could not have been obtained from this sequential program had we first applied the coordinate algorithm to I and then to K . (See Note 13.)

Implementing the coordinate method in a compiler involves many practical details. A complete discussion of these details is beyond the scope of this paper. We briefly discuss some of them below.

Choice of indices: Choosing the indices to which the coordinate method should be applied can be a difficult problem. Parallel execution may place certain requirements on how arrays are stored. This means that the choice of whether to apply the coordinate algorithm to a particular FOR index must be based upon an examination of the entire program, even though the coordinate algorithm only uses that FOR statement itself. This choice could be made either by a suitable heuristic algorithm, or else by the user in an interactive system.

Step C2: Choosing the pruning in C2 is an important problem. It is easy to construct a pruning which yields the maximum possible parallelism. More precisely, let $\overset{*}{\Rightarrow}$ denote the tree completion of the relation $\Rightarrow$ on the tree $\mathcal{T}$. Then ρ can be chosen so that an occurrence u is in a sequential F_k iff there is a node μ such that $u \in \underline{\text{tribe}}(\mu)$ and μ is either a blocking node or else is contained in a cycle $\mu \overset{*}{\Rightarrow} \ldots \overset{*}{\Rightarrow} \mu$. However, maximizing parallelism will not always produce an optimal program for a parallel computer. The use g in our example could have

been executed within a SIM I statement. However, all that this parallel execution would do is put the values into temporary registers from which they must later be fetched sequentially. The optimal choice for ρ will depend heavily upon the details of the computer on which the program is to be executed, and finding it would be a very difficult task. However, it should not be too hard to devise a heuristic algorithm that will usually yield a good pruning, and will never produce a less efficient program than the original.

Scalar variables: It is clear that a generation of a scalar variable cannot be placed inside a SIM FOR ALL statement. Moreover, if an index node J is a son of I which contains such a generation, then J will be a blocking node and none of its descendants can be place inside a SIM FOR ALL. It is therefore very important to try to remove such a scalar variable from the FOR I statement before applying the coordinate algorithm. There are two common cases where this is easily done.

(a) The value of the variable is a function of the values of the indices. The variable is then eliminated by direct substitution. An example of this appears on pages 92-93 of [4].

(b) The variable holds the result of an intermediate calculation. For example suppose the following sequence of assignment statements appear in a FOR I statement:

```
X[ ] := A[ I ]**2 ; B[ I ] := 2*X[ ] ; C[ I ] := 1/ X[ ]   ,
```

and the value of X[ ] is not used elsewhere. These occurrences

of X can be eliminated when forming the tree representation of the program. Part T2 of the representation can specify the relations between the value fetched from A[I] and the values stored into B[I] and C[I] .

Other methods of parallel execution: Each FOR $I^{(k)}$ statement implies a certain amount of additional index computation, so the number of different sequential elements F_k of the pruning should usually be kept as small as possible without reducing the amount of parallel execution. However, a FOR statement whose DO clause contains a single assignment statement can sometimes be executed very efficiently on an array computer using Kogge's recursive doubling technique [2] or on a vector computer using special vector operations. For example, the statement

FOR I := 1 TO n DO X[ ] := X[ ] + A[I]*B[I] OD

can be executed on an array computer in $\log_2 n$ steps, and on a vector computer with a single vector inner product operation. If it is possible for the pruning to include a sequential F_k containing a single generation, then the compiler should see if the resulting FOR $I^{(k)}$ statement can be executed efficiently by one of these methods.

The Hyperplane Method

The hyperplane method described in [4] can yield parallel execution not

obtainable by the coordinate method. It was restricted to a "tight nest" of FOR statements of the form

```
FOR I1 := λ1 TO μ1
    DO FOR I2 := λ2 TO μ2
           DO
            .
             .
              .
                FOR In := λn TO μn
                    DO S OD
                     .
                      .
                       .
                                  OD
```

with $n \geq 2$. By using index variable substitutions, the coordinate algorithm can provide a generalization of the hyperplane method of [4]. This is illustrated with the program of Figure 4. The coordinate algorithm applied to J or K cannot obtain any significant parallel execution of the FOR K statement's DO clause. Replacing the index K by the new index K' = J+K transforms this program into the one of Figure 5. Applying the coordinate algorithm to the index J in this new program, we find that the program can be rewritten by replacing the FOR J with a SIM FOR ALL J. Hence, the index substitution permits the parallel execution of the innermost DO clause.

As another example, consider the rewriting performed with the hyperplane method on pages 83-84 of [4]. Essentially the same rewriting can be obtained as follows: (1) replace K by K' = J+K , (2) apply the coordinate algorithm to the index J , (3) replace K' by K'' = K' + 2I , (4) apply the coordinate algorithm to the index I . A more complete discussion of the generalized hyperplane method is beyond the scope of this paper.

Conclusion

The coordinate method transforms a sequential program into a parallel one for execution on an array or vector computer. It employs the coordinate algorithm, which converts a sequential FOR statement into a single SIM statement or into several FOR and SIM statements. This coordinate method generalizes the one described in [4]. By introducing index substitutions, it can also generalize the hyperplane method of [4].

We have not considered the cost of applying the coordinate method when compiling a program. Except for the sets $\langle u, v \rangle$ and the relation $\rightarrow$, the tree representation will be similar to the internal representation into which the compiler would normally translate the program. Each outer level FOR statement can be treated separately, and these statements are seldom very long. Hence, computing the sets $\langle u, v \rangle$ and the relation $\rightarrow$, and performing steps C1 and C3 of the coordinate algorithm should not require an excessive amount of computation. Choosing the indices to which the coordinate algorithm is applied, and choosing the pruning

in step C2 cannot be done by exhaustive searching, so heuristic methods must be developed.

Although we have only discussed array and vector computers, the coordinate algorithm and the techniques used to derive it have other applications to compiling. We list some of these below.

Pipelined sequential computers: It has been found that execution on a pipelined sequential computer such as the CDC 7600 can be speeded up by re-writing the program in terms of vector operations, and then executing these operations with carefully optimized subroutines [1]. The coordinate method can be used to perform this rewriting, since it generates vector operations. However, the sequential computer does not actually require vector operations, but rather very short FOR statements which can be executed efficiently by exploiting the pipelining. The coordinate algorithm can be made to produce such FOR statements by using the appropriate criteria to choose the pruning in step C2.

Compilers for array computers: Producing efficient code for an array computer from a parallel program is a non-trivial problem if the SIM statements specify more parallel execution than is actually possible with the fixed number of processors. A special case of this is compiling a parallel program for execution on an ordinary sequential computer. Our method of analysis can be used to solve this problem. (See [6].)

Multiple instruction stream computers: The tree representation of a program suppresses the unnecessary ordering between executions. Let $\overset{*}{\rightarrow}$ denote the tree completion of the relation $\rightarrow$. If index nodes I and J are cousins, then the execution of the FOR I loop must precede the execution of the FOR J loop iff $I \overset{*}{\rightarrow} J$. If neither $I \overset{*}{\rightarrow} J$ nor $J \overset{*}{\rightarrow} I$ holds, then the two FOR statements can be executed concurrently (for a single iteration of any FOR statement's DO clause containing them both). Thus, the ordering $\overset{*}{\rightarrow}$ can be used to generalize the techniques described in [8] for parallel execution by a multiple instruction stream computer. If each instruction stream functions like an ordinary sequential computer, then the tree representation of the original program is used. If an individual instruction stream can perform parallel operations -- for example, in a "multiple pipe" vector computer or a multiple quadrant ILLIAC IV -- then the coordinate method is first applied to introduce parallel execution into the tree representation.

Virtual memory: Reducing page faults in a virtual memory system may require changing the order of execution of operations -- for example, changing a FOR I/FOR J nest to a FORJ/FOR I nest. The method of analysis we have developed can be used to determine when such a rewriting is possible.

In general, the tree representation is useful in implementing any compiler optimization technique which exploits the FOR statement structure of the program.

References

[1] Buzbee, B.L., and Rudsinski, L.E. Exploiting vector mode in an SISD computer. Proc. 1975 Sagamore Comput. Conf. on Parallel Processing, Syracuse University, p. 251.

[2] Kogge, P.M. Parallel solution of recurrence problems. IBM J. Res. and Devel., March 1974, pp. 138-148.

[3] Kuck, David J. Parallel processor architechture - a survey. Proc. 1975 Sagamore Comput. Conf. on Parallel Processing, Syracuse University, pp. 15-39.

[4] Lamport, Leslie. The parallel execution of DO loops. Comm. ACM 17, 2 (February 1974), pp. 83-93.

[5] Lamport, Leslie. The coordinate method for the parallel execution of DO loops. Proc. 1973 Sagamore Comput. Conf. on Parallel Processing, Syracuse University, pp. 1-12.

[6] Lamport, Leslie. Parallel execution on array and vector computers. Proc. 1975 Sagamore Comput. Conf. on Parallel Processing, Syracuse University, pp. 187-191.

[7] Muroaka, Yoishi. Parallelism exposure and exploitation in programs. Ph.D. Th., U. of Illinois, Urbana, Illinois, 1971.

[8] 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, New Jersey, pp. 1-15.

[9] Schneck, Paul B. Automatic recognition of parallel/vector operations in a higher level language. Proc. ACM 1972 National Conference, pp. 772-779.

[10] Schneck, Paul B. Movement of implicit parallel and vector expressions out of program loops. Proc. Conf. on Programming Languages and Compilers for Parallel and Vector Machines. Sigplan Notices 10, 3 (March 1975), pp. 103-106.

[11] Wedel, Dorothy. Fortran for the Texas Instrument ASC system. Proc. Conf. on Programming Languages and Compilers for Parallel and Vector Machines. Sigplan Notices 10, 3 (March 1975), pp. 119-132.

[12] Cohagan, W. L. Vector Optimization for the ASC. Proc. 7th Annual Princeton Conf. on Information Science Systems. Dept of Elec. Eng., Princeton Univ. (1973), pp 169-174.

```
FOR I := 1 TO R[ ]
          (r)

    DO FOR J := 1 TO 100

        DO IF A[ J ] > 0 THEN B[ I ] := B[ I ] + F[ I, J ]
             (a1)          (b1)    (b2)    (f1)
           ELSE A[ J ] := A[ J ] + G[ I, J ] FI
                (a2)     (a3)     (g)

        OD ;

    SIM FOR ALL K ε { k : 0 ≤ k ≤ 63 and F[ I, k ] > 0 }
                                        (f2)

        DO FOR L:= 3 TO 9

            DO C[ I, K, L ] := C[ I-1, K+1, L-3 ]*B[ I+1 ] OD
                 (c1)              (c2)            (b3)

        OD

OD
```

Figure 1

```
S := { i : 1 ≤ i ≤ R[ ] }

SIM FOR ALL I(1) ε S

    DO SIM FOR ALL K ε { k : 0 ≤ k ≤ 63 and F[ I, k ] > 0 }
                                                    (f2)

            DO FOR L := 3 TO 9

                    DO C[ I(1), K, L ] := C[ I(1)-1, K+1, L-3 ]*B[ I(1)+1 ] OD
                       (c1)                (c2)                 (b3)

            OD

    OD ;

FOR J := 1 TO 100

    DO FOR I(2) := 1 TO R[ ]

            DO T[ I(2) ] := (A[ J ] > 0) ;
                            (a1)

                IF T[ I(2) ] THEN

                        ELSE A[ J ] := A[ J ] + G[ I(2), J ] FI
                             (a2)     (a3)      (g)

            OD ;

        SIM FOR ALL I(3) ε S

            DO IF T[ I(3) ] THEN B[ I(3) ] := B[ I(3) ] + F[ I(3); J ]
                                  (b1)       (b2)          (f1)

                       ELSE                                        FI

            OD

    OD
```

Figure 2

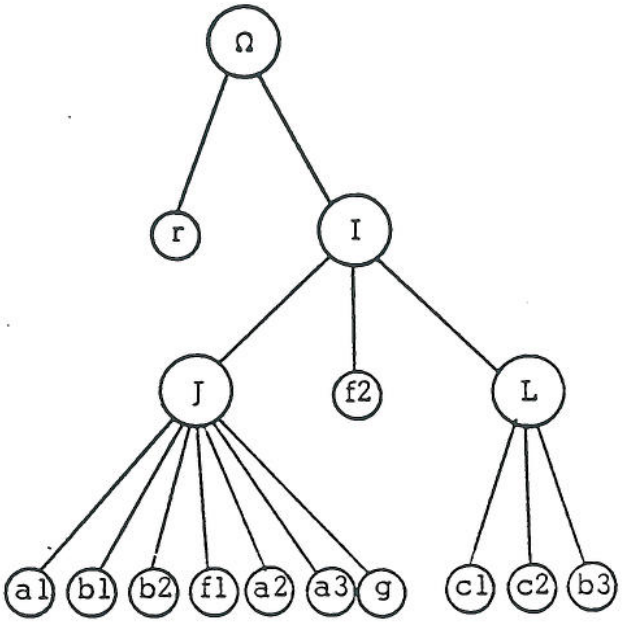

Figure 3

```
FOR J := 1 TO M[]

    DO B[J] := .25*(1 - A[J]) ;

        FOR K := 1 TO N[]

            DO U[J, K] := A[J]*U[J, K] + B[J]*
                          (U[J+1, K] + U[J, K+1]
                           + U[J-1, K] + U[J, K-1])

            OD

    OD
```

Figure 4

```
FOR J := 1 TO M[]

    DO B[J] := .25*(1 - A[J]) ;

        FOR K' := 1+J TO N[]+J

            DO U[J, K'-J] := A[J]*U[J, K'-J] + B[J]*
                             (U[J+1, K'-J] + U[J, K'-J+1]
                              + U[J-1, K'-J] + U[J, K'-J-1])

            OD

    OD
```

Figure 5

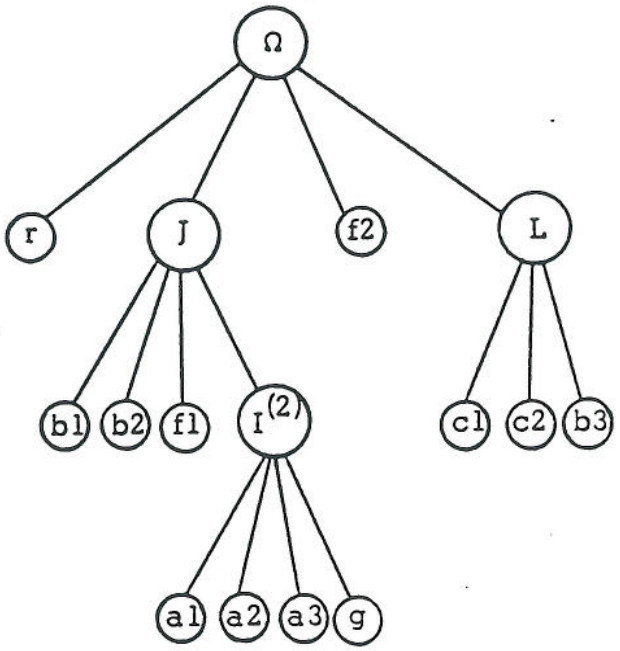

Figure 6

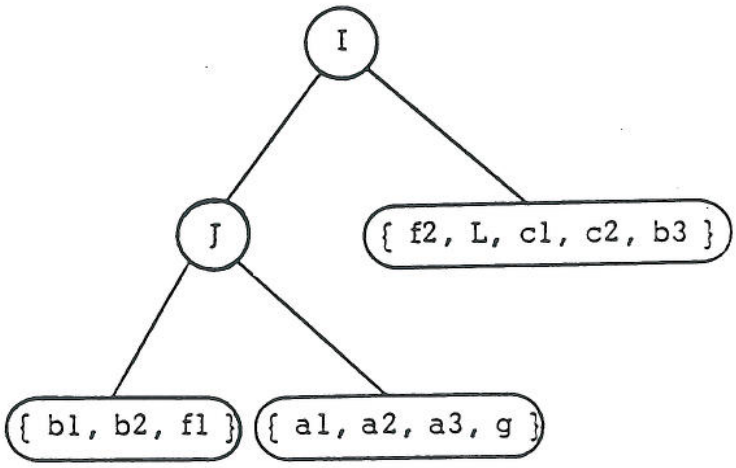

Figure 7

$\langle a1, a2 \rangle = \{ (x, 0) : x \text{ any integer} \}$

$\langle a2, a2 \rangle = \{ (x, 0) : x \neq 0 \}$

$\langle a2, a3 \rangle = \{ (x, 0) : x \text{ any integer} \}$

$\langle b1, b1 \rangle = \{ (0, y) : y \neq 0 \}$

$\langle b1, b2 \rangle = \{ (0, y) : y \text{ any integer} \}$

$\langle b1, b3 \rangle = \{ (-1) \}$

$\langle c1, c1 \rangle = \emptyset$

$\langle c1, c2 \rangle = \{ (1, 3) \}$

Table I

	a1	a2	a3	b1	b2	b3	c1	c2	f1	f2	g	r
a1		→ →	→	→	→				→		→	
a2	--→	--→	--→									
a3		→ --→										
b1												
b2				→								
b3				--→			→					
c1												
c2							→					
f1				→								
f2						→	→	→				
g		→										
r	→	→	→	→	→	→	→	→	→	→	→	

Table II