

**University of Toronto**

# **Dependence Testing for Parallelizing Compilers**

**ECE 1754 Survey Paper**

**Student: Franjo Plavec**

**Toronto, May 2003**

# Abstract

Dependence testing is an important aspect of a parallelizing compiler. Precise data dependence information is necessary to detect parallelism. Main source of the parallelism in programs are loops. Dependence testing discovers data dependences between iterations of nested loops.

In this paper we present the dependence testing problem, terminology and concepts used in dependence testing. We present several simple dependence tests, and two complex algorithms, the Omega test and Rice test. We compare the effectiveness and efficiency of various tests and algorithms and discuss possible improvements.

## 1 Introduction

One of the main requirements of any computer system is high performance. Building a parallel system is one of the ways to achieve performance improvements. Many algorithms have a lot of inherent parallelism. Parallelization exploits the parallelism by executing independent parts of the algorithm on a multiprocessor computer system.

Writing programs for parallel computers is a complex task. Programmer has to identify parallel parts of the algorithm, re-write the algorithm in a parallel manner, and implement synchronization. Debugging parallel programs is extremely difficult, since it is hard to repeat the execution of the program that causes the problem due to side-effects, such as system calls or timer interrupts. One of the ways to deal with these issues is to automate the process of program parallelization.

Parallelizing compiler analyzes the sequential program, and transforms it to the parallel version. It detects the parts of the program that can be parallelized, and produces the parallel program as the output. Output program is usually expressed in the same programming language as the input program. Most existing parallelizing compilers target the Fortran code. Fortran is the most popular programming language for parallelization for two reasons. First, many algorithms in scientific computing are written in Fortran. Scientific applications are of a special interest to the parallelization, since they are complex enough to profit from parallelization. Second, Fortran has many properties that make the analyses of the program and program transformations easier. These properties include lack of pointers, static memory model, and others [1].

A compiler transformation may not change the semantics of the original program. Therefore, the parallelizing compiler can only parallelize independent parts of the program. Process of detecting dependences among the parts of the program is called dependence testing. While compilers handle dependences caused by scalar variables easily, array references introduce another dimension to the problem. Array reference typically consists of the array name and subscript expression. Array reference  $A(k)$  accesses the  $k$ -th element in the array  $A$ .  $A$  is usually referred to as an array name, while  $k$  is called subscript expression. Two references in the program, referring to distinct arrays, such as  $A(3)$  and  $B(3)$  do not incur any dependence (in the absence of aliasing). Depending on the subscript expression, two references referring to the same array, may, or may not access the same memory location. Accesses to  $A(1)$  and  $A(5)$ , for instance, are independent.

Ninety percent of the execution time of the program is usually spent in the ten percent of the code [6], namely loops. Loops are the main source of the parallelism in a program. As the size of the problem increases, the amount of work performed by most loops in the program usually scales appropriately. Therefore, loops are of the particular interest to the parallelization.

If there is no dependence among the iterations of the loop, all iterations can be executed in parallel. Determining if the dependence among the iterations of the loop exists, requires analysis of subscript expressions in terms of loop control variables. Depending on the subscript expressions and loop bounds, array references may, or may not introduce dependence. Dependence testing is done in pairs. Each array reference in a loop is paired with each other reference to the same array, and tested for dependence. Dependence exists if any two iterations of the loop access the same array with the same subscript (i. e. the same memory location). If any of the tests yields dependence, loop can not be parallelized. However, depending on the type of the dependence, applying some transformations may enable partial or full parallelization of the loop [1].

If the subscript expressions are linear functions of loop index variables, dependence testing is equivalent to the problem of finding integer solutions to the system of linear equalities and inequalities, also known as Integer Linear Programming. This problem is known to be NP-complete [5], making it inefficient to perform the general test for all references. However, there exist many simple and efficient tests that can handle simple, special cases. Therefore, most dependence testing algorithms apply divide and conquer

strategy. First, special case tests are applied to simple cases. If any references are left unresolved, general test is applied. Measurements show that simple cases prevail, thus making this strategy efficient [5, 9].

Dependence testing algorithms are classified into two groups. Exact algorithms will report dependence if and only if the dependence really exists. In-exact tests try to prove the independence. If the independence can be neither proved nor disproved, test will report dependence, since the semantics of the program could change if the loop is parallelized in the presence of the dependence.

In this paper we present various techniques and algorithms for dependence testing. We also analyze and evaluate the effectiveness and efficiency of various dependence testing algorithms.

The rest of the paper is organized as follows. In section 2 we give precise formulation of the problem. Simple tests for dependence testing are described in the section 3. Sections 4 and 5 present two exact tests for data dependence testing, Omega, and Rice test. In section 6 we show how memoization is used to reduce the number of times the data dependence test needs to be applied. In section 7 we discuss the effectiveness and efficiency of various tests. We conclude in section 8.

## **2 Problem Formulation**

This section presents the problem definition and terminology used in the rest of the paper. We start by presenting types of dependences that can exist in the program.

### *2.1 Types of Dependences*

Two statements in the program are said to be dependent if they access (read or write) the same data. There are four types of dependences. If two statements in the program  $S_1$  and  $S_2$  access the same location, and  $S_1$  precedes  $S_2$  in the original order of execution, the dependence is said to be:

- flow dependence, if  $S_1$  writes the value that  $S_2$  reads. Flow dependence is also called a true dependence.
- anti dependence, if  $S_1$  reads the value that  $S_2$  writes. It is important to mention that by the value we mean register, or the memory location, not the actual value that is written. This is valid for all types of the dependences, except for the true

dependence, which could be defined by actual value.

- output dependence, if both  $S_1$  and  $S_2$  write the same value.
- input dependence, if both  $S_1$  and  $S_2$  read the same value.

Flow dependence is also called a true dependence since it is the only type of the dependence that actually incurs the transfer of the value from the statement  $S_1$  to  $S_2$ . Anti and output dependence can be eliminated by register renaming [1]. Input dependence is not real dependence in most scientific applications, since it does not impose execution order among statements. However, input dependence can be important if reading the value has some side-effects (e. g. reading serial I/O port), or when the data reuse information is needed [7]. For the purpose of this paper, we will ignore input dependences.

Following example illustrates the types of dependences:

(S <sub>1</sub> )	$A = B + C;$
(S <sub>2</sub> )	$C = D * E;$
(S <sub>3</sub> )	$F = A / B;$
(S <sub>4</sub> )	$F = X [J] + G;$
(S <sub>5</sub> )	$X [K] = H - I;$

There is an anti dependence between statements  $S_1$  and  $S_2$  because of the use of the variable  $C$ . Statement  $S_2$  can not be executed before statement  $S_1$ , because it would overwrite the value that  $S_1$  needs. However, if we change the statement  $S_2$  to write to some other variable  $W$ , and replace all later uses of variable  $C$  in the program with  $W$ , the dependence does not exist any more, and statements can be executed in any order. This process is called register renaming [1].

There is a true dependence between statements  $S_1$  and  $S_3$  because of the use of the variable  $A$ . This is the true dependence, since  $S_3$  reads the actual value that  $S_1$  produces, thus imposing the order of execution of these two instructions.

There is also an input dependence between statements  $S_1$  and  $S_3$  because of the use of the variable  $B$ . Input dependence does not impose any execution order, since the reading of the values can be done in any order in the absence of side-effects.

The output dependence exists between statements  $S_3$  and  $S_4$  because of the write to the variable  $F$ . This dependence could be eliminated by register renaming, or by simply removing

statement  $S_3$  from the program, since its result is never used.

There is a possible anti dependence between statements  $S_4$  and  $S_5$  depending on the values of  $J$  and  $K$ . If  $J$  and  $K$  are equal, both statements access the same memory location, hence, there is a dependence. If  $J$  and  $K$  are never equal, the dependence does not exist.

All dependences in the program can be represented by a dependence graph  $G = (V, E)$ . Set of vertices  $V$  represents statements in the program, while set of directed edges  $E$  represents dependences. Edges of the graph may be labeled, to represent the type of the dependence. If two program statements,  $S_1$  and  $S_2$ , are dependent, and  $S_1$  precedes  $S_2$  in original program execution, edge in the dependence graph is directed from  $S_1$  (source of the dependence) to  $S_2$  (sink of the dependence).

In the rest of the paper we will deal with the ways to determine if there is a dependence due to array references inside the loops. Dependence analysis due to scalar variables is well known problem, whose solution can be found in the literature [10].

## *2.2 Dependence Testing Inside the Loop Nests*

Dependence testing is the method used to determine whether dependences exist between two subscripted references to the same array in a loop nest [5]. We consider only Fortran DO loops, because of their regular structure. All loops are assumed to be normalized, having the step size of 1. Only perfect loop nests are considered. A perfect loop nest does not contain any code, except for the loop headers, in any loop body, except the body of the innermost loop. If the program contains non-perfectly nested loops, loop distribution can be used to convert it to the perfectly nested loop [1]. The following code shows typical perfectly nested loop with two array references.

```

DO i1 = L1, U1
  DO i2 = L2, U2
    ...
    DO in = Ln, Un
      (S1)      A (f1(i1, ..., in), ..., fm(i1, ..., in)) = ...
      (S2)      ... = A (g1(i1, ..., in), ..., gm(i1, ..., in));
    ENDDO
  ...
  ENDDO
ENDDO

```

Functions  $f_1$  to  $f_m$  and  $g_1$  to  $g_m$  are called subscript expressions. Most methods for data dependence testing only consider the case when subscript expressions are linear functions of loop indices  $i_1$  to  $i_n$ . If subscript expressions are not linear, these methods automatically assume dependence [3, 5, 9, 12]. Position at which subscript expression occurs (i. e. 1 to  $m$ ) is called subscript position. Two subscript expressions at the same subscript position are usually referred to as a subscript pair. Number of subscript positions  $m$ , is determined by the number of dimensions of the array  $A$ . An instance of loop indices  $i_1, \dots, i_n$  can be expressed in a form of an iteration vector  $\mathbf{I} = (i_1, \dots, i_n)$ . Number of dimensions of the iteration vector is equal to the number of nested loops  $n$ .

There is a dependence between two array references  $S_1$  and  $S_2$  if there exist two iteration vectors  $\boldsymbol{\alpha}$  and  $\boldsymbol{\beta}$  such that:

$$\mathbf{L} \leq \boldsymbol{\alpha}, \boldsymbol{\beta} \leq \mathbf{U}$$

$$f_k(\boldsymbol{\alpha}) = g_k(\boldsymbol{\beta}), \forall k, 1 \leq k \leq m$$

where  $\mathbf{L} = (L_1, \dots, L_n)$ , and  $\mathbf{U} = (U_1, \dots, U_n)$  are vectors of lower and upper loop bounds. Otherwise, two references are independent [5, 13]. If all the references in the loop nest are independent, loop iterations can be executed in parallel, since the instances of the statements inside the loop nest (with respect to loop indices) may be executed in any order.

The set of constraints  $\mathbf{L} \leq \boldsymbol{\alpha}, \boldsymbol{\beta} \leq \mathbf{U}$  is usually referred to as a set of inequality constraints, while the set of constraints  $f_k(\boldsymbol{\alpha}) = g_k(\boldsymbol{\beta})$  is called set of equality constraints. The term set of

constraints refers to both equality and inequality constraints.

If the dependence exists between two array references in the same iteration of the enclosing loop, the dependence is said to be loop-independent. If the dependence exists between two array references in different iterations, the dependence is said to be loop-carried [1].

### 2.3 Distance and Direction Vectors

If there is a dependence between two references to the array, distance and direction vectors can be used to characterize the dependence. If the dependence exists for two iteration vectors  $\alpha = (\alpha_1, \dots, \alpha_n)$  and  $\beta = (\beta_1, \dots, \beta_n)$ , the distance vector  $\mathbf{D} = (D_1, \dots, D_n)$  is defined as  $\beta - \alpha$ . The direction vector  $\mathbf{d} = (d_1, \dots, d_n)$  is defined as [5]:

$$d_i = \begin{cases} < & \text{if } \alpha_i < \beta_i \\ = & \text{if } \alpha_i = \beta_i \\ > & \text{if } \alpha_i > \beta_i \end{cases}$$

For example, consider the following loop nest

```
DO i1 = 1, 2
  DO i2 = 1, 2
    A (i1 + 1, i2) = ...
    ... = A (i1, i2) + D;
  ENDDO
ENDDO
```

Dependence testing yields the dependence for the following values of iteration vectors:  $\alpha = (1, 1)$ ;  $\beta = (2, 1)$ ; and  $\alpha = (1, 2)$ ;  $\beta = (2, 2)$ . These iteration vectors yield the same distance and direction vectors in both cases; distance vector  $\mathbf{D} = (1, 0)$ , and direction vector  $\mathbf{d} = (<, =)$ . In some cases, multiple distance and direction vectors may be necessary to characterize dependences:

```

DO i = 1, 3
    A (i-1) = ...
    ... = A (0);
ENDDO

```

Dependence testing yields the dependence for the following values of iteration vectors:  $\alpha = (1)$ ,  $\beta = (1)$ ;  $\alpha = (1)$ ,  $\beta = (2)$  and  $\alpha = (1)$ ,  $\beta = (3)$ . In this case three distinct distance vectors are needed to characterize the dependence:  $\mathbf{D}_1 = (0)$ ,  $\mathbf{D}_2 = (1)$ , and  $\mathbf{D}_3 = (2)$ . However, only two direction vectors suffice:  $\mathbf{d}_1 = (=)$ , and  $\mathbf{d}_2 = (<)$ . Although distance vectors typically provide more information about the dependence, it is often more practical to describe dependences in the terms of direction vectors. If the loop in the example above had a hundred iterations, a hundred distance vectors would be needed to characterize the dependence, whereas only two direction vectors suffice.

Direction vectors are useful for calculating the level of loop-carried dependences. Dependence is carried by the outermost loop for which the direction in the direction vector is not  $=$ . Carried dependences are important because they determine which loops can not be executed in parallel without synchronization. Direction vectors are also useful in determining whether loop interchange is legal and profitable [5]. Distance vectors specify actual distance in loop iterations between two accesses to the same memory location. Distance vectors are used to guide optimizations to exploit parallelism or the memory hierarchy [5].

Conventionally, distance and direction vectors have to be lexicographically positive [1, 3]. This means that the first non-zero element in a distance vector, starting from the outermost loop, has to be positive. Similarly, first element of the direction vector not equal to  $=$  has to be  $<$ . Distance vector represents the distance between the source and the sink of the dependence. Since the source of the dependence always precedes in execution the sink of the dependence, distance vector needs to be lexicographically positive. The definitions for the distance and direction vectors given earlier can still be used. If the lexicographically negative distance or direction vector results, the direction of the vector needs to be reversed. Lexicographically negative direction/distance vector means that the dependence is anti dependence, as demonstrated in the following example.

```

DO i1 = 1, 10
    DO i2 = 1, 10
(S1)        A (i1, i2 + 1) = ...
(S2)        ... = A (i1 + 1, i2);
    ENDDO
ENDDO

```

Dependence test results in distance vector of (-1, 1). Reversed distance vector is (1, -1), which means that if a memory location is read in some iteration (x, y), the same memory location is written in the iteration (x + 1, y - 1). Statement S<sub>2</sub> reads A(2, 3) in iteration (1, 3), while statement S<sub>1</sub> writes A(2, 3) in iteration (2, 2).

## 2.4 Exact Tests

Dependence testing is performed in two phases. In the first phase, dependence testing algorithm tries to prove independence. If the independence is proved, the test ends. Otherwise, the dependence test tries to characterize existing dependences as a minimal complete set of distance and/or direction vectors.

If dependence testing can not either prove or disprove the dependence, it has to conservatively assume that the dependence exists. Otherwise, the validity of any optimization based on dependence information could not be guaranteed. Test that reports the dependence if and only if the dependence exists, is called exact test [5]. All other tests are called in-exact.

None of the tests described in this paper is exact, since all of them assume dependence in case of non-linear subscript expression. However, we will use the relaxed definition of the exact test, which assumes that the test is exact, if it is exact for linear subscript expressions only.

## 2.5 Integer Linear Programming (ILP)

Dependence testing problem can be expressed as a set of linear equalities and inequalities. Equalities are the result of the array subscript expressions. Inequalities result from the loop bounds. When array subscripts are linear expressions of the loop index variables, dependence testing is equivalent to the problem of finding integer solutions to the systems of linear Diophantine equations. This problem is known to be NP-complete [5]. The most efficient integer programming algorithms known, either depend on the value of the loop

bounds, or are order  $O(n^{O(n)})$ , where  $n$  is the number of loop variables [9]. Such algorithms are aimed at large systems with at least few hundred constraints and variables. They are not suitable for data dependence analysis, where a large number of small systems need to be analyzed [8, 13]. In addition, Linear Integer Programming cannot be used to calculate distance/direction vectors [13].

Due to the problem complexity, it is inefficient to perform the general test for all cases. Algorithms therefore apply divide and conquer strategy instead. There exist many simple and efficient tests that can handle simple, special cases. If the special case is recognized in an existing set of equations, corresponding test is applied. If any of the tests yields independence, the system is independent. Complex test is applied to any references left unresolved,. Experiments show that most of the cases occurring in practice are indeed simple [5]. Therefore, although general test is expensive, it is applied rarely, which amortizes the impact on the total cost.

### 3 Simple Tests

In this section we present number of simple tests applicable for special cases. These tests are simple and inexpensive. Many complex algorithms use these tests to solve simple cases.

#### 3.1 Constant Test

If subscript expressions in array references are loop invariant and have the same value, then there will always be a dependence for all potential direction vectors. If subscript expressions are different constants, there can never be a dependence, regardless of other subscripts [11]. Most algorithms perform the constant test in the phase of building the dependence problem [9, 12]. If the independence is proven, the rest of the algorithm, including the rest of the problem building need not be executed, thus saving computation time.

#### 3.2 Lamport's Test

If the subscript expression pair is expressed in terms of a single loop index variable, and the coefficients of the index variable are the same in both expressions, Lamport's test can be applied. Subscript expressions  $a*\alpha_1 + c_1$ , and  $a*\beta_1 + c_2$  can be equal if and only if  $d = (c_1 - c_2)/a$  is an integer. If  $d$  is inside the loop bounds  $L_1 \leq |d| \leq U_1$ , there is a dependence, and  $d$  is the dependence distance. Dependence direction is given by:

$$\text{direction} = \begin{cases} < & \text{if } d > 0 \\ = & \text{if } d = 0 \\ > & \text{if } d < 0 \end{cases}$$

If Lamport's test can be applied to multiple subscript positions, dependence exists if and only if all tests yield dependence.

### 3.3 GCD Test

GCD (Greatest Common Divisor) test is derived directly from the number theory. If the greatest common divisor of all coefficients in a dependence equation does not divide evenly into the constant term, the dependence equation has no integer solutions [3]. For instance, dependence equation  $2*i_1 + 4*i_2 = 3$  does not have integer solutions since  $\text{GCD}(2, 4) = 2$ , does not divide evenly into 3.

GCD test can only be used to prove independence. In many cases, greatest common divisor of all coefficients will be 1, which divides any constant. Therefore, even if the GCD does evenly divide constant, that still does not mean there is a dependence, since the solution may not lie in the iteration space bounds [2].

Generalized GCD test [2] is an extension of GCD test that considers all subscripts in a multiply dimensioned array simultaneously. The system of equations is written in the matrix form, and factored in a form that is easier to solve. The generalized GCD test can still prove independence only [11]. Generalized GCD test is also called Banerjee's Extended GCD test [9].

### 3.4 Single-Index Exact Test

Banerjee [2] shows how the single Diophantine equation in two variables can be solved exactly. The Diophantine equation  $a\alpha_k + b\beta_k = c$  has a solution if and only if  $d = \text{GCD}(a, b)$  divides  $c$ , which is the application of the GCD test described in the previous section. If the solution exists, the general solution is given by  $(\alpha_k, \beta_k) = (c'\alpha_0 - b't, c'\beta_0 + a't)$ , where  $a' = a/d$ ,  $b' = b/d$ ,  $c' = c/d$ ;  $(\alpha_0, \beta_0)$  any particular solution to the equation  $a\alpha_k + b\beta_k = d$ ; and  $t$  an arbitrary integer. Proof of the correctness can be found in [2].

The following example shows how this can be used in dependence testing. Consider the program [2]:

```
DO i1 = 0, 20
(S1)  A (9i1 + 22) = ...
(S2)  ... = A (6i1 - 17);
ENDDO
```

Corresponding dependence equation is  $9\alpha_1 - 6\beta_1 = -39$ , subject to the inequality constraints  $0 \leq \alpha_1, \beta_1 \leq 20$ . Since  $\text{GCD}(9, -6) = 3$ , we pick a solution (1, 1) to the equation  $9\alpha_1 - 6\beta_1 = 3$ . Therefore, the general solution to the dependence equation is  $(-13 + 2t, -13 + 3t)$ . Substituting these expressions into inequality constraints results in  $0 \leq -13 + 2t \leq 20$ , and  $0 \leq -13 + 3t \leq 20$ . Taking tight bounds on  $t$  yields  $7 \leq t \leq 11$ . Therefore, the dependence exists for the set of iteration pairs  $\{(-13 + 2t, -13 + 3t): 7 \leq t \leq 11\}$ , or  $\{(1, 8), (3, 11), (5, 14), (7, 17), (9, 20)\}$ .

Mydan et al [9] call this test Single-Index exact test, since it produces the solution to the constraint involving a single index exactly.

### 3.5 Banerjee Test

Banerjee test verifies the existence of the solution to a system of linear Diophantine equations, with respect to the given set of inequalities. Assume the equality constraint has a form of  $f(\mathbf{I}) = c$ , with bounds on  $\mathbf{I}$ ,  $\mathbf{L} \leq \mathbf{I} \leq \mathbf{U}$ . If function  $f$  is bounded on the region defined by the constraints  $\mathbf{L} \leq \mathbf{I} \leq \mathbf{U}$ , we can find the minimum  $f_{\min}(\mathbf{I})$  and the maximum  $f_{\max}(\mathbf{I})$  of the function  $f$  on the region. The equality  $f(\mathbf{I}) = c$  has a solution only if the GCD of all the coefficients in  $f$  divides  $c$  (GCD test), and  $f_{\min}(\mathbf{I}) \leq c \leq f_{\max}(\mathbf{I})$ . Inequalities of the form  $f_{\min}(\mathbf{I}) \leq c \leq f_{\max}(\mathbf{I})$  have become known as the Banerjee's inequalities [2].

Finding the minimum and the maximum of the equality constraint, when the bounds on indices are simple is straightforward. If  $\forall k, L_k \leq i_k \leq U_k$ , and the function  $f$  has the form

$$\sum_{k=1}^n a_k i_k, \quad \forall k, a_k \geq 0;$$

minimum and maximum of the function  $f$  on a given region are given by

$$f_{\min}(\mathbf{I}) = \sum_{k=1}^n a_k L_k, \quad \text{and} \quad f_{\max}(\mathbf{I}) = \sum_{k=1}^n a_k U_k.$$

The expression with non-negative coefficients is obviously minimal when all its variables are set to their minimal value, and maximal when all variables are set to their maximum value [4]. Banerjee has shown that, in a general case, the

minimum and the maximum of the function can be found on any region that is either a rectangle, a trapezoid, or a region that can be built up from trapezoids by a finite number of operations of union and intersection [2].

Banerjee test is not exact. If the constant term  $c$  is inside the bounds  $f_{\min}(\mathbf{I}) \leq c \leq f_{\max}(\mathbf{I})$ , there exists a real solution, but not necessarily an integer one [4]. For instance, the constraint  $3i_1 + 5i_2 = 7$  on the region  $0 \leq i_1, i_2 \leq 1$  produces the constraint  $0 \leq 7 \leq 8$ , which is satisfied. However, the equation does not have integer solutions on a given region.

The Banerjee test is in some cases exact. Depending on the coefficients and loop bounds, it is possible to determine whether the test will produce exact result. This fact is used in the I-test, a more accurate algorithm based on the iterative application of Banerjee's inequalities and GCD test. I-test is still inexact in general case [4].

Banerjee test can be applied to each equality constraint in the set independently. This results in additional inexactness, since all constraints have to hold simultaneously. The extension of this test, known as  $\lambda$ -test [8], considers all constraints simultaneously. Similarly to the Banerjee test,  $\lambda$ -test can only determine the existence of real-valued solutions [8].

## 4 Omega Test

The Omega test is an integer programming algorithm that can determine whether a dependence exists between two array references, and if so, under what conditions [12]. The Omega test is based on an extension of Fourier-Motzkin variable elimination to integer programming. Fourier-Motzkin variable elimination is a linear programming method. Although Omega test has worst-case exponential time complexity, it has low order polynomial time complexity for many common cases [12].

### 4.1 Normalization

The input to the Omega test is a set of linear equalities and inequalities resulting from array subscript expressions, and loop bounds respectively:

$$\sum_{k=0}^n a_k i_k = 0, i_0 = 1$$

$$\sum_{k=0}^n a_k i_k \geq 0, i_0 = 1$$

All constraints are first normalized. A constraint is said to be normalized if all its coefficients are integers, and the greatest common divisor of the coefficients is one. If the initial constraints involve rational coefficients, they can be scaled to obtain integer coefficients.

Equality constraints can be normalized by finding the  $g = \text{GCD}(a_1, \dots, a_n)$ , and dividing all the coefficients by  $g$ . If the  $g$  does not evenly divide  $a_0$ , there can be no solution to the constraint (GCD test). Therefore, the constraint set has no solution, and test has proven independence.

Inequality constraints can be normalized in a similar fashion. If the  $g = \text{GCD}(a_1, \dots, a_n)$  does not divide  $a_0$  evenly,  $a_0/g$  is replaced by  $\lfloor a_0/g \rfloor$ . Taking the floor of the constant term is legal, since only integer solutions are sought. This tightens the inequality constraint, which can result in non-satisfiable constraint. If the original set of constraints has rational, but not integer solutions, tightening the inequalities may produce a set of constraints without rational solutions, thus making it easier to prove independence.

#### 4.2 Equality Constraint Elimination

Given a system of inequality and equality constraints, equality constraints are eliminated first, producing a new system of inequality constraints that has integer solutions if and only if the original system has integer solutions.

For the purpose of equality constraint elimination, Pugh [12] defined new operation  $\overline{\text{mod}}$  as follows:

$$a \overline{\text{mod}} b = a - b \left\lfloor \frac{a}{b} + \frac{1}{2} \right\rfloor$$

The  $\overline{\text{mod}}$  operation can be expressed in terms of the regular mod operation as follows:

$$a \overline{\text{mod}} b = \begin{cases} a \text{ mod } b, & \text{if } (a \text{ mod } b) < b/2 \\ (a \text{ mod } b) - b, & \text{if } (a \text{ mod } b) \geq b/2 \end{cases}$$

This infers that the absolute value of  $(a \overline{\text{mod}} b)$  is always less than, or equal to  $b/2$ .

Equality constraint elimination is performed as follows [12]. To eliminate the equality

$\sum_{k=0}^n a_k i_k = 0$ , we first check if there exist  $j \neq 0$ , such that  $|a_j| = 1$ . If such  $a_j$  exists, the constraint can be eliminated by solving the constraint for the  $i_j$ , and substituting the result into all other constraints.

Otherwise, we choose the coefficient  $a_j$ ,  $j \neq 0$ , with the smallest absolute value. Let  $m = |a_j| + 1$ . We introduce a new constraint with new variable  $\sigma$ :

$$m\sigma = \sum_{k=0}^n (a_k \overline{\text{mod } m}) i_k$$

Note that  $\sigma$  has to be an integer, since all other variables in the new constraint are integers. It can also be shown, by substituting  $m = |a_j| + 1$ , that  $(a_j \overline{\text{mod } m}) = -\text{sign}(a_j)$ . We now solve the new constraint for  $i_j$ :

$$i_j = -\text{sign}(a_j)m\sigma + \sum_{k=0, k \neq j}^n \text{sign}(a_j) (a_k \overline{\text{mod } m}) i_k$$

and substitute the result in all constraints. In the original constraint, this substitution produces:

$$-|a_j|m\sigma + \sum_{k=0, k \neq j}^n (a_k + |a_j|(a_k \overline{\text{mod } m})) i_k = 0$$

Since  $|a_j| = m - 1$ , this is equal to:

$$-|a_j|m\sigma + \sum_{k=0, k \neq j}^n ((a_k - (a_k \overline{\text{mod } m})) + m(a_k \overline{\text{mod } m})) i_k = 0$$

All terms in the above equation are divisible by  $m$ , since:

$$(a_k - (a_k \overline{\text{mod } m})) = m \left\lfloor \frac{a_k}{m} + \frac{1}{2} \right\rfloor$$

Dividing the equation by  $m$  produces:

$$-|a_j|\sigma + \sum_{k=0, k \neq j}^n \left( \left\lfloor \frac{a_k}{m} + \frac{1}{2} \right\rfloor + (a_k \overline{\text{mod } m}) \right) i_k = 0$$

The absolute value of the coefficient of  $\sigma$  in the above equation is the same as the

absolute value of the original coefficient of  $i_j$ . Since the absolute value of  $(a_k \overline{\text{mod}} m)$  is always less than or equal to  $m/2$ , and  $m \geq 3$  by definition, all other coefficients are reduced to at most  $2/3$ 'rds of their value in the original constraint. Therefore, repeated application of the above procedure will eventually force one of the coefficients to become 1, and allow us to eliminate the constraint.

The complexity of the equality elimination depends on the value of the coefficients. However, coefficients decrease exponentially from one step to another, thus amortizing the effect on the complexity.

#### *4.3 Inequality constraints*

Once all equality constraints have been eliminated, we deal with inequality constraints. General approach based on a Fourier-Motzkin variable elimination is described in the following section. However, there are some special cases that can be handled more easily.

If two inequality constraints directly contradict one another, the original set of constraints does not have a solution. For instance, inequalities  $2x + 3y \geq 0$  and  $-2x - 3y - 1 \geq 0$  can never hold simultaneously.

Since the equalities can be handled more efficiently than inequalities, pairs of tight inequalities are replaced by equality. Inequalities  $2x + 3y \geq 0$ , and  $-2x - 3y \geq 0$  can be replaced by an equality  $2x + 3y = 0$ , since that is the only feasible solution to given set of inequalities. Equalities are then dealt with as described in the previous subsection.

Some of the inequalities can be redundant. For instance, set of inequalities  $2x + 3y \geq 0$ , and  $2x + 3y + 2 \geq 0$  can be replaced by a single inequality  $2x + 3y \geq 0$ , since it covers both constraints.

If the resulting set of inequality constraints contains only a single variable, without contradicting constraints, algorithm reports that the problem has integer solutions. Otherwise, Fourier-Motzkin variable elimination is applied iteratively, until only one variable is left.

#### *4.4 Fourier-Motzkin Variable Elimination*

Fourier-Motzkin variable elimination eliminates a variable from a linear programming problem. Intuitively, Fourier-Motzkin variable elimination finds the  $n-1$  dimensional shadow cast by an  $n$  dimensional object [12].

Figure 1 shows the square defined by the set of constraints  $y \leq -x + 9$ ,  $y \leq x$ ,  $y \geq -2x + 10$ , and  $y \geq x - 4$ . Eliminating the variable  $x$  from given set of constraints is equal to calculating the shadow cast along the  $x$  axis, by the square defined by given constraints. This can be done by first solving all inequalities for  $x$ , and then combining all lower bounds on  $x$  with all upper bounds. For the example in Figure 1, this results in  $x \leq -y + 9$ ,  $x \leq y + 4$ ,  $x \geq y$ , and  $2x \geq -y + 10$ . Combining the bounds produces:  $0 \leq 4$ ,  $y \leq 9/2$ ,  $y \geq 2/3$ ,  $y \leq 8$ . Some constraints are redundant, and can be eliminated. The resulting shadow is  $2/3 \leq y \leq 9/2$ .

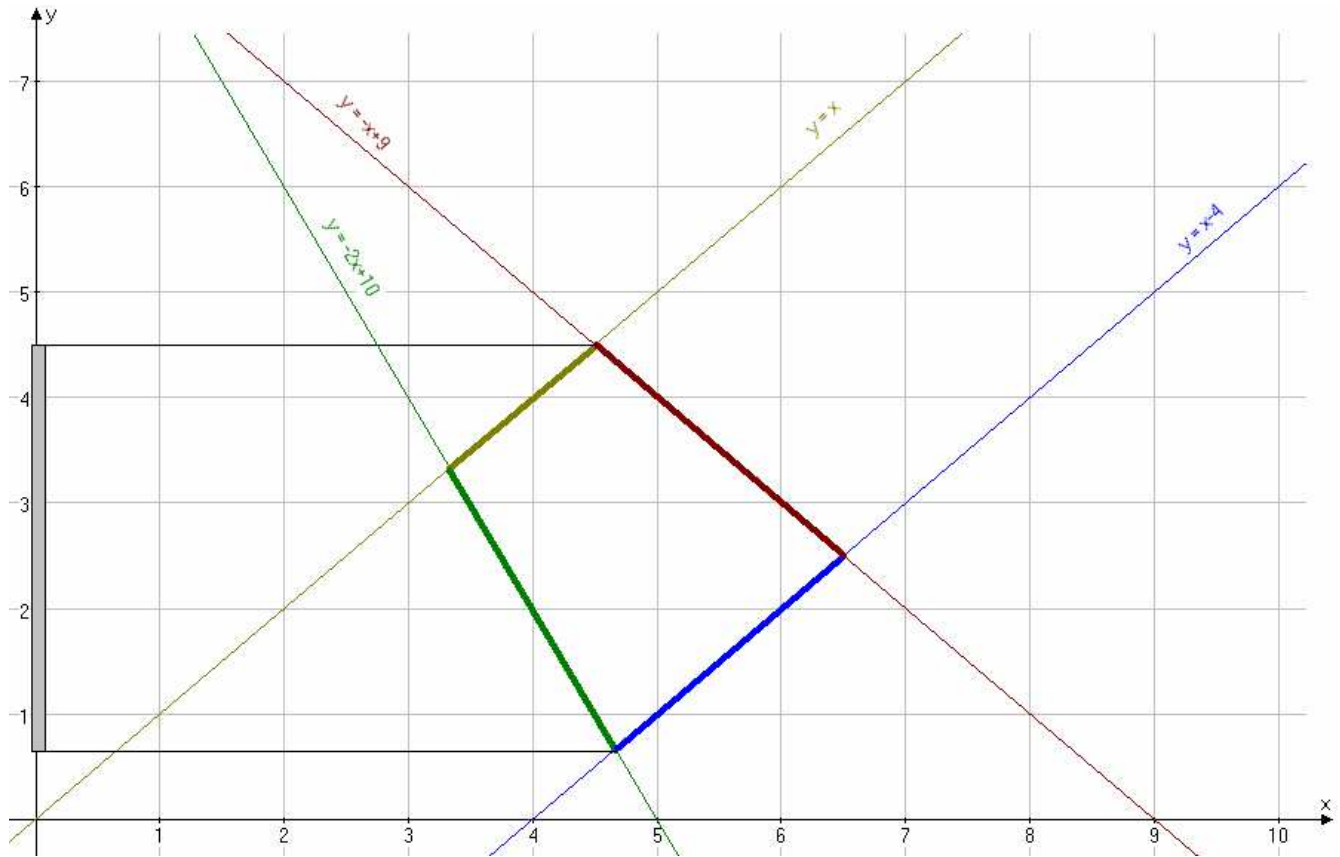


Figure 1. Shadow cast by square in a plane

Generally, we consider two constraints on  $x$ , a lower bound  $bx \geq g$ , and an upper bound  $ax \leq f$ , where  $a$  and  $b$  are positive integers, and  $f$  and  $g$  are functions of other problem variables. Combining these two constraints produces  $ag \leq abx \leq bf$ . The constraint  $ag \leq bf$ , does not contain variable  $x$ , and represents the shadow of the intersection of two constraints. By combining the shadow of the intersection of each pair of upper and lower bounds on  $x$ , we obtain a set of constraints that define the shadow of the original object [12]. This is not limited

to two dimensional objects, but can be applied to objects with any number of dimensions.

The shadow obtained by the procedure described above describes the set of real values of remaining variables, for which the original constraint set has real solutions. Therefore, Pugh [12] refers to it as a real shadow of the set of constraints. If there are no integer points in the real shadow of an object, there are no integer solutions to the set of constraints that define the object [12]. However, if there are integer solutions in the real shadow, we do not know if there are integer solutions to the original set of constraints. For instance, if the original object is very thin and wide, there may be many integer points in the real shadow, but none in the actual object [12]. Therefore, a way to determine an integer shadow of the object is desired.

Integer shadow is a shadow of the object, such that, for every integer point in the shadow, there is at least one corresponding integer point in the object above it, and vice versa. For an arbitrary object in  $n$  dimensions it is not always possible to determine the integer shadow directly. Pugh [12] has therefore introduced an algorithm to approximate the integer shadow.

Approximating the integer shadow can be visualized as finding the shadow of a translucent object, where thicker parts of the object cast a darker shadow. Dark shadow is a shadow of the object, such that, for every integer point in the shadow, there is an integer point in the object above it [12]. Unlike the integer shadow, if there is an integer point in the object, there may not be an integer point in the corresponding dark shadow.

One obvious way to define a dark shadow is the area below any part of the object that is at least one unit thick, along the dimension that is being eliminated. That part of the object has to contain at least one integer point. However, since all the coefficients in the constraints are integers, Pugh [12] uses a looser definition of the dark shadow.

Using the same notation as before, if there are integer points in the real shadow, then the constraint  $ag \leq bf$  has integer solutions. If, at the same time, there are no integer solutions in the original object, then there is no integer solution to the constraint  $ag \leq abx \leq bf$ . This implies that there is no multiple of  $ab$  between  $ag$  and  $bf$ . Then

$$ab i < ag \leq bf < ab (i + 1).$$

Where  $a$  and  $b$  are positive integers, and  $i = \lfloor g/b \rfloor$ . Since  $ab(i + 1) - bf \geq b$ ,  $ag - abi \geq a$ , and  $ab(i + 1) - abi = ab$ , then  $bf - ag \leq ab - a - b$ . If this is not valid (i. e.  $bf - ag \geq ab - a - b + 1$ ), then

there has to exist an integer solution in the original object. Therefore, the dark shadow of  $g \leq bx$ , and  $ax \leq f$  is:

$$bf - ag \geq (a - 1)(b - 1)$$

since  $ab - a - b + 1 = (a - 1)(b - 1)$ . In the case of  $a = 1$ , or  $b = 1$ , the dark shadow and the real shadow are identical. If the dark and real shadow resulting from each combination of an upper and lower bound are identical, the projection is called an exact projection [12]. If the projection is not exact, there are three possibilities [12]:

1. If there are no integer solutions to the real shadow, there can be no integer solutions to the original set of constraints.
2. If there are integer solutions to the dark shadow, there are integer solutions to the original set of constraints
3. If there are integer solutions to the real shadow, but no integer solutions to the dark shadow, integer solutions to the real shadow need to be checked for corresponding integer solutions in the original object. Since any part of the object that is at least one unit thick will cast the dark shadow, possible integer solutions to the original object must be closely nested between a lower and upper bound. Since all coefficients in the original constraint pair are integers, the solution has to lie on a (hyper)plane that is parallel to lower (or upper) bound.

According to the definition of the dark shadow, the following has to be valid:

$$ag \leq abx \leq bf \wedge 0 \leq bf - ag \leq ab - a - b.$$

First condition consists of upper and lower bounds combined. The second condition defines the area outside the dark shadow. These two conditions directly imply:

$$ag \leq abx \leq ag + ab - a - b$$

Therefore, the set of parallel planes that need to be checked are  $bx = g + i$ , for each  $i$ , such that  $0 \leq i \leq (ab - a - b)/a$ . We choose the largest coefficient  $a$  of  $x$  in any upper bound, which gives us the lowest upper bound. This process is repeated for each lower bound  $bx \geq g$ . Checking if the solution lies on any of the parallel lines is performed by solving the original problem combined with  $bx = g + i$ . Although these

steps are computationally expensive, most cases occurring in practice do not require usage of these methods [12].

The Fourier-Motzkin variable elimination can be summarized as follows [12]:

1. Choose the variable to eliminate. If possible, variable whose elimination will result in exact projection is chosen. If non-exact projection needs to be performed, the variable with lowest absolute value of coefficients is chosen.
2. Calculate the real and dark shadows of the object defined by the given set of constraints along selected dimension.
3. If the real and dark shadows are identical, there are integer solutions to the original set of constraints if and only if there are integer solutions to the shadow.
4. Otherwise, perform the non-exact projection, and check for the existence of the solutions.

#### *4.5 Distance and Direction Vectors*

The Omega test, as described in previous sections, simply decides if there exists an integer solution to the given set of constraints. In order to calculate distance and direction vectors, the Omega test needs to be adapted so it can be used for symbolic projection.

The input to the symbolic projection is the original set of constraints, and a designated set of protected variables. The Omega test projects the original set of constraints into constraints involving only variables from the protected set. Since the equality constraint elimination sometimes introduces new variables, projection results may be expressed in terms of other variables. In that case Omega test also produces equalities expressing protected variables in terms of introduced variables [12].

Distance and direction vectors can be calculated by introducing a new variable for the dependence distance for each loop in the loop nest, and performing projection on these variables. The resulting set of constraints defines the conditions on dependence distances.

Consider the following example:

```

DO i1 = 0, 2
  DO i2 = 0, 2
(S1)    A (i1, i2) = ...
(S2)    ... = A (i1 - i2, i1 + i2);
  ENDDO
ENDDO

```

By definition, there is a dependence between two array references  $S_1$  and  $S_2$  if there exist two iteration vectors  $\alpha$  and  $\beta$  such that  $\mathbf{L} \leq \alpha, \beta \leq \mathbf{U}$ , and  $f_k(\alpha) = g_k(\beta), \forall k, 1 \leq k \leq m$ . For the example above it results in the following set of constraints:

$$\begin{aligned} \alpha_1 &= \beta_1 - \beta_2 \\ \alpha_2 &= \beta_1 + \beta_2 \\ 0 &\leq \alpha_1, \alpha_2, \beta_1, \beta_2 \leq 2 \end{aligned}$$

We now introduce two variables that represent dependence distances, one for each of the loops in the nest:

$$\begin{aligned} \delta_1 &= \beta_1 - \alpha_1 \\ \delta_2 &= \beta_2 - \alpha_2 \end{aligned}$$

Projecting given set of constraints onto  $\delta_1$  and  $\delta_2$  produces:

$$\begin{aligned} -\delta_2 - 2 &\leq \delta_1 \leq -\delta_2 \\ -2 &\leq \delta_2 \leq 0 \\ \delta_1 - 2 &\leq \delta_2 \leq \delta_1 \\ 0 &\leq \delta_1 \leq 2 \end{aligned}$$

The projected system may, in some cases, be a better way to describe dependences, since it typically contains more information than distance or direction vectors. The projected system can also be used to determine the dependence direction and distance vectors using the following procedure [12]. The projected system is first scanned for constraints that involve only one variable. If such constraint describes the distance/direction vector precisely, the distance/direction in that dimension is determined. For other constraints, we unprotect any

dependence distance variable that is uncoupled, or whose sign is completely determined. If the sign of the variable is not fully determined, we analyze the subproblems for all possible signs of the variable. If coupled variable was unprotected, the problem needs to be projected onto the protected variables, and the process repeated.

In our example, we choose to analyze two possible signs for  $\delta_1$ : zero, and positive. If the sign of  $\delta_1$  is zero,  $\delta_2$  can be either zero, or negative. If  $\delta_1$  is positive,  $\delta_2$  can only be negative. This results in the following set of legal direction vectors  $(=, =)$ ,  $(=, <)$ ,  $(<, >)$ . These direction vectors describe the corresponding dependences correctly.

Symbolic projection can also be used to handle symbolic constants. If the subscript expression contains the constant whose value is not known at compile time, the system can be projected on such a variable. The resulting condition imposed on the constant must be fulfilled, in order for the dependence to exist. Otherwise, there is no dependence.

The Omega test can also handle integer division, and integer remainder operations in subscript expressions. For integer division operation  $e = a \text{ div } m$ , new variable  $\sigma$ , and the inequality constraints  $0 \leq a - m\sigma \leq m - 1$  are introduced. In all the original constraints,  $\sigma$  is used instead of  $e$ . For the mod operation, the same inequality constraints are used, and  $a - m\sigma$  is used instead of  $e$ .

## 5 Rice Test

Unlike the Omega test, that formulates and solves the dependence testing problem purely mathematically, Rice test is based on classifying pairs of subscript expressions. Most of the classes of subscript expressions used in Rice test can be tested using exact and fast dependence tests. Goff et al [5] have shown that these references dominate scientific Fortran codes. However, the Rice test is not exact, since it uses some in-exact tests if necessary.

The goal of the Rice test is to construct the complete set of distance and direction vectors representing potential dependences between an arbitrary pair of array references [5]. In the rest of the section, we will use the term subscript to refer to the pair of subscript expressions in some dimension of the array reference, i. e. pair  $f_k(\mathbf{I})$ ,  $g_k(\mathbf{I})$  in the reference to the same array. We will also use term index to refer to the loop index variable.

## 5.1 Classification

For the purpose of the Rice test, subscripts are classified by two orthogonal criteria: complexity and separability [5].

Subscript complexity is defined by the number of indices that appear within subscript. A subscript is said to be ZIV (Zero Index Variable) if it contains no index in either subscript expression. A subscript is said to be SIV (Single Index Variable) if only one index occurs in the subscript. Any subscript containing more than one index is MIV (Multiple Index Variable). For instance, if the statement  $A(5, i+1, j) = A(N, i, k-1) + C$ , is enclosed in a loop nest with index variables  $i, j$ , and  $k$ , the first subscript is ZIV, second subscript is SIV, and the third subscript is MIV [5].

A subscript is said to be separable if its indices do not occur in other subscripts. Two subscripts containing the same index are said to be coupled. If all subscripts are separable, simple technique, called subscript-by-subscript testing can be used for dependence testing. The technique consists of testing each subscript separately, and intersecting the resulting sets of distance/direction vectors. Since subscripts are independent of each other, the subscript-by-subscript testing yields the exact result [5].

Subscript-by-subscript testing may yield false dependences if applied to the coupled subscripts. Assuming statement  $A(i+1, i+2) = A(i, i) + C$ , is enclosed within loop with index variable  $i$ , subscript-by-subscript testing will yield the single direction vector ( $<$ ) for both subscript position. However, it is obvious that the dependence does not exist.

The following example illustrates that the complexity and separability are indeed orthogonal criteria. Consider two statements:

$$A(i, j+1, h) = A(N, k, h) + C;$$

$$A(i, i+1, h) = A(i, i, h) + C;$$

assuming both statements are enclosed in a loop nest with indices  $i, j, k$ , and  $h$ , but each statement in its own loop nest. Second subscript in the first statement is a MIV subscript, but all subscripts are separable, and subscript-by-subscript testing will produce exact results. In the second statement, all subscripts are SIV, but the first and the second subscript are coupled, and the subscript-by-subscript testing would report false dependences. The classification presented in this subsection is a basis for subscript partitioning.

## 5.2 Subscript partitioning

Rice test takes advantage of separability by partitioning subscripts into separable and minimal coupled groups. A coupled group is minimal if it cannot be partitioned into two non-empty subgroups, with distinct set of indices. Once all the subscripts are partitioned, each separable subscript, and each minimal coupled group have completely disjoint set of indices. Each partition can then be tested separately, and the resulting distance/direction vectors can be merged, without any loss of precision [5].

Rice test uses the following subscript partitioning algorithm:

```
partition ( $S_1 = \{f_1(I), g_1(I)\}, \dots, S_m = \{f_m(I), g_m(I)\}$ )
/* Returns the set of subscript partitions  $P = \{P_1, P_2, \dots, P_{m'}\}, P_k = \{S_1, S_2, \dots\}, m' \leq m$  */
for k = 1 to m
     $P_k = \{S_k\}$ ;
endfor
for each index  $i_k, k = 1$  to n
    z = <none>;
    for each  $P_j \in P$ 
        if  $\exists S_u \in P_j$  such that  $S_u$  contains  $i_k$ 
            if z = <none>
                z = j;
            else
                 $P_z = P_z \cup P_j$ ;
                 $P = P - P_j$ ;
            endif
        endif
    endfor
endfor
return P;
```

The above algorithm searches for each index in every subscript, and puts all subscripts that contain that index in the same minimal coupled group.

After testing each partition, results need to be merged. The merge operation is simply a Cartesian product of direction/distance vectors produced by individual tests. In the following

example:

```
DO i1 = 0, N
  DO i2 = 0, N
(S1)    A (i1+1, 5) = ...
(S2)    ... = A (i1, N);
  ENDDO
ENDDO
```

there are two partitions:  $P_1 = \{\{i_1 + 1, i_1\}\}$ , and  $P_2 = \{\{5, N\}\}$ . The first partition yields the direction vector ( $<$ ) for the loop with index  $i_1$ . The second partition does not yield direction vector, since it does not involve any indices (It does, however, impose the condition  $N=5$ , in order for the dependence to exist). Since  $i_2$  does not appear in any subscript, the full set of direction vectors needs to be assumed:  $\{<, =, >\}$ . The full set of direction vectors is sometimes represented by the character '\*' [1]. Therefore, the Cartesian product of direction vectors for  $i_1$  and  $i_2$  yields the set of direction vectors  $\{(<, <), (<, =), (<, >)\}$ , or  $\{(<, *)\}$ . If any of the tests proved independence, merge is not necessary, since the overall test results in independence.

### 5.3 Separable Subscript Tests

In this subsection we present several tests applicable to some special cases of separable subscripts. Subscripts that do not fall into any of these categories can be handled by general tests. SIV subscripts can be tested by the Single-Index exact test. For SIV subscripts involving symbolic constants, special case of Banerjee test is used. Generalized GCD test is used for MIV subscripts [5].

#### 5.3.1 ZIV test

If the subscript is ZIV, and subscript expressions are constants, constant test is used. The ZIV test can also handle symbolic constants. If the difference between subscript expressions simplifies to a non-zero constant, independence is proven. If independence cannot be proven, the subscript does not produce any direction vectors, and may be ignored.

#### 5.3.2 Strong SIV Test

An SIV subscript for an index  $i_k$  is said to be strong if it has the form  $\{a*i_k + c_1, a*i_k + c_2\}$ . Strong SIV subscript test is equivalent to the Lamport's test, described in the

section 3.2.

### 5.3.3 Weak-zero SIV Test

An SIV subscript for an index  $i_k$  is said to be weak-zero if it has the form  $\{c_1, a*i_k + c_2\}$ . In order for the dependence to exist,  $i_k = (c_1 - c_2)/a$  has to be an integer within the loop bounds. The weak-zero SIV test finds the dependences caused by a particular iteration  $i_k = (c_1 - c_2)/a$ . In scientific codes, the dependence is usually caused by the first or the last iteration, and may be eliminated by applying the loop peeling transformation [1, 5].

### 5.3.4 Weak-crossing SIV Test

An SIV subscript for an index  $i_k$  is said to be weak-crossing if it has the form  $\{-a*i_k + c_1, a*i_k + c_2\}$ . Dependence can exist only if  $(c_2 - c_1)/a$  is an integer. Weak-crossing SIV subscripts cause loop-carried dependences whose endpoints all cross iteration  $(c_2 - c_1)/2a$ . These dependences may be eliminated by the loop distribution (splitting) transformation [1, 5].

## 5.4 Delta test

Delta test is a multiple subscript test designed to be exact, yet efficient for common coupled subscripts. The main idea behind the Delta test is that constraints derived from SIV subscripts may be propagated into other subscripts in the same coupled group efficiently, usually without any loss of precision.

### 5.4.1 Constraints

Constraints are assertions on indices derived from subscripts. For instance, the SIV subscript  $\{a_1*i_k + c_1, a_2*i_k + c_2\}$  generates the constraint  $a_1*\alpha_k - a_2*\beta_k = c_2 - c_1$  for index  $i_k$ . A constraint vector  $C = (\delta_1, \delta_2, \dots, \delta_h)$  is the vector with one constraint for each of the  $h$  indices in the coupled subscript group. The constraint vectors can easily be converted to distance and direction vectors, since they describe the relations between iteration vectors causing dependences [5].

A constraint  $\delta_k$  may have one of the following forms [5]:

- dependence line – a line ( $a\alpha_k + b\beta_k = c$ ) representing the dependence equation
- dependence distance – a value ( $d$ ) of the dependence distance; it is equivalent to the dependence line ( $\alpha_k - \beta_k = -d$ )

- dependence point – a point  $(\alpha_k, \beta_k)$  representing dependence from iteration  $\alpha_k$  to  $\beta_k$ .

Dependence distances and lines are result of SIV tests. Dependence points result from the constraint intersection.

All dependence equations in a coupled subscript group must hold simultaneously for the dependence to exist. If the constraint intersection operation is defined properly, dependence equations can be tested separately, and resulting constraint vectors intersected. Constraint intersection takes two constraints, and produces a new constraint, equivalent to the original constraints holding simultaneously. If the result of the constraint intersection is the empty set, then the original set of constraints can never hold simultaneously, and hence no dependence can exist. Delta test employs constraint intersection only for SIV subscripts in a coupled group.

Result of the constraint intersection depends on the type of constraints being intersected. Two dependence distances need to be equal in order for the dependence to exist. The same is valid for two dependence points. Two dependence lines can produce dependence only if they intersect. The result of the intersection is the intersection point. A dependence point and a dependence line produce dependence if the point lies on the line. Intersection result is the dependence point.

Constraint intersection is performed using the following algorithm:

```
intersect ( $\delta_1, \delta_2, L_k, U_k$ )
```

```
/*  $L_k, U_k$  are lower and upper bound for the index that occurs in  $\delta_1$  and  $\delta_2$ . Algorithm returns new constraint  $\delta$ , or  $\emptyset$  */
```

```
/*If either  $\delta_1$  or  $\delta_2$  imposes no constraint, return the other one */
```

```
if  $\delta_1 = \langle \text{none} \rangle$ 
```

```
    return  $\delta_2$ ;
```

```
endif
```

```
if  $\delta_2 = \langle \text{none} \rangle$ 
```

```
    return  $\delta_1$ ;
```

```
endif
```

```

/* If both constraints are distances, they need to be the same for dependence to exist */
if  $\delta_1 = (d_1)$  and  $\delta_2 = (d_2)$ 
    if  $d_1 = d_2$ 
        return  $(d_1)$ ;
    else
        return  $\emptyset$ 
    endif
endif

/* If both constraints are dependence points, they need to be equal for dependence to exist */
if  $\delta_1 = (x_1, y_1)$  and  $\delta_2 = (x_2, y_2)$ 
    if  $x_1 = x_2$  and  $y_1 = y_2$ 
        return  $(x_1, y_1)$ ;
    else
        return  $\emptyset$ ;
    endif
endif

/* If both constraints are dependence lines/distances, find the intersection */
if  $\delta_1 = (a_1x + b_1y = c_1)$  and  $\delta_2 = (a_2x + b_2y = c_2)$ 
    /* If slopes are equal, the lines are parallel */
    if  $a_1/b_1 = a_2/b_2$ 
        /* If lines are parallel, they either do not intersect, or it is a single line */
        if  $c_1/b_1 = c_2/b_2$ 
            return  $(a_1x + b_1y = c_1)$ ;
        else
            return  $\emptyset$ ;
        endif
    endif
    calculate  $(x_1, y_1) = \text{intersection of } \delta_1, \delta_2$ ;
    if  $L_k \leq x_1 \leq U_k$  and  $L_k \leq y_1 \leq U_k$  and  $x_1$  and  $x_2$  are both integers
        return  $(x_1, y_1)$ ;
    else
        return  $\emptyset$ ;

```

```

    endif
endif
/* The only option left is that one of the constraints is a dependence line/distance, while the
other one is a dependence point. Without the loss of generality assume
 $\delta_1 = (a_1x + b_1y = c_1)$  and  $\delta_2 = (x_1, y_1)$ . There is a dependence iff the dependence point lies on
the dependence line/distance */
if  $a_1x_1 + b_1y_1 = c_1$ 
    return  $(x_1, y_1)$ ;
else
    return  $\emptyset$ ;
endif

```

Notice the difference between  $\emptyset$  and <none> notations. The notation <none> means there is no constraint imposed on the particular index. The notation  $\emptyset$  means the imposed constraints are not satisfiable, and there is no dependence.

#### 5.4.2 Delta Test

The Delta test first analyzes all separable ZIV and SIV subscripts using ZIV and SIV tests. If any of the tests yields independence, the test reports independence. Otherwise, all SIV subscripts are turned into constraints, and propagated into MIV subscripts. If the propagation process results with new SIV subscripts, the process is repeated until no new SIV subscripts are produced.

Next, MIV subscripts are scanned for RDIV (Restricted Double Index Variable) subscripts. RDIV subscripts have form  $\{a_j * i_j + c_j, a_k * i_k + c_k\}$ , and are similar to SIV subscripts, except that  $i_j$  and  $i_k$  are distinct indices. By observing different bounds for  $i_j$  and  $i_k$ , SIV test may be extended to exactly test RDIV subscripts [5]. Testing the RDIV subscripts produces new constraints, which are then propagated into remaining MIV subscripts.

At the end, remaining MIV subscripts are tested subscript-by-subscript, possibly resulting in false dependences. Described procedures are performed by the Delta test algorithm [5]:

```

delta (P = {P1, P2, ...,})
/* Pk = {S1, S2, ...} – minimal coupled groups
   Algorithm returns distance/direction vector */
/* Initialize constraint vector C to <none> */
for each δk ∈ C, k = 1 to n'      /* n' is the number of indices that occur in all Pk ∈ P */
    δk = <none>;
endfor
while ∃ untested SIV subscripts
    apply SIV tests to all untested SIV subscripts;
    return independence, or derive new constraint vector C';
    for each k = 1 to n'
        δk' = intersect (δk, δk', Li, Ui); /* δk ∈ C, δk' ∈ C', Li, Ui - lower and upper
                                                    bound on index occurring in δk and δk' */
    endfor
    if ∃ δk' ∈ C', δk' = ∅
        return independence;
    endif
    if C ≠ C'
        C = C';
        propagate constraints from C into MIV subscripts;
        apply ZIV test to possibly newly created ZIV subscripts;
        return independence, or continue;
    endif
endwhile
while ∃ untested RDIV subscripts
    test and propagate RDIV constraints;
endwhile
test remaining MIV subscripts;
    intersect resulting direction vectors with C and store it into C;
return distance/direction vectors from C;

```

Constraint propagation algorithm used by Delta test is described in more details in the following subsection.

### 5.4.3 Constraint propagation

Constraint propagation algorithm takes constraint vector  $C$ , and propagates the constraints from  $C$  into given MIV subscript. The result of the propagation is constrained ZIV, SIV, or MIV subscript. The constraint propagation is performed as follows [5]:

```
propagate ( $S = \{a_1 i_1 + \dots + a_h i_h + e, a_1' i_1 + \dots + a_h' i_h\}$ ,  $C = (\delta_1, \delta_2, \dots, \delta_h)$ )
```

```
/* Algorithm returns constrained ZIV, SIV, or MIV subscript */
```

```
for each index  $i_k$  with non-zero  $a_k$  or  $a_k'$ 
```

```
  if  $\delta_k = (\alpha_k, \beta_k)$  /* dependence point */
```

```
     $e = e + a_k \alpha_k - a_k \beta_k$ ;  $a_k = 0$ ;  $a_k' = 0$ ;
```

```
  else if  $\delta_k = (d)$  /* dependence distance */
```

```
     $e = e - a_k d$ ;  $a_k = 0$ ;  $a_k' = a_k' - a_k$ ;
```

```
  else if  $\delta_k = (u \alpha_k + v \beta_k = c)$  /* dependence line */
```

```
    if  $u = 0$ 
```

```
       $e = e - a_k' c / v$ ;  $a_k' = 0$ ;
```

```
    else if  $v = 0$ 
```

```
       $e = e + a_k c / u$ ;  $a_k = 0$ ;
```

```
    else if  $u = v$ 
```

```
       $e = e + a_k c / u$ ;  $a_k = 0$ ;  $a_k' = a_k' + a_k$ ;
```

```
    else
```

```
      temp =  $a_k$ ;
```

```
      for each  $\tau \in \{a_1, \dots, a_h, a_1', \dots, a_h', e, e'\}$ 
```

```
         $\tau = u * \tau$ ;
```

```
      endfor
```

```
       $e = e + temp * c$ ;  $a_k' = a_k' + temp * v$ ;  $a_k = 0$ ;
```

```
    endif
```

```
  endif
```

```
endfor
```

Constraint propagation is performed differently for each type of the constraint. Generally, propagation problem can be analyzed on a subscript pair  $\{a_0 + a_k i_k + e,$

$a_0' + a_k' i_k' + e'\}$ , and constraint  $\delta_k$ , where  $a_0 = \sum_{j=1, j \neq k}^h a_j i_j$  and  $a_0' = \sum_{j=1, j \neq k}^h a_j' i_j'$ . In order for

the original subscript pair to cause the dependence, there have to exist two iterations  $\alpha_k$ ,  $\beta_k$ , such that dependence equation  $a_0 + a_k\alpha_k + e = a_0' + a_k'\beta_k + e'$  is satisfied. Propagating the constraint into such an equation is straightforward.

If the constraint is a dependence point, we simply need to substitute the dependence point into the equation, which results in  $a_0 + e + a_k\alpha_k - a_k'\beta_k = a_0' + e'$ . Subexpression  $e + a_k\alpha_k - a_k'\beta_k$  is now a constant, and it can be treated as a new value of  $e$ . Product terms  $a_k i_k$  and  $a_k' i_k'$  have been eliminated from the subscript pair. The algorithm performs these operations by assigning new values to the  $e$ ,  $a_k$ , and  $a_k'$ .

If the constraint is a dependence distance ( $d$ ), it imposes the constraint  $\beta_k - \alpha_k = d$ . Using simple algebraic manipulations, the dependence equation can be re-written in the form  $a_0 + e - a_k d = a_0' + (a_k' - a_k) i_k' + e'$ . The algorithm assigns new values to the appropriate variables, according to given equation.

Similar manipulations are performed when the constraint is dependence line  $u\alpha_k + v\beta_k = c$ . In order to propagate the constraint into subscript pair, dependence equation is first multiplied by  $u$ , resulting in  $ua_0 + ua_k\alpha_k + ue = ua_0' + ua_k'\beta_k + ue'$ . After substituting the value for  $u\alpha_k$ , and simple algebraic manipulations dependence equation turns into  $ua_0 + a_k c + ue = ua_0' + (ua_k' + a_k v)\beta_k + ue'$ . Two special cases of a dependence line constraint ( $u = 0$ , and  $v = 0$ ) are resolved in a similar fashion. Note that  $c/u$  and  $c/v$  are always integers. If they were not, the SIV test would return independence.

Constant propagation will eliminate both instances of index  $i_k$  in a subscript pair if the constraint is a dependence point, or the constraint is a dependence distance, and  $a_k' = a_k$ . Although in a general case the algorithm may only eliminate one instance of an index, it will increase the precision of other tests. For instance, in a minimal coupled group  $\{\{i_1, i_1\}, \{i_1 + 2i_2, -i_1 + 2i_2 + 5\}\}$ , applying the GCD test on each subscript separately does not prove independence. However, propagating the dependence distance constraint (0) from the first subscript, results in a subscript pair  $\{2i_2, -2i_1 + 2i_2 + 5\}$ . The GCD test can now show the independence, since the GCD of all coefficients is 2, which does not evenly divide 5.

Constraint propagation algorithm can only propagate SIV constraints. Propagating MIV constraints is expensive in general case [5]. However, special case consisting of coupled RDIV subscripts can be propagated. Common case of RDIV subscripts occurs

in a matrix transpose operation  $A(i_1, i_2) = A(i_2, i_1)$ . Subscript partitioning produces a partition  $\{\{i_1, i_2\}, \{i_2, i_1\}\}$ . Dependence equations for this example are  $\alpha_1 = \beta_2$ , and  $\alpha_2 = \beta_1$ . Subscript-by-subscript testing would report the dependence for all possible direction vectors. However, introducing new variables that represent dependence distances, and propagating that variable into both subscripts simplifies the problem. Since  $d_1 = \beta_1 - \alpha_1$ , and  $d_2 = \beta_2 - \alpha_2$ , propagating these new constraints into original dependence equations yields equations  $\alpha_1 = d_2 + \alpha_2$ , and  $\alpha_2 = d_1 + \alpha_1$ . Propagating one of the constraints into another results in final constraint  $d_1 + d_2 = 0$ . As a result, distance vectors must have the form  $(d, -d)$ . Therefore, the final result are direction vectors  $(<, >)$ , and  $(=, =)$ .

The idea of introducing new variable that represents the distance is similar to the symbolic projection employed by the Omega test. However, the Omega test uses this technique to calculate distance/direction vectors, not for dependence testing. Delta test, on the other hand, has to employ this technique for the test to be exact.

#### 5.4.4 Rice Test Precision

The precision of the Rice test depends on the type of the coupled subscripts being tested [5]. All of the ZIV and SIV tests employed by the Rice test are exact. Result of the Rice test is therefore exact, if all subscripts were tested using ZIV and SIV tests.

Delta test can handle three types of constraints: dependence points, dependence distances, and dependence lines. However, complex iterations spaces may impose constraints not utilized by the Delta test. For instance, the loop nest

```
DO i1 = 0, N
  DO i2 = i1, N
    ... /* Some code with array references */
  ENDDO
ENDDO
```

will, along with other constraints, produce constraint  $i_2 \geq i_1$ , which the Delta test cannot handle.

Separable MIV subscripts that remain after the Delta test has been applied are tested using Generalized GCD test. The precision of the Rice test is therefore limited by the precision of the Generalized GCD test. During constraint propagation, it is sometimes

impossible to eliminate both instances of an index. Also, constraints from general MIV subscripts are not propagated. As a result, coupled MIV subscripts may remain at the end of the Delta test. In such cases, algorithm needs to report dependence for all possible directions of indices occurring in remaining subscripts. Alternatively, more expensive techniques that are always exact, could be employed [5].

## 6 Memoization to Improve Performance

Maydan et al [9] introduced another data dependence test using divide and conquer approach. They suggest that the choice of individual tests is not very important, providing the simple tests are exact for subscripts often occurring in practice. They also show that the same subscript expressions tend to repeat throughout the program. Therefore, the same tests are often repeated for the same inputs. Applying memoization, i. e. storing the results of previous tests, significantly reduces the number of tests performed.

In this section we present the tests not encountered in other algorithms: single variable per constraint test, acyclic test, and simple loop residue test. We also show how memoization is used to improve performance

### 6.1 Single Variable Per Constraint Test

Single variable per constraint test assumes all equality constraints have been eliminated. A simple way to eliminate the constraint  $a_1 i_1 + \dots + a_n i_n = a_1 'i_1' + \dots + a_n 'i_n'$  is to introduce two inequality constraints  $a_1 i_1 + \dots + a_n i_n \leq a_1 'i_1' + \dots + a_n 'i_n'$ , and  $a_1 i_1 + \dots + a_n i_n \geq a_1 'i_1' + \dots + a_n 'i_n'$ . Although such an algorithm is easy to implement, it results in significant increase in the number of constraints. Maydan et al suggest using the Banerjee's Extended GCD test for equality constraint elimination, which produces fewer constraints.

If each inequality constraint contains only single variable, the dependence testing can be performed exactly. Each constraint can be viewed as a lower or upper bound on a variable. Traversing constraints one by one, and noting the tightest constraints on each loop index variable, produces upper and lower bound on each of the variables. If any lower bound is greater than the corresponding higher bound, the test returns independence, since the system is unsatisfiable. Otherwise, there is a dependence for the values of indices between the lower and upper bounds.

Single variable test is also applicable to many common multi-dimensional cases, including

coupled subscripts, as shown in the following example [9]:

```

DO i1 = 1, 10
    DO i2 = 1, 10
        A (i1, i2) = A (i2 + 10, i1 + 9);
    ENDDO
ENDDO

```

The original set of constraints is  $\alpha_1 = \beta_2 + 10$ ,  $\alpha_2 = \beta_1 + 9$ ,  $1 \leq \alpha_1, \alpha_2, \beta_1, \beta_2 \leq 10$ . Substituting  $\alpha_2$  and  $\beta_2$  from equality constraints to the inequality constraints produces the set of inequality constraints  $1 \leq \alpha_1 \leq 10$ ,  $1 \leq \beta_1 + 9 \leq 10$ ,  $1 \leq \beta_1 \leq 10$ ,  $1 \leq \alpha_1 - 10 \leq 10$ . The tightest lower and upper bound on  $\alpha_1$  are  $11 \leq \alpha_1 \leq 10$ , resulting from the first and the last constraint. Since lower bound on  $\alpha_1$  is greater than the upper bound, it is obvious that this constraint cannot be satisfied. Therefore, no dependence exists.

## 6.2 Acyclic Test

The acyclic test handles the constraints involving more than one variable. The constraint set is first scanned for variables which are only constrained in one direction. If the variable  $i_k$  is bounded only from above, it can be set to its lower bound determined by the single variable per constraint test. For instance, if set of constraints  $1 \leq i_1 \leq 10$  and  $i_1 \leq i_2 + i_3 + 4$ , has a solution for  $i_1 > 1$ , then setting the  $i_1$  to 1 does not violate any constraints. Similarly, if the variable is bounded only from below, it can be set to its upper bound.

Setting the variable to its upper or lower bound effectively eliminates the variable. Procedure can be repeated until the independence is proven by contradicting constraints, or no more variables constrained only in one direction can be found. Maydan et al [9] show that the problem of finding variables constrained in one direction is equivalent to finding leaf nodes in the appropriately constructed directed graph. If the graph has no cycles, the system can be solved exactly.

## 6.3 Simple Loop Residue Test

Simple loop residue test can be applied to the system of constraints, which are all of the form  $i_j \leq i_k + c$ . The idea is to create a directed graph with a node for each variable. For the inequality  $i_j \leq i_k + c$ , the edge from the node  $i_j$  to  $i_k$ , with weight  $c$  is added to the graph. Assume the constraint set contains another inequality  $i_k \leq i_g + d$ . By transitivity, this implies

that  $i_j \leq i_g + c + d$ . Since the graph will contain edges  $(i_j, i_k)$  with weight  $c$ , and  $(i_k, i_g)$  with weight  $d$ , the relation between  $i_j$  and  $i_g$  is represented by the path from node  $i_j$  to  $i_g$  and its weight.

The graph also contains a special node  $i_0$ , to handle the constraints with only one variable. The constraint  $i_k \leq c$  is represented by an edge from  $i_k$  to  $i_0$ , with the weight of  $c$ . The constraint of the form  $i_k \leq c$  is equivalent to  $-c \leq -i_k$ , and is therefore represented by an edge from  $i_0$  to  $i_k$ , with the weight  $-c$ .

The path in the graph from node  $i_j$  to  $i_k$  of the total weight  $c$  implies the constraint  $i_j \leq i_k + c$ , which is equivalent to  $i_j - i_k \leq c$ . The cycle in the graph, therefore implies the constraint of the form  $i_j - i_j \leq c$ , or  $0 \leq c$ . Therefore, all cycles in the graph need to have positive weight, for the dependence to exist. If any cycle has a negative weight, there can be no dependence [9].

The algorithm can be extended to handle constraints of the form  $ai_j \leq ai_k + c$ . This is equivalent to the constraint  $a(i_j - i_k) \leq c$ , which is in turn equivalent to  $(i_j - i_k) \leq \lfloor c / a \rfloor$ . This constraint is now in the form acceptable by simple loop residue test. Notice that  $c/a$  is replaced by  $\lfloor c/a \rfloor$ , which is equivalent to the constraint tightening performed by the normalization step of the Omega test (see section 4.1).

#### 6.4 Memoization

Many array references in programs are very simple, and the same subscript expressions tend to repeat frequently [9]. For instance, bounds are often of the form  $i_k = 1$  to  $N$ . Although upper bound  $N$  can vary significantly across programs, it is often the same for many loops in a single program. Storing the results of previous tests, and returning stored results when the test is called for the same input again should reduce number of tests performed [9].

Test results are stored in a hash table for a quick access. Since GCD test does not consider loop bounds, two hash tables are used; one using loop bounds, and one not. Maydan et al use simple hashing scheme, with hashing function that avoids collision of symmetrical or partially symmetrical references. Example of symmetrical references is  $\{i_1, i_1 + 1\}$  and  $\{i_1 + 1, i_1\}$ .

The effectiveness of the memoization scheme could be further improved by eliminating the loop bound constraints on loop indices that are not used in subscript expressions [9]. As shown earlier (see section 5.2), all possible direction vectors need to be assumed for such loops. Another opportunity for improvement are symmetrical references. Dependence

equations  $\alpha_1 = \beta_1 + d$ , and  $\beta_1 = \alpha_1 + d$  are equivalent, except that the direction/distance vectors have opposite signs.

Storing the hash table across the compilations could significantly reduce the cost of incremental compilation. In addition, a set of benchmarks could be used to set up a standard hash table with subscripts that often occur in practice, which could then be used by all programs. Even with this simple scheme, experiments show that memoization reduces the total number of tests applied from 5,679 to 332 for the PERFECT Club benchmark set [9].

Concept similar to memoization was proposed by Eisenbeis and Sogno [4] for the GCD test. They suggest storing a table of precomputed GCD's in memory. The table should contain a GCD of every pair of integers smaller than a given value  $t$ . Computing the GCD of the numbers that are less than  $t$  can then be done in a constant time by simply consulting the table. Even if the numbers are larger, the table will speed up the computation of their GCD. By Euclid algorithm, the problem of finding GCD of two large numbers will eventually require the GCD of smaller numbers. The same principle can also be applied to more advanced algorithms for determining the GCD of two numbers. Eisenbeis and Sogno call this test Fast GCD [4].

## 7 Discussion

In this section we evaluate various dependence testing methods. We discuss and compare the effectiveness and efficiency of tests and algorithms described in previous sections.

We have shown that simple tests are fast and, in some cases exact, for special cases they can be applied to. Goff et al have shown experimentally that most subscripts are indeed simple. They have applied Rice test to four groups of Fortran programs: RiCEPS (Rice Compiler Evaluation Program Suite), the Perfect and SPEC benchmark suites, and two math libraries, eispack and linpack [5]. They have analyzed the types of the subscripts encountered, and the number of times each test was applied.

Experiments show that simple cases prevail. First of all, most of the arrays in the programs are one or two dimensional, with three dimensional arrays appearing only in some applications. None of the applications in all benchmark sets contained arrays with more than four dimensions. Most subscripts occurring in array references were separable. Coupled subscripts account for 20 % of all subscripts, with many programs (30 %) containing only

separable subscripts. Most of the coupled subscripts were of size two. Only 19 out of 8449 coupled subscripts were of size three, all of them in a single program. Nonlinear subscripts accounted for 6 % of all subscripts [5 – Table1].

The structure of the subscripts encountered in a program, defines which tests will be applied. Over all reference pairs tested, most subscript pairs were tested using ZIV (59 %), and strong SIV test (37 %). Only 3 % of all subscripts were MIV, and required usage of expensive tests. Of all successful tests, the ZIV and strong SIV test combined account for 86 %. The test is considered to be successful, if it proves independence, or eliminates at least one direction vector. Of all the subscript pairs proven independent, the ZIV test accounted for 94 %. However, in 23 % of all successful tests, symbolic constant manipulation was required. This shows that the ability of the test to perform symbolic analysis is important in dependence testing [5 – Tables 2 and 3]. Maydan et al have also shown that the vast majority of subscripts can be handled with the Single variable per constraint test, one of the simplest tests in their suite [9 – Table 1].

Different dependence testing algorithms use various combinations of simple tests. Maydan et al [9] suggest that the choice of individual tests is not very important. The key concept is the use of special case exact tests. When considering the applicability of various tests to particular subscript, only one of the tests will be applied. Otherwise, if in-exact test is used, and it returns dependence, it is unknown if the dependence really exists. Other tests need to be applied, since they might still be able to prove independence.

The ordering of the individual tests should be by their cost. If more than one test is applicable, the least expensive test will be used. Although they suggest usage of exact tests, Maydan et al [9] still use in-exact version of Fourier-Motzkin test for subscripts that can not be resolved by other tests.

Pugh [12] describes the Omega test as an integer programming algorithm with polynomial complexity for most common cases. However, more careful examination shows that Omega test also considers many special cases, which in turn results into reduced complexity. Constant test is applied while building the dependence testing problem. GCD test is applied during constraint normalization, and the test ends if independence can be proven. Inequality constraints are tightened, which also simplifies the problem. Before applying Fourier-Motzkin variable elimination, the system is checked for contradicting constraints, and tight and

redundant inequalities. Therefore, simple cases are in fact handled separately.

Pugh shows how the Omega test relates to other tests with respect to their performance. If the Single variable per constraint test can be applied to the constraint set, the Omega test can stop after the check for the contradicting constraints. If the system is independent, check for the contradicting constraints will result in contradiction. Otherwise, there is a dependence [12].

However, the equality constraint elimination in the Omega test requires  $O(enm \log |C|)$  worst-case time, where  $e$  is the number of equality constraints,  $m$  is the total number of constraints,  $n$  is the number of variables, and  $C$  is the coefficient with the largest absolute value in the constraint (see section 4.2). As opposed to that, even the trivial equality elimination method suggested for the Single variable per constraint test, by turning each equality into two inequalities, will produce  $m + e$  constraints in  $O(e)$  time. Each of the constraints needs to be traversed only once to check its lower and upper bound. Therefore, the total running time is  $O(e + m + e) = O(2e + m)$  worst-case time.

The Omega test will test subscripts that can be tested with the Acyclic test, and Simple loop residue test in the worst-case polynomial time, comparable to the times required for these tests [12]. For the subscripts that can be handled by the Delta test without using MIV tests, the Omega test will perform in polynomial time, without the need to perform Fourier-Motzkin variable elimination. The Omega test treats the dependence analysis problem as a single integer programming problem. Therefore, it automatically achieves the propagation effects of the Delta test [12]. The main advantage of the Omega test over other algorithms is that it is always exact for linear subscript expressions. Although it results in increased complexity, common cases are solved in polynomial time.

The Omega test was tested on a limited set of five programs of the NASA NAS benchmark suite [12]. Measurements show that the average time required for the Omega test to determine the direction vectors for a single array pair is less than  $500 \mu\text{s}$  on a 12 MIPS workstation. Interesting observation is the distribution of the running time. Pugh found that, for many array pairs, the cost of scanning array subscripts and loop bounds, and building the constraints, was nearly as high, or even higher than the cost of the actual test. This suggests that even the algorithm that is significantly faster than the Omega test would not lead to significant overall performance improvement [12]. This may mean that the performance

improvements of dependence testing algorithms could possibly be achieved by devising new algorithms that use simpler data structures to represent constraints.

Pugh also claims that, although memoization could be added to the Omega test, it is not profitable. According to his estimate, the cost of computing a hash key and verifying the cache hit would be approximately equal to the cost of scanning array subscripts and loop bounds and building the constraints. Therefore, memoization would not produce significant savings for typical cases [12]. This argument is not quite clear, since neither Pugh, nor Maydan provide experimental results showing the actual running time. Maydan provides experimental results only in terms of number of unique cases that occur. No data is given that would show the comparison of test performance with, and without memoization. Since many subscript expressions are simple, it is possible that retrieving the stored result is no faster than performing the actual test.

We have shown that simple subscript expressions prevail in practice. However, more complicated tests are still needed. If a loop contains many simple subscript expressions, and only one complex, not being able to resolve the complex expression prevents the parallelization. Most of the papers report the percentage of successful tests, which is not precise measure for the actual parallelism gained from the dependence analysis. Ideally, a standard model that could measure the parallelism found would be required to compare the performance of various dependence testing algorithms [9]. Of the tests described, the Omega test will yield the maximum parallelism, since it is the only algorithm that is always exact for linear subscript expressions.

The measurement results provided in papers discussed do not provide enough comparable experimental results to make a general conclusion as to which test performs the best. However, it is safe to say that the divide and conquer approach is most important aspect of dependence testing. Although the problem is generally very complex, considering the sufficient number of special cases separately, amortizes the cost of expensive tests, since they are applied rarely. However, expensive tests are necessary, since in many cases, all subscripts need to be tested exactly to prove independence.

## 8 Conclusion

Dependence testing plays significant role in parallelizing compilers. A set of distance and/or direction vectors describing dependences between loop iterations are necessary to discover the parallelism in a program. General techniques, based on an Integer Linear Programming are not suitable for dependence testing because of their complexity.

In this paper we have presented various techniques used by dependence testing algorithms. We have described details of two complex data dependence algorithms, the Omega test and Rice test.

Most data dependence testing algorithms use a kind of divide and conquer approach. Algorithms rely on the fact that many cases occurring in practice are simple, and can be solved using simple tests. Simple cases are handled first, using inexpensive techniques. If some data dependence problems cannot be solved by simple techniques, more complex algorithms are employed. Complex algorithms are necessary to produce exact results.

The Omega test is based on an extension of Fourier-Motzkin variable elimination to integer programming. Although its complexity is exponential in the worst case, most common subscript expressions are resolved in polynomial time. The Omega test is always exact for linear subscript expressions.

Rice test is partition based algorithm. Subscript expressions are classified into partitions. Each partition is tested separately, using the simplest test applicable, and the results are merged. Rice test has to apply in-exact tests for some subscripts. Therefore, the Rice test is not always exact.

Memoization can be used to reduce the number of times the test is applied. Memoization is applicable to any dependence testing algorithm. However, memoization does not necessarily result in performance improvement. Since many tests are simple, applying the test may be as fast as retrieving the stored result.

For a definite answer as to which of the algorithms performs the best in practice, thorough analysis including experimental methodology, which is beyond the scope of this paper, is required. If the exactness is more important than the test performance, the Omega test should be used, since it is the only dependence testing algorithm that is always exact for linear subscript expressions.

## References

- [1] D. Bacon, S. Graham and O. Sharp. Compiler Transformations for High-Performance Computing. *ACM Computing Surveys*, vol. 26, no. 4, pages 345-420, 1994.
- [2] U. Banerjee. *Dependence Analysis for Supercomputing*. Kluwer Academic Publishers, 1988.
- [3] M. Burke, R. Cytron. Interprocedural Dependence Analysis and Parallelization. In *Proceedings of the SIGPLAN Symposium on Compiler Construction*, pages 162-175, 1986.
- [4] C. Eisenbeis, J-C. Sogno. A General Algorithm for Data Dependence Analysis. In *Proceedings of the International Conference on Supercomputing*, pages 292-302, 1992.
- [5] G. Goff, K. Kennedy, and C-W. Tseng. Practical Dependence Testing. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 15-29, 1991.
- [6] J. Hennessy, D. Patterson. *Computer Architecture, Quantitative approach*, Third Edition. Morgan Kaufmann Publishers, 2003.
- [7] K. Kennedy and K. McKinley. Optimizing for Parallelism and Data Locality. In *Proceedings of the International Conference on Supercomputing*, pages 323-334, 1992.
- [8] Z. Li, P. Yew and C. Zhu. An Efficient Data Dependence Analysis for Parallelizing Compilers. *IEEE Transactions on Parallel and Distributed Systems*, vol. 1 no. 1, pages 26-34, 1990.
- [9] D. Maydan, J. Hennessy and M. Lam. Efficient and Exact Data Dependence Analysis. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 1-14, 1991.
- [10] S. Muchnik. *Advanced Compiler Design and Implementation*. Morgan Kaufmann Publishers, 1997.
- [11] P. Petersen. *Evaluation of Programs and Parallelizing Compilers Using Dynamic Analysis Techniques*. Doctoral Thesis, University of Illinois at Urbana-Champaign, 1993.
- [12] W. Pugh. A Practical Algorithm for Exact Array Dependence Analysis. *Communications of the ACM*, vol. 35, no. 8, pages 102-114, 1992.

[13] M. Wolfe and C-W. Tseng. The Power Test for Data Dependence. IEEE Transactions on Parallel and Distributed Systems, vol. 3, no. 5, pages 591-601, 1992.