Yale University
Department of Computer Science

Subdomain Dependence Decision Algorithm for Massively Parallel Computing

Lee-Chung Lu and Marina C. Chen

YALEU/DCS/TR-772
April, 1990

This work has been supported in part by the Office of Naval Research under Contract N00014-86-K-0310, N00014-86-K-0564 and N00014-89-J-1906.

Subdomain Dependence Decision Algorithm for Massively Parallel Computing

Lee-Chung Lu Marina C. Chen

Department of Computer Science
Yale University
P.O. Box 2158 Yale Station
New Haven, CT 06520
lu-lee-chung@yale.edu chen-marina@yale.edu

April, 1990

Abstract

The dependence analysis and testing problem has been well studied in parallelizing compilers. All of the existing dependence decision algorithms, however, are conservative in considering data dependence between statements occurring in conditional branches. By conservative we mean that a true dependence does not exist due to the presence of the conditional yet the dependence test may come out positive. We propose a new decision algorithm which can obtain more accurate dependence information when statements have conditionals. Experimently, this decision algorithm is quite efficient. This decision algorithm can also be used in cycle breaking and in testing dependences for functional programs.

1 Introduction

The grand challenge of trans-teraflop computing requires innovative software technologies for realizing the full capability of the new generation of massively parallel computers. By now it is widely recognized that the direct programming of this class of machines by explicit communication or synchronization can be tedious, error-prone, and often unwieldy for producing and maintaining efficient application code.

The Crystal approach to programming massively parallel machines [5] is to begin with a machine-independent, high-level problem specification. We then apply a sequence of transformations, either suggested by the programmer or incorporated into the compiler, to the source specification. Some of these transformations are specific to particular machine architectures so that efficient target code with explicit communication or synchronization can be generated.

One of the central issues in discovering parallelism automatically is to generate correct parallel control structures that can take advantage of the large number of processors. This dependence analysis and testing problem has been well studied in parallelizing compilers [1, 2, 3, 4, 12, 13].

All of the existing dependence decision algorithms, however, are *conservative* in considering data dependence between statements occurring in conditional branches. By conservative we mean that a true dependence does not exist due to the the presence of the conditional yet the dependence test may come out positive. The reasons for not considering exact dependence test for conditional statements in the past may be that (1) the decision algorithm may at the least involve integer programming, which was considered expensive, (2) some of the early vectorizing compilers do not even parallelize conditional statements and hence there is no point getting a more accurate test, and (3) the payoff in speedup as a result of more accurate dependence test may not be significant for small scale (tens of processors) shared-memory multi-computers.

In attempting to parallelize certain algorithms for execution on the Connection Machine which consists of tens of thousands of processors, we realize that the potential speedup can be of orders of magnitude due to a more accurate test for conditional statements. We are thus motivated to devise a decision algorithm which can gain large scale parallelism in practice and yet be fast enough to be used in a parallelizing compiler.

In the following, we first review the concepts and terminology appearing in the dependence test literature. We then formulate precisely the problem of testing data dependence between conditional statements in Section 3. A new dependence decision algorithm called *subdomain dependence test* is described in Section 4. In Section 5, we discuss how to do subdomain dependence test when some loop bounds are unknown. Finally, we describe the application of subdomain dependence test to the problem of "dependence cycle breaking" [2] and the use of subdomain dependence test in functional programs.

2 Previous Work

Throughout this paper, programming examples are written in a Fortran-like notation. In order to make this paper self-contained, we first review the terminology and concepts of dependence analysis.

Let S_1 and S_2 be two statements of a program. A *flow dependence* exists from S_1 to S_2 if S_1 writes data that can subsequently be read by S_2. An *anti-dependence* exists from S_2 to S_1 if S_1 reads data that S_2 can subsequently write. An *output dependence* exists from S_1 to S_2 if S_1 writes data that S_2 can subsequently write. We use the notation $(S_1 \Rightarrow S_2)$ to denote a dependence from S_1 to S_2.

The dependences between n statements of a program can be represented as a graph, called the *dependence graph*, consisting of n nodes, each node labeled by one statement. For each

dependence from statement S_1 to statement S_2, there is an arc from node S_1 to node S_2 in the graph.

In this paper, we discuss flow dependence only. Anti-dependence and output dependence can be treated similarly.

We define an *index domain* (also called an *iteration space* in [12]) of a d-nested do loop

$$\begin{array}{l} \mathsf{DO}\ (i_1 = l_1, u_1)\,\{ \\ \quad \mathsf{DO}\ (i_2 = l_2, u_2)\,\{ \\ \quad\quad \ldots \} \qquad\quad \} \end{array}$$

to be the Cartesian product of d interval domains of integers $l_k \leq i \leq u_k$ for $1 \leq k \leq d$.

Since indices of nested do loops are used throughout the paper, we use I and J as the tuples of indices $(i_1, i_2, \ldots)$ and $(j_1, j_2, \ldots)$ respectively. For a nested do loop with index domain D, we use the notation

$$\begin{array}{l} \mathsf{DO}\ (I{:}D)\,\{ \\ \quad \ldots \} \end{array}$$

as a shorthand for

$$\begin{array}{l} \mathsf{DO}\ (i_1 = \ldots)\,\{ \\ \quad \mathsf{DO}\ (i_2 = \ldots)\,\{ \\ \quad\quad \ldots \} \qquad\quad \} \end{array}$$

We now define relations on elements I and J of a d dimensional index domain. We define "$\prec$" to be the lexicographical ordering: we say $(I \prec J)$ if there exists k, $1 \leq k \leq d$, such that $(i_l = j_l)$ for all l, $l < k$, and $(i_k < j_k)$. We define "$<$" to be an element-wise ordering of "$<$" and say $(I < J)$ if $(i_k < j_k)$ for all k, $1 \leq k \leq d$. Similarly, "$=$" is defined to be $(i_k = j_k)$ for all k, $1 \leq k \leq d$. We say $(I \preceq J)$ if $(I \prec J)$ or $(I = J)$, and $(I \leq J)$ if $(I < J)$ or $(I = J)$.

In this paper, we use the following perfectly nested loop as a generic example, where D is a d dimensional index domain and $\tau[a]$ is an expression containing a:

Program G (Generic)

$$\begin{array}{l} \mathsf{DO}\ (I{:}D)\,\{ \\ \quad \ldots \\ \quad S_1: \quad \mathsf{IF}(P_1(I)) \quad A(X(I)) = \ldots \\ \quad \ldots \\ \quad S_2: \quad \mathsf{IF}(P_2(I)) \quad B(Z(I)) = \tau[A(Y(I))] \\ \quad \ldots \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \} \end{array}$$

For statement S_2 to compute the value $B(Z(J))$ at iteration J, the value $A(Y(J))$ is needed. If $A(Y(J))$ is computed from statement S_1 at iteration I, i.e. $(A(Y(J)) = A(X(I)))$, then we say S_2 at iteration J is flow dependent on S_1 at iteration I, denoted by $(S_1@I \Rightarrow S_2@J)$.

We are interested in the relationship of two iterations I and J such that $(S_1@I \Rightarrow S_2@J)$. Let **sig** be a function mapping from the set of integers $\mathcal{Z}$ to the set of ordering relations "<", "=", and ">":

$$\mathbf{sig}(z) = \left\{ \begin{array}{l} z < 0 \rightarrow \text{“<”} \\ z = 0 \rightarrow \text{“=”} \\ z > 0 \rightarrow \text{“>”} \end{array} \right\}$$

For any two iterations I and J such that the dependence relation $(S_1@I \Rightarrow S_2@J)$ holds, we call the tuple $(e_1, \ldots, e_d)$, where $e_k = \mathbf{sig}(i_k - j_k)$, a *direction vector* associated with the dependence. We use "$*$" as a shorthand for ("<" or "=" or ">").

Traditional dependence tests for Program G is formulated as: There is a flow dependence from S_1 to S_2 if and only if there exist integer indices I and J satisfying Equations (1),(2), (3) and (4). Note that the presence of conditional expressions is ignored:

$$\begin{aligned} I &\in D, && (1)\\ J &\in D, && (2)\\ X(I) &= Y(J), \quad \text{and} && (3)\\ (I &\preceq J \quad \text{(for flow dependence)}, \quad \text{or} && (4)\\ I &\succ J \quad \text{(for anti-dependence))}. && (5) \end{aligned}$$

Equation (4) implies that $(d+1)$ direction vectors must be considered. For example, if the dimension d is 3, then the direction vectors are $(<,*,*)$, $(=,<,*)$, $(=,=,<)$, and $(=,=,=)$. Similarly, Equation (5) for testing anti-dependence implies d direction vectors.

Let $\mathcal{Z}^d$ be the d-dimensional Cartesian product of $\mathcal{Z}$. Let l_{ij} and u_{ij} be rational constants where $2 \leq i \leq d$ and $1 \leq j \leq d-1$. Let P and Q be two d-tuples of constant rational numbers, and L and U be two constant matrices:

$$L = \begin{pmatrix} 0 & 0 & 0 & \ldots & 0 \\ l_{21} & 0 & 0 & \ldots & 0 \\ l_{31} & l_{32} & 0 & \ldots & 0 \\ & & & \ldots & \\ l_{d1} & l_{d2} & l_{d3} & \ldots & 0 \end{pmatrix}, U = \begin{pmatrix} 0 & 0 & 0 & \ldots & 0 \\ u_{21} & 0 & 0 & \ldots & 0 \\ u_{31} & u_{32} & 0 & \ldots & 0 \\ & & & \ldots & \\ u_{d1} & u_{d2} & u_{d3} & \ldots & 0 \end{pmatrix}.$$

A subspace of $\mathcal{Z}^d$ that can be expressed as $\{I \mid (P \leq I \leq Q)\}$ is called a *rectangle* [3]. A subspace of $\mathcal{Z}^d$ that can be expressed as $\{I \mid ((LI^T)^T + P \leq I \leq (UI^T)^T + Q)\}$ is called a *trapezoid* [3].

Traditional dependence tests [3] restrict that an index domain of a do loop be either a rectangle or a trapezoid, or a union of such domains. For the purpose of a dependence test, each member domain of the union can be considered separately. Hence we can further restrict, without loss of generality, that an index domain of a do loop be a rectangle or a trapezoid. Conventional dependence tests also restrict all subscript functions, e.g. X, Y and Z in Program G, to be affine functions of the indices.

The methods used [3, 12, 13] to determine if there exist I and J that satisfy Equations (1),(2),(3) and (4), are summarized as follows:

1. For a single level (one dimensional) loop, exact dependence is obtained by Wolfe's method [12], which requires linear time with respect to the dimensionality of the arrays.

2. For one dimensional arrays occurring in a loop with nested levels, the Banerjee-Wolfe test[12] determines if there exist *rational* indices. Wolfe's method [13] determines further if *integer* indices exist. Both tests only apply to index domains that are rectangles or trapezoids. Furthermore, Wolfe's integer test requires exponential time with respect to the dimensionality of the index domain.

3. A multi-dimensional array occurring in a loop with nested levels is linearized as a one dimensional array so as to use the methods described in 2. Shostack's loop residues algorithm [4, 11] and linear programming algorithms [8] can be used to determine if *rational* indices exist. Exact test for the existence of *integer* indices needs to be obtained by integer programming.

3 Subdomain Dependence Test

We will now formulate the same problem with conditionals taken into consideration. Since expressions of domain indices appearing as guards in conditional statements can be thought of as conditions that divide an index domain into subdomains, we call such a dependence test a *subdomain dependence test.*

3.1 Example

We now use a simple example to show the importance of subdomain dependence test. Consider the following program:

Program 1

$$
\begin{array}{l}
\mathsf{DO}\ \ (i = 2, n)\,\{ \\
\quad \mathsf{DO}\ \ (j = 2, n)\,\{ \\
\qquad S_1: \quad A(i,j) = B(i, j-1) + i \\
\qquad S_2: \quad B(i,j) = A(i,j) + j \qquad \}\,\}
\end{array}
$$

A flow dependence from S_1 to S_2 exists because of array A, and another from S_2 to S_1 due to array B. The cyclic dependence as shown in the dependence graph Figure 1(a) prevents the inner loop from being parallelized. Now consider the same program except S_1 and S_2 are modified as conditional statements:

Program 2

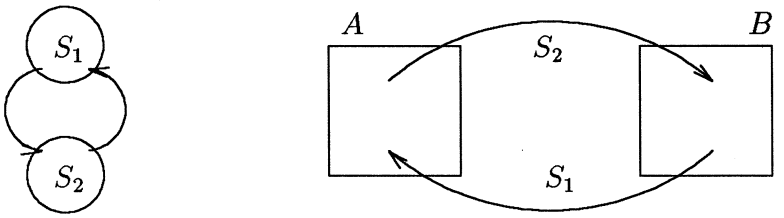
$$
\begin{array}{l}
\mathsf{DO}\ \ (i = 2, n)\,\{ \\
\quad \mathsf{DO}\ \ (j = 2, n)\,\{ \\
\qquad S_1: \quad \mathsf{IF}(i > j) \quad A(i,j) = B(i, j-1) + i \\
\qquad S_2: \quad \mathsf{IF}(i \leq j) \quad B(i,j) = A(i,j) + j \qquad \}\,\}
\end{array}
$$

All conventional dependence test would test Program 2 the same way as Program 1 where a cyclic dependence between S_1 and S_2 exists. But in reality, the data read by S_1 is not written by S_2, nor is the data read by S_2 written by S_1. Therefore, no flow dependence between S_1 and S_2 exists and the dependence graph is shown in Figure 1(b). Both levels of the loop in Program 2 can thus be fully parallelized. Note that even the *cycle breaking* techniques discussed by Banerjee [2, 13] cannot break this cycle.

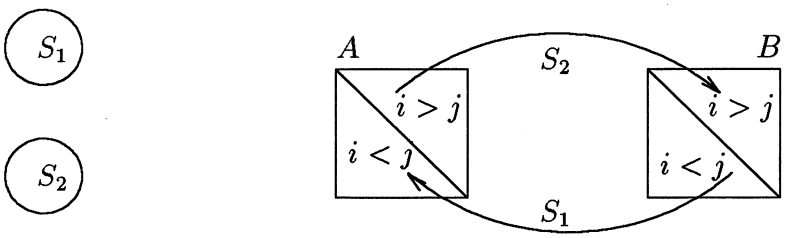
3.2 Notation and Definitions

Let h_k and c be rational constants where $1 \leq k \leq d$. Let H be a d-tuple $(h_1, h_2, \ldots, h_d)$. A subspace of $\mathcal{Z}^d$ that can be expressed as $\{I \mid HI^T = c\}$ is called a *hyperplane*. A subspace of $\mathcal{Z}^d$ that can be expressed as $\{I \mid HI^T \leq c\}$, is called a *half space*. The intersection of a finite number of hyperplanes and half spaces is called a *polyhedron*. A bounded polyhedron is called a *polytope* [10].

A conditional expression is said to be in *disjunctive normal form* if it is the disjunction (or) of conjunctions (and) of predicates. It is well-known that any conditional expression can be transformed into disjunctive normal form.



(a) S_1 and S_2 of Program 1 have cyclic flow dependence



(b) S_1 and S_2 of Program 2 do not have cyclic flow dependence when the index domain is partitioned

Figure 1: Effect of subdomain dependence test

Consider a statement S with a conditional expression P in a loop over index domain D:

$$\begin{array}{l} \mathsf{DO}\ (I{:}D)\,\{ \\ \quad \ldots \\ \quad S:\ \ \mathsf{IF}(P)\ \ \ldots \\ \quad \ldots \qquad\qquad\quad \} \end{array}$$

We define the index domain of statement S to be D under the restriction of P, denoted by $(D{\downarrow}P)$.

3.3 Problem Formulation

Now consider Program G. A subdomain dependence test is formulated as: A flow dependence from S_1 to S_2 exists if and only if there exist integer index tuples I and J satisfying

$$I \in (D{\downarrow}P_1), \tag{6}$$
$$J \in (D{\downarrow}P_2), \tag{7}$$
$$X(I) = Y(J), \quad \text{and} \tag{8}$$

$$(I \preceq J \quad \text{(for flow dependence)}, \quad \text{or} \tag{9}$$
$$I \succ J \quad \text{(for anti-dependence))}. \tag{10}$$

As usual, we restrict the form of a conditional expression to be an affine expression of the indices. Under this restriction, a predicate in the disjunctive normal form of a conditional expression is either an equality ($HI^T = c$) or an inequality ($HI^T \leq c$). Therefore, a conjunction of predicates specifies a polyhedron and a disjunction specifies a union of polyhedrons. Since we can test multiple polyhedrons one by one, it suffice to consider predicates that specify a polyhedron. To summarize, the index domain of a statement in a conditional branch generated by predicate P is $(D{\downarrow}P)$ which is the intersection of a rectangle or a trapezoid with a polyhedron, which is, in general, a polytope.

Note that a rectangle or a trapezoid is a polytope, but not vice versa. This is another way of seeing that, in the presence of conditional expressions, the Banerjee-Wolfe test [12] and Wolfe's integer test [13] are too conservative, because the tests are applied to rectangle or trapezoid that encloses the polytope in question.

Since Equations (6),(7) and (8) specify a polytope while Equations (9) requires the consideration of $(d+1)$ direction vectors, the union of $(d+1)$ polytopes must be considered. A flow dependence from statement S_1 to statement S_2 exists in Program G if and only if the $(d+1)$ polytopes contain integer points. Similarly, in the anti-dependence case, Equations (6),(7),(8) and (10) specify the union of d polytopes.

It appears that to obtain both flow and anti-dependence information, $(2d+1)$ polytopes must be tested for each pair of statements S_1 and S_2 in Program G. This complexity can be reduced by applying the techniques of *hierarchical dependence test* [4] so as to reduce the number of direction vectors need to be examined and thus the number polytopes to be tested.

We now want to show that subdomain dependence test is as hard as deciding whether a *general* polytope contains integer points. By general we mean that the polytope can not be characterized as a member of a special class such as rectangles or trapezoids. Consider Program G. Suppose we want to know if there is any dependence between statements S_1 and S_2 due to references to array A. That is, we want to test whether the polytope P specified by Equations (6),(7), and (8) contains integer points. We use the notation $F(E)$ to mean the set $\{F(I) \mid I \in E\}$ where F is an affine function and E is an index domain. From the equality $(X(I) = Y(J))$, it is easy to see that polytope P can also be specified as

$$X(D{\downarrow}P_1) \cap Y(D{\downarrow}P_2),$$

which can not be characterized as a member of a special class.

4 Algorithms for Subdomain Dependence Test

The known technique for testing whether a polytope contains integer points is by integer programming. Since the dimensionality of the polytope is small in practice, we developed the following algorithm based on the existing integer programming algorithms developed for various special cases:

1. *Base case:* Two dimensional integer programming problems can be solved in $O(n \log n)$ time by Feit's algorithm where n is the number of constraints [6].

2. *Reduction of dimensionality:* If the dimension of a polytope P is greater than two, then we can reduce the dimension by either of the following two methods. If the dimension can be reduced to one or two, then we use Feit's algorithms to test.

   (a) *Linearly Independent Equalities:* Assume P is specified by n linearly independent equalities. We can use the methods given in [3] to solve the diophantine equations of these n equalities to decrease the dimension of P by n.

   (b) *Subspace:* If a variable does not appear in a set of equalities and inequalities specifying P, then this set specifies a polytope which is one dimensional less than P. If this new polytope is empty, then P is empty.

3. *General case:* If the dimensionality can not be reduced to 2 or less, we use Lenstra's algorithm [7] and Scarf's *base reduction* algorithm [9]. The time complexity of these algorithms is polynomial in the number of equations that specify a polytope and exponential in the dimensionality of a polytope.

We have coded the above algorithm and found that, experimently, testing a 3-dimensional polytope takes 1 second, while testing a 12-dimensional polytope takes 20 seconds using the base reduction algorithm on a Sparc station. Since most programs uses arrays with dimensions no more than 3, the timing for subdomain dependence tests is quite acceptable.

5 Unknown Loop Bounds

At compile time, some loop bounds may be specified by unknown constants. In this case, we will treat these unknown constants as extra variables. Therefore, a d dimensional polytope P specified by u unknown constants can be considered as a $(d+u)$ dimensional index domain P' without unknown constants. Since there is no bounds for these extra variables, P' can be a polyhedron. If the dimensionality of P' can be reduced to 2, then Feit's algorithm [6] can decide whether P' contains integer points even when P' is a polyhedron. The base reduction algorithm [9] can not test if a polyhedron is free of integer point but it can determine whether P' is a polytope or a polyhedron. If the dimensionality of P' can not be reduced to 2 or less and P' is a polyhedron, then we assume that P' contains integer points.

6 Application of Subdomain Dependence Test

6.1 Cycle Breaking

Banerjee's *cycle breaking* technique [2, 12, 13] is cappable of breaking a cycle consisting of a flow and an anti-dependence between two statements in a single level loop of the following form, where B is a one dimensional array and l, u and c_k for $1 \leq k \leq 4$ are integer constants:

Program 3

$$\begin{array}{ll} \mathsf{DO}\ (i = l, u)\,\{ & \\ \quad S_1: & A(\ldots) = B(c_1 i + c_2) \\ \quad S_2: & B(c_3 i + c_4) = \ldots \quad \} \end{array}$$

This kind of cycle can always be removed because the flow and anti-dependences are over disjoint sub-index domains of the two statements. The boundaries of the sub-index domains can be computed only when c_1 and c_3 are not both zero [2], i.e. the inverse of either $(c_1 i + c_2)$ or $(c_3 i + c_4)$ is well defined. In the following program [13]:

Program 4

$$\begin{array}{ll} \mathsf{DO}\ (i = 1, 101)\,\{ & \\ \quad S_1: & A(i) = B(101 - i) + i \\ \quad S_2: & B(i) = E(i) \quad \} \end{array}$$

The flow dependence $(S_2@j \Rightarrow S_1@i)$ is over $(i \in [51, 100])$ and $(j \in [1, 50])$; and the anti-dependence $(S_1@i \Rightarrow S_2@j)$ is over $(i \in [1, 50])$ and $(j \in [51, 100])$. By splitting index domain $[1, 101]$ into disjoint domains $[1, 50]$ and $[51, 101]$, the cycle is removed and the loop can be fully parallelized as shown below:

Program 5

$$\begin{array}{ll} \mathsf{DOALL}\ (i = 1, 50)\,\{ & \\ \quad S_1: & A(i) = B(101 - i) + i \\ \quad S_2: & B(i) = E(i) \quad \} \\ \mathsf{DOALL}\ (i = 51, 101)\,\{ & \\ \quad S_1: & A(i) = B(101 - i) + i \\ \quad S_2: & B(i) = E(i) \quad \} \end{array}$$

But subdomain dependence test break cycles for more general cases. It can break cycles for any number of dependences between more than multiple statements in a multi-level loop with reference to multi-dimensional arrays. We use the following example to show the power of subdomain dependence test in breaking cycles:

Program 6

$$
\begin{array}{lll}
\mathsf{DO}\ (I{:}D)\,\{ & & \\
\ldots & & \\
S_1: & \mathsf{IF}(P_1(I)) & A(X(I)) = C(Y(I)) \\
S_2: & \mathsf{IF}(P_2(I)) & B(\ldots) = A(Z(I)) \\
S_3: & \mathsf{IF}(P_3(I)) & C(W(I)) = B(\ldots) \\
\ldots & & \}
\end{array}
$$

Assume there are cyclic flow dependences ($S_1 \Rightarrow S_2 \Rightarrow S_3 \Rightarrow S_1$). We know that ($S_1@I \Rightarrow S_2@J$) for I and J in the following polytope:

$$
\begin{aligned}
I &\in (D{\downarrow}P_1), && (11)\\
J &\in (D{\downarrow}P_2), && (12)\\
X(I) &= Z(J), \quad \text{and} && (13)\\
I &\preceq J. && (14)
\end{aligned}
$$

Similarly, ($S_3@K \Rightarrow S_1@I$) for I and K in the following polytope:

$$
\begin{aligned}
I &\in (D{\downarrow}P_1), && (15)\\
K &\in (D{\downarrow}P_3), && (16)\\
Y(I) &= W(K), \quad \text{and} && (17)\\
I &\succeq K. && (18)
\end{aligned}
$$

Let D_1 be the domain containing all I satisfying Equations (11),(12),(13) and (14), and let D_2 be the domain containing all I satisfying Equations (15),(16),(17) and (18). By subdomain dependence test, we know whether integer tuples I, J and K exist which satisfy Equations from (11) to (18). If I_1, J_1 and K_1 are such integer tuples, then ($I_1 \in D_1$) and ($I_1 \in D_2$); therefore, D_1 and D_2 are not disjoint. Conversely, if no such integer tuples exist, then D_1 and D_2 are disjoint, and we can decompose S_1 into two sub-statements S_{1a} and S_{1b}:

$$
\begin{array}{lll}
S_{1a}: & \mathsf{IF}(I \in (D{\downarrow}P_1) - D_2) & A(X(I)) = \ldots \\
S_{1b}: & \mathsf{IF}(I \in (D_2)) & A(X(I)) = \ldots
\end{array}
$$

such that the flow dependences become ($S_{1a} \Rightarrow S_2 \Rightarrow S_3 \Rightarrow S_{1b}$), and the cycle is removed.

In order to obtain D_2, the inverse of the subscript function W should be well defined, same as in Banerjee's celebrity technique. Let W^{-1} be the inverse of W. Equation (17) implies that $(K = W^{-1}(Y(I)))$ and it is clear that

$$D_2 = \{I \mid I \in (D{\downarrow}P_1), (W^{-1}(Y(I)) \in (D{\downarrow}P_3)), \text{ and for } (I \succeq W^{-1}(Y(I)))\}.$$

With D_2, statement S_1 can be decomposed and the cycle can be removed.

6.2 Use of Subdomain Dependence Test in Functional Programs

Subdomain dependence test is useful not only for imperative programs but also for functional programs. We use the following notation to express a functional program, where A and B are two function definitions, D and E are index domains, P_1 and P_2 are conditional expressions and Y is an affine function:

$$A(I{:}D) = \left\{ \begin{matrix} P_1 \rightarrow \ldots \\ \ldots \end{matrix} \right\}$$

$$B(I{:}E) = \left\{ \begin{matrix} P_2 \rightarrow \tau[A(Y(I))] \\ \ldots \end{matrix} \right\}$$

$$\ldots$$

If only dependences between function definitions, e.g. A and B, are considered, then no dependence test is required to say that B dependeds on A. However, A and B can be decomposed into sub-definitions according to conditional expressions as

$$\begin{aligned} A_1(I{:}(D{\downarrow}P_1)) &= \ldots \\ B_1(I{:}(E{\downarrow}P_2)) &= \tau[A(Y(I))] \\ &\ldots \end{aligned}$$

More precise dependences can be obtained between these sub-definitions. To know if B_1 depends on A_1, subdomain dependence test can be used to determine if the following polytope contains integer points:

$$(D{\downarrow}P_1) \cap Y(E{\downarrow}P_2).$$

Due to functionality, only one polytope, instead of $(2d + 1)$, needs to be tested to obtain dependence information between each pair of sub-definitions.

7 Conclusion

We describe in this paper a new dependence decision algorithm which gives a more accurate test of dependence between conditional statements. The algorithm is practical and may discover large scale parallelism in cases where previous decision algorithms fail.

In the parallelizing compiler literature, the "loop skewing" technique can be used to transform the loop body so as to generate parallelism in cases where dependence test gives positive result. One remaining question is that can such loop skewing technique be improved so as to generate massive parallelism? This result will be presented in a sequel.

References

[1] J.R. Allen. *Dependence Analysis for Subscript Variables aand Its Application to Program Transformation.* PhD thesis, Rice University, April 1983.

[2] U. Banerjee. *Speedup of Ordinary Programs.* PhD thesis, University of Illinois at Urbana-Champaign, October 1979.

[3] U. Banerjee. *Dependence Analysis for Supercomputing.* Kluwer Academic Publishers, 1988.

[4] Michael Burke and Ron Cytron. Interprocedural dependence analysis and parallelization. In *Proceedings of the SIGPLAN'86 Symposium on Compiler Construction*, pages 162–175. ACM, 1986.

[5] Marina Chen, Young-il Choo, and Jingke Li. Compiling parallel programs by optimizing performance. *The Journal of Supercomputing*, 1(2):171–207, July 1988.

[6] S. D. Feit. A fast algorithm for the two-variable integer programming problem. *Journal of the ACM*, 31(1):99–113, Jan. 1984.

[7] H. W. Lenstra. Integer programming with a fixed number of variables. *Math. of Operations Research*, pages 538–548, 1983.

[8] C. H. Papadimitriou and K. Steiglitz. *Combinatorial Optimization Algorithms and Complexity.* Prentice-Hall, INC., 1982.

[9] Scarf. Integer programming and lattice theory. *Lecture Notes, Department of Computer Science, Yale University*, 1990.

[10] A. Schrijver. *Theory of Linear and Integer Programming. Wiley-Interscience series in Discreate Mathematics.* John Wiley and Sons, 1986.

[11] Robert Shostak. Deciding linear inequalities by computing loop residues. *Journal of the ACM*, 28(4), Oct. 1981.

[12] Michael Wolfe. *Optimizing Supercompilers for Supercomputers.* PhD thesis, University of Illinois at Urbana-Champaign, October 1982.

[13] Michael Wolfe. *Optimizing Supercompilers for Supercomputers.* The MIT Press, 1989.