Geometric Techniques for Parallelizing and Scheduling *Do-loops*

Tang Dongxing
Gopal Gupta
Laboratory for Logic, Databases, and Advanced Programming
Department of Computer Science
New Mexico State University
Las Cruces, NM 88003-0001
{dtang,gupta}@nmsu.edu

Abstract

To parallelize a *DO-loop* in a Fortran program, the independence of various iterations of the *DO-loop* has to be detected. Two commonly used tests to check for iteration independence are the *GCD test* and the *Banerjee test*. In this paper we introduce a new data dependence test called the *Diagonal test* that is based on some very simple geometric properties of straight lines and hyper-planes. We show that the Diagonal test is equivalent to Banerjee test. However, the Diagonal test is more intuitive, especially for singly-nested loops. The Diagonal test also inspires a new approach to parallelizing *DO-loops* by repeatedly partitioning a loop into smaller loops, and scheduling these smaller loops for parallel execution based on the data-dependence analysis.

Keywords: Parallelizing Compilers, Data-dependence Analysis, Diophantine Equation, Euclidean Geometry, Scientific Computing.

1. Introduction

Automatic parallelization of Fortran programs for parallel execution on multiprocessors has been an active topic of research in the last decade [1, 5, 4, 20]. It is well known that a large part of most scientific programs consists of operations over matrices using *DO-loops*. Hence parallelizing *DO-loops* will extract a significant amount of parallelism from these programs. Research on parallelizing *DO-loops* in Fortran programs has focused on two issues:

- **Dependence Analyses and Tests:** involves checking if the different iterations of a *DO-loop* are *independent*. If they are, then they can be executed in parallel.
- **Program Transformation and Scheduling:** If the iterations of a loop are not independent, then the loop is transformed (without changing its meaning) so that two or more iterations may become independent. Once two iterations have been determined to be *independent* (after perhaps some suitable transformations), these iterations must be scheduled for parallel execution on a multiprocessor in a way that maximum speed-up is obtained.

Whether a loop may be parallelized (or vectorized) depends primarily on whether two array references access the same location in the memory [2, 4, 16, 14, 20]. That is, whether two references to the same array from different iterations of a loop refer to the same location of that array. When the referenced array is m-dimensional and the array subscripts are linear[1], this problem is easily reduced to that of determining whether a simultaneous system of m linear equations, each of the form:

$$a_1 i_1 + a_2 i_2 + \cdots + a_n i_n = c \tag{1.1}$$

has an *integer* solution which satisfies *constraints* of the form:

$$I_{k0} \leq i_k \leq I_{k1}, \quad 1 \leq k \leq n. \tag{1.2}$$

Each i_k is either a loop index variable or some other variable. In this paper, we assume that i_k is a loop index variable and also that the lower-bound I_{k0} and the upper-bound I_{k1} of each index variable are constant.

The GCD test and the Banerjee test [4, 2] are two frequently used tests to determine whether the single linear equation (1.1) has a solution. When the system of equations under consideration contains more than one equation, it is common practice to test the equations individually (*subscript-by-subscript*) rather than testing for the existence of a simultaneous solution.

Regarding scheduling, once the nature of dependence of iterations in the loop has been determined, the various iterations that are independent can be executed in parallel, while those that are dependent have to be executed sequentially. Scheduling of iterations has also received a lot of attention [20, 10, 7, 8, 13].

In this paper we introduce a new data dependence test called the *Diagonal test* that is based on some very simple geometric properties of straight lines and hyper-planes. We show that the diagonal test is equivalent to the Banerjee test. However, the diagonal test is more intuitive, especially for singly-nested loops. Our diagonal test also inspires a new approach to parallelizing DO-loops by repeatedly partitioning a loop into smaller loops, and scheduling these smaller loops for parallel execution based on the data-dependence analysis. Our parallelizing technique is very general and is also applicable when the dependence equation is non-linear [19].

The rest of the paper is organized as follows: Section 2 reviews the GCD test and the Banerjee test while Section 3 introduces our new test called the *Diagonal test.* Section 4 introduces a new technique for parallelizing and scheduling a single *DO-loop* that is inspired by our Diagonal test. Section 5 presents related work (Banjerjee's Index-set Splitting Transformation) while Section 6 presents our conclusions.

2. GCD Test and Banerjee Test

The GCD test [3] and the Banerjee test [3] are two standard tests used to determine whether an n-variable linear equation of the form (1.1) has any solution constrained by (1.2). The GCD test is based on a theorem of elementary number theory that says that the linear equation (1.1) has

[1] In this paper we restrict ourselves to linear array subscripts.

an integer solution iff $\gcd(a_1, a_2, \cdots, a_n)$ divides the constant c. The test consists of checking for the indicated divisibility. The main disadvantage of the test is that it entirely ignores the variable constraints in (1.2).

On the other hand, the Banerjee test is based on a result of the theory of Real Analysis that says that any continuous function in a connected region must achieve every value that lies between the maximum and minimum values of the function in that region (the Intermediate Value Theorem). Therefore, the Banerjee test first computes the maximum value and the minimum value of the left hand side in (1.1), namely, the function $g(x_1, x_2, \cdots, x_n) = a_1x_1 + a_2x_2 + \cdots + a_nx_n$, in the region formed by (1.2) as follows:

$$min = \sum_{i=1}^{n}(a_i^+ I_{i0} - a_i^- I_{i1}) \quad \text{and} \quad max = \sum_{i=1}^{n}(a_i^- I_{i1} - a_i^- I_{i0}) \tag{2.1}$$

where $a^+ = a$ if $a \geq 0$ and 0 otherwise; $a^- = -a$ if $a \leq 0$ and 0 otherwise.

It then checks if the constant c lies between the computed minimum value and the maximum value. The answer is positive iff there is a solution to (1.1) satisfying the constraints in (1.2). However, the Banerjee test will succeed even if there is a real solution (but no integer solution) within the constraints (1.2). Therefore, we can see that the GCD test and the Banerjee test are complementary to each other.

3. Diagonal Test

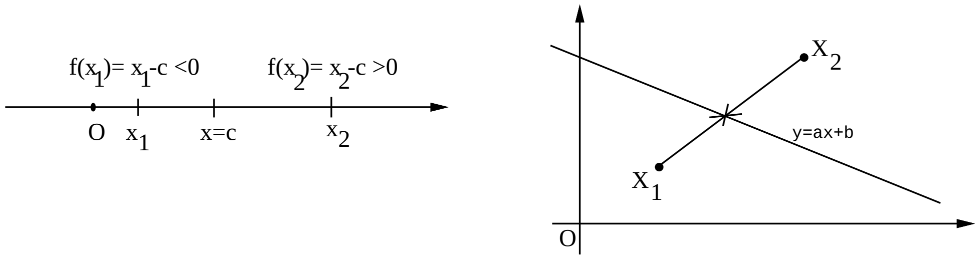
In this section we introduce a new method based on some very simple results of Euclidian Geometry for testing data dependency.

Consider a point $x = c$ in one dimensional space. Let $f(x) = x - c$, then for any point x_0, $f(x_0) \geq 0$ iff $x_0 \geq c$ and $f(x_0) \leq 0$ iff $x_0 \leq c$. Thus, for any two different points x_1 and x_2, it follows that $f(x_1) \times f(x_2) < 0$ iff x_1 and x_2 lie on different sides of the point $x = c$ (see Figure 1.(i)).

In the two dimensional space (i.e. a plane), similarly, consider a line $y = ax + b$. Let $f(x, y) = y - ax - b$. Consider two different points $X_1 = (x_{11}, x_{12})$ and $X_2 = (x_{21}, x_{22})$, then $f(X_1) \times f(X_2) \leq 0$, namely, $f(X_1)$ and $f(X_2)$ have different signs or one of them is zero, iff the line $y = ax + b$ intersects with the line segment between X_1 and X_2 (see Figure 1.(ii)). Where, the fact that $f(X_i)$ for some i is zero means that the line passes through the point X_i.

Note that a constraint of the form (1.2) in 2-dimensional space (i.e. $n = 2$ in constraint (1.2)), can be represented as a rectangle. The property mentioned above is made use of in determining the relative placement of the line represented by (1.1) with respect to this rectangle. The following text generalizes the above property and determination of the placement of a hyperplane w.r.t. a hypercuboid in n-dimensional space.

Consider a higher dimensional space. A rectangle in two dimensions generalizes to a hypercuboid in an n-dimensional space. Such a hypercuboid in n-dimensional space will have the same number of vertices as the n-dimensional hypercube (i.e. 2^n). Let a hypercuboid be denoted by $[I_{10} : I_{11},\ I_{20} : I_{21},\ \cdots,\ I_{n0} : I_{n1}]$, namely, the area represented by the constraints $I_{i0} < x_i < I_{i1},\ i = 1, 2, \cdots, n$, then the coordinate of a vertex is $(I_{1i_1}, I_{2i_2}, \cdots, I_{ni_n})$, where i_j is either 0 or 1.



(I) (II)

Figure 1: Relative Position of a Point in 1-D and 2-D Space

In the following, we first define the opposite vertex for a given vertex of a hypercuboid and the *diagonal* that joins these two opposite vertices. We then describe our Diagonal test and show that it is equivalent to the Banerjee test.

Definition 3.1. Opposite Vertices: Let I^n denote an n-dimensional hypercuboid $[I_{11} : I_{12},\ I_{21} : I_{22},\ \cdots,\ I_{n1} : I_{n2}]$, then $(I_{1\overline{i_1}}, I_{2\overline{i_2}}, \cdots, I_{n\overline{i_n}})$ denotes the vertex opposite to the vertex $(I_{1i_1}, I_{2i_2}, \cdots, I_{ni_n})$, where $\overline{j}$ means the negation of j.

Definition 3.2. Diagonal: Consider an n-dimensional hypercuboid $I^n = [I_{10} : I_{11},\ I_{20} : I_{21},\ \cdots,\ I_{n0} : I_{n1}]$. The line segment between a vertex and its opposite vertex is defined as a *diagonal* of the hypercuboid I^n.

It is not difficult to check that the above definitions for the n-dimensional hypercuboid are generalizations of the corresponding terminologies for one and two-dimensional rectangles.

Lemma 3.1: Let $f(x_1, x_2, \cdots, x_n) = a_1x_1 + a_2x_2 + \cdots + a_nx_n - c$, $X_1 = (x_{11}, x_{12}, \cdots, x_{1n})$ and $X_2 = (x_{21}, x_{22}, \cdots, x_{2n})$, and α and β be two scalars, then $f(\alpha X_1 + \beta X_2) = \alpha f(X_1) + \beta f(X_2) + (\alpha + \beta - 1)c$.

Proof: Since X_1 and X_2 are two vectors, hence,

$$\begin{aligned}\alpha X_1 + \beta X_2 &= (\alpha x_{11}, \alpha x_{12}, \cdots, \alpha x_{1n}) + (\beta x_{21}, \beta x_{22}, \cdots, \beta x_{2n})\\ &= (\alpha x_{11} + \beta x_{21}, \alpha x_{12} + \beta x_{22}, \cdots, \alpha x_{1n} + \beta x_{2n}).\end{aligned}$$

Thus, we have the following:

$$\begin{aligned}f(\alpha X_1 + \beta X_2) &= a_1(\alpha x_{11} + \beta x_{21}) + a_2(\alpha x_{12} + \beta x_{22}) + \cdots + a_n(\alpha x_{1n} + \beta x_{2n}) - c\\ &= a_1\alpha x_{11} + a_2\alpha x_{12} + \cdots + a_n\alpha x_{1n} + a_1\beta x_{21} + a_2\beta x_{22} + \cdots + a_n\beta a_n x_{2n} - c\\ &= \alpha(a_1x_{11} + a_2x_{12} + \cdots + a_nx_{1n}) + \beta(a_1x_{21} + a_2x_{22} + \cdots + a_nx_{2n}) - c\\ &= \alpha(a_1x_{11} + a_2x_{12} + \cdots + a_nx_{1n} - c + c) + \beta(a_1x_{21} + a_2x_{22} + \cdots + a_nx_{2n} - c + c) - c\\ &= \alpha f(X_1) + \beta f(X_2) + (\alpha + \beta - 1)c. \ \Box\end{aligned}$$

Lemma 3.2: Let $f(x_1, x_2, \cdots, x_n) = a_1x_1 + a_2x_2 + \cdots + a_nx_n - c$. Let $X_1 = (x_{11}, x_{12}, \cdots, x_{1n})$ and $X_2 = (x_{21}, x_{22}, \cdots, x_{2n})$ be two points in n-dimensional space. If $f(X_1) \leq 0$ and $f(X_2) \geq 0$, then there is a point in the line segment between X_1 and X_2 at which the hyperplane $a_1x_1 + a_2x_2 + \cdots + a_nx_n = c$ intersects with the line segment between X_1 and X_2.

Proof: If $f(X_i) = 0$ for some i, then we know that the plane passes through the point X_i and we are done. Thus, in the remaining part of the proof, we assume that $f(X_i) \neq 0$ for $i = 1$ and 2.

To prove the lemma, it suffices to prove that there is a point $X = X_1 + (X_2 - X_1)\alpha$ with $0 < \alpha < 1$ on the line segment joining X_1 and X_2 such that $f(X) = 0$, i.e., the point X lies on the hyperplane $a_1x_1 + a_2x_2 + \cdots + a_nx_n = c$. Since $X = X_1 + (X_2 - X_1)\alpha$, that is, $X = (1-\alpha)X_1 + \alpha X_2$, thus, by lemma 3.1, we have the following.

$$\begin{aligned} f(X) &= f((1-\alpha)X_1 + \alpha X_2) \\ &= (1-\alpha)f(X_1) + \alpha f(X_2) + (1 - \alpha + \alpha - 1) \\ &= f(X_1) - \alpha(f(X_2) - f(X_1)). \end{aligned}$$

Let $\alpha = -f(X_1)/(f(X_2) - f(X_1))$. Since $f(X_1) < 0$ and $f(X_2) > 0$, thus, $(f(X_2) - f(X_1)) > 0$ and $-f(X_1) > 0$. Hence, we have that $\alpha > 0$. If we rewrite α, then we have that $\alpha = (-f(X_2) + f(X_2) - f(X_1))/(f(X_2) - f(X_1)) = -f(X_2)/(f(X_2) - f(X_1)) + 1$. Since $f(X_2) > 0$ and $(f(X_2) - f(X_1)) > 0$, thus, $-f(X_2)/(f(X_2) - f(X_1)) < 0$, hence, we have that $\alpha < 1$. This implies that X is the point where the hyperplane $a_1x_1 + a_2x_2 + \cdots + a_nx_n = c$ intersects with the line segment between X_1 and X_2. $\square$

¿From lemma 3.2, we have the following result.

Theorem 3.3: Let $f(x_1, x_2, \cdots, x_n) = a_1x_1 + a_2x_2 + \cdots + a_nx_n - c$, and I^n be a hypercuboid in n-dimensional space. If there is a diagonal of the hypercuboid I^n such that the function $f(X)$ on the two corresponding vertices has different signs, or one of them is zero, then the hyperplane $a_1x_1 + a_2x_2 + \cdots + a_nx_n = c$ intersects with the hypercuboid, namely, there is a point in the hypercuboid I^n that also lies in the hyperplane $a_1x_1 + a_2x_2 + \cdots + a_nx_n = c$.

This theorem shows us that if we can find a diagonal such that the function at the two vertices of this diagonal has values with different signs or one of them is zero, then we claim that linear equation (1.1) has some solution satisfying the constraints in (1.2). We call this test as the *Diagonal test*.

In theorem 3.3, we can see that the condition, namely, the function $f(X)$ on the two corresponding vertices has different signs, or one of them is zero, is equivalent to the condition that one of values $g(x_1, x_2, \cdots, x_n) = a_1x_1 + a_2x_2 + \cdots + a_nx_n$ on the two corresponding vertices of a diagonal is not greater than the constant c and the other is not smaller than the constant c. This is because the function values on the two vertices of the diagonal must be between the maximum value and the minimum value of the function g. This implies that if the Diagonal test is positive (theorem 3.3), then the Banerjee test must be positive. That is, the Diagonal test in theorem 3.3 guarantees the Banerjee test.

Next, let us consider the other direction of theorem 3.3, i.e., if there is a solution for $f(X) = 0$ in the hypercuboid I^n, then there is a diagonal with vertices X_1 and X_2 such that $f(X_1)$ and $f(X_2)$ have opposite signs or one of them is zero. We first show the following lemma:

Lemma 3.4: Let the function $f(x_1, x_2, \cdots, x_n) = a_1x_1 + a_2x_2 + \cdots + a_nx_n - c$ with all a_i's being non-zero, the hypercuboid I^n be $[I_{10} : I_{11},\ I_{20} : I_{21},\ \cdots,\ I_{n0} : I_{n1}]$. Then, for any point $X = (x_1, x_2, \cdots, x_n)$ in the hypercuboid I^n, we have the following inequality.

$$a_1 I_{1i_1} + a_2 I_{2i_2} + \cdots + a_n I_{ni_n} - c \leq$$
$$f(x_1, x_2, \cdots, x_n) = a_1x_1 + a_2x_2 + \cdots + a_nx_n - c \leq$$
$$a_1 I_{1\overline{i_1}} + a_2 I_{2\overline{i_2}} + \cdots + a_n I_{n\overline{i_n}} - c \qquad (3.4)$$

where i_k is either 0 or 1 and $\overline{j}$ means the negation of j.

Proof: We prove this result by induction on n.
Base Case: for $n = 1$, there are two cases.
Case 1: $a_1 > 0$. In this case, for $I_{10} \leq x \leq I_{11}$, we have that $a_1I_{10} - c \leq f(x) \leq a_1I_{11} - c$. we can rewrite it as $a_1I_{10} - c \leq f(x) \leq a_1I_{1\overline{0}} - c$. Hence, the inequality (3.4) holds.
Case 2: $a_1 < 0$. In this case, for $I_{10} \leq x \leq I_{11}$, we have that $a_1I_{11} - c \leq f(x) \leq a_1I_{10} - c$. we can rewrite it as $a_1I_{11} - c \leq f(x) \leq a_1I_{1\overline{1}} - c$. Hence, the inequality (3.4) holds.
Induction Hypothesis: for $n >= 1$, assuming that the inequality (3.4) holds.
Induction: let $k = n + 1$ and $X = (x_1, x_2, \cdots, x_k)$ be a point in the hypercuboid I^k, then for each i between 1 and k, we have that $I_{i0} \leq x_i \leq I_{i1}$. Hence, similar to the base case, we have that $a_1I_{1i_1} \leq a_1x_i \leq a_1I_{1\bar{i}_1}$. By the induction hypothesis, we have that

$$a_2I_{2i_2} + \cdots + a_nI_{ni_n} - c \leq a_2x_2 + \cdots + a_kx_k - c \leq a_2I_{2\overline{i_2}} + \cdots + a_kI_{k\overline{i_k}} - c.$$

Combining the above two facts yields (3.4) for $k = n + 1$. □

Note that $a_1I_{1i_1} + a_2I_{2i_2} + \cdots + a_nI_{ni_n} - c$ and $a_1I_{1\overline{i_1}} + a_2I_{2\overline{i_2}} + \cdots + a_nI_{n\overline{i_n}} - c$ in the formula (3.4), in fact, are the function values corresponding to the vertex $(I_{1i_1}, I_{2i_2}, \cdots, I_{ni_n})$ and $(I_{1\overline{i_1}}, I_{2\overline{i_2}}, \cdots, I_{n\overline{i_n}})$. This implies that for any point X in the hypercuboid I^n, there is a vertex such that the function value at X lies between the function value of the vertex and that of its opposite vertex. We rewrite this fact in the following lemma.

Lemma 3.5: Let $f(x_1, x_2, \cdots, x_n) = a_1x_1 + a_2x_2 + \cdots + a_nx_n - c$. For any point X in I^n, there must exist a vertex such that $f(X)$ lies between the function value of the vertex and that of its opposite vertex.

Following the above lemma, we can show the following result.

Theorem 3.6: Let $f(x_1, x_2, \cdots, x_n) = a_1x_1 + a_2x_2 + \cdots + a_nx_n - c$, and I^n be an n-dimensional hypercuboid, then, if the hyperplane $a_1x_1 + a_2x_2 + \cdots + a_nx_n = c$ intersects with the hypercuboid I^n, then, there is a diagonal of the hypercuboid I^n such that the function $f(X)$ at the two vertices corresponding to this diagonal has different signs, or is zero at one of these two vertices, that is, the hyperplane intersects some diagonal of the hypercuboid.

Proof: Assume that the hyperplane $a_1x_1 + a_2x_2 + \cdots + a_n = c$ does not intersect with any diagonal of the hypercuboid I^n, then, by lemma 3.2, for any vertex X_1 and its opposite one X_2, $f(X_1)$ and $f(X_2)$ must have the same sign. Moreover, for any point X in I^n, by lemma 3.5, there must exist a vertex such that $f(X)$ lies between the function value of the vertex and that of its opposite vertex. Since the function $f(X)$ has the same signs at each vertex and its opposite vertex, hence, we can see that for each point X in I^n, $f(X)$ must be either negative or positive. This implies that no point in I^n can satisfy the hyperplane. That is, this hyperplane does not intersect with the hypercuboid, which is a contradiction. Therefore, if the hyperplane intersects I^n, it also intersects at least one diagonal of the hypercuboid. □

This theorem tells us that if we try all diagonals (i.e., evaluate the hyperplane at end points of each diagonal), then we can surely test whether the hyperplane intersects the hypercuboid. Since a positive outcome from the Banerjee test means that the function $g(X)$ attains the value c in the region constrained by (1.2), this implies that $f(X) = 0$ for some X in that region. Thus, by the above theorem, we know that the diagonal test will definitely give a positive answer for some diagonal. If we fail the diagonal test for all diagonals, then the Banerjee test must also fail.

¿From theorem 3.3, theorem 3.6, and the above discussion, we have the following main result:

Theorem 3.7: Let $f(x_1, x_2, \cdots, x_n) = a_1x_1 + a_2x_2 + \cdots + a_nx_n - c$, and I^n be a hypercuboid in n-dimensional space, then, the hyperplane $a_1x_1 + a_2x_2 + \cdots + a_nx_n = c$ intersects with the hypercuboid I^n **iff** there is a diagonal of the hypercuboid I^n such that the function $f(X)$ on the two corresponding vertices has different signs or zero at one of these two vertices.

If the diagonal that the hyperplane intersects is known, then the diagonal test is equivalent to the Banerjee test. In such a case, one can see that the Diagonal test is more intuitive and easier to use than the Banerjee test in which we first have to compute both a^+ and a^- for each coefficient in (1.1) and then the *min* value and the *max* value in (2.1). However, we do not know for which diagonal the Diagonal test will succeed unless we exhaust all diagonals. Thus, in order to be sure that the equation (1.1) has a solution satisfying the constraints in (1.2), in the worst case we have to try the diagonal test for every diagonal in the hypercuboid.

Therefore, in order to use our diagonal test for determining whether there exists any solution of (1.1) satisfying (1.2), we have to have a systematic way of enumerating all the diagonals. In two dimensional space (i.e. a plane), since there are only two diagonals, it is easy to select all diagonals for the test. For n-variable linear equations, there is a simple way of enumerating all the diagonals. There is a 1-1 mapping from vertices of the n-dimensional hypercuboid to an n-dimensional hypercube. The vertices of an n-dimensional hypercube can be numbered using n bits such that two diagonally across vertices have complementary bit patterns. Thus, for our purposes we start with the n-bit *number* $00 \cdots 0$ as the bit pattern for the first vertex in the hypercube. Given a vertex in the hypercube, we compute the *negation* of the bit-pattern that corresponds to that vertex, which gives us the bit-pattern of the vertex diagonally across. With the bit patterns of two diagonally across vertices known, we can easily find the corresponding coordinates of these vertices in the hypercuboid $[I_{10} : I_{11},\ I_{20} : I_{21},\ \cdots,\ I_{n0} : I_{n1}]$. If the ith bit in the bit-pattern for the vertex is 0, then the ith coordinate is I_{i0}, if it is 1, the ith coordinate is I_{i1}. Thus, once the two diagonally opposite vertices are obtained, the diagonal test can be applied. If the answer is positive, the process is done. If not, then we increase the *number* by 1 and repeat the above procedure until *number* is 2^{n-1}. For example, in a 3-d hypercuboid ($n = 3$) $[I_{10} : I_{11},\ I_{20} : I_{21},\ I_{30} : I_{31}]$, we start with 000 which corresponds to the vertex (I_{10}, I_{20}, I_{30}), compute the bit pattern for its opposite vertex which is 111, which corresponds to the vertex (I_{11}, I_{21}, I_{31}). We then apply the Diagonal test for these vertices. If the test fails, we increment the number to 001, which corresponds to vertex (I_{10}, I_{20}, I_{31}). The bit-pattern of the vertex diagonally across is 110, which corresponds to the vertex (I_{11}, I_{21}, I_{30}). The Diagonal test can then be applied to these two vertices, and so on. In the worst case this test may have to be repeated for all 4 diagonals of the 3-d cuboid. In the general case, in the worst case we may have to repeat the Diagonal test 2^{n-1} times.

Let us now consider the following examples.

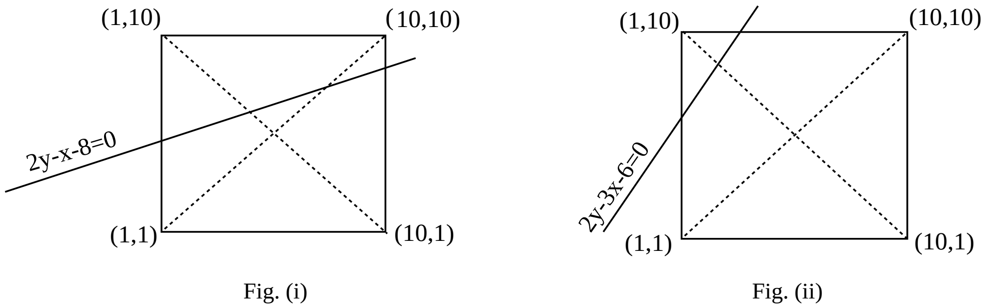


Figure 2: Geometrical Representation of Example 3.1 and 3.2

Example 3.1: Assume that we have the following *DO-loop*.

```
for i=1, 10
      ...
  a(2i) = ···
      ...
  ··· = ··· a(i+8) ···
     ...
```

We can see that the corresponding linear equation is $2y - x - 8 = 0$. Using our diagonal test, we start with the *number* 00. The negation of 00 is 11. Thus, we can see that the two vertices of the corresponding diagonal have the coordinates (1,1) and (10,10). We find that the function $f(x, y) = 2y - x - 8$ has values -7 and 2 at these two vertices. Since $-7 * 2 < 0$, by theorem 3.3 we conclude that the linear equation has (real) solution in the region $[1 : 10, 1 : 10]$. That is, there may exist data dependency between iterations. □

By looking at Figure 2.(i), one can tell that $2y - x - 8 = 0$ has a solution in the region of interest. This example shows that the diagonal test is more intuitive and easier to understand and use than the Banerjee test.

Example 3.2: Assume that we have the following *Do-loop*.

```
for i=1, 10
     ...
  a(2i) = ···
      ...
  ··· = ··· a(3i+6) ···
     ...
```

We can see that the corresponding linear equation is $2y - 3x - 6 = 0$. Using the diagonal test, we start with the *number* 00. The negation of 00 is 11. Thus, we can see that the two vertices of the corresponding diagonal have the coordinates (1,1) and (10,10). We find that the function $f(x, y) = -3x + 2y - 6$ has values -7 and -16 at these two vertices. Since they have the same sign, the Diagonal test fails this time. We next increase the *number* 00 to 01. It is easy to see that the negation of 01 is 10. Thus, the coordinates of the two corresponding vertices are (10,1) and (1,10), their corresponding values of $f(x, y)$ are -34 and 11. Since $-34 * 11 < 0$, by theorem 3.3 we know that the linear equation $-3x + 2y - 6 = 0$ has (real) solution in the region $[1:10,\ 1:10]$. □

In this example we had to try the Diagonal test on both the diagonals (although the line $2y-3x-6 = 0$ only intersects the second diagonal as shown in Figure 2.(ii)) because for the first diagonal we picked, the Diagonal test did not succeed. We claim that each test above gives us some intuitive feeling about the placement of the line w.r.t. the rectangle. This gives us an idea about where exactly the data dependencies lie, which can then be used for partitioning and scheduling different iterations of a loop on a multiprocessor. This is discussed next.

4. Parallelizing and Scheduling *Do-loops*

Shen, Li and Yew empirically studied a large suit of Fortran programs and found that nearly 60 percent of array operations involve only one dimensional arrays [17]. In the light of this observation, and for simplicity, in this section we will only consider singly-nested *Do-loop* of the following form:

```
for i=I1,I2                         or        for i=I1,I2
   ...                                            ...
  a(b1 i + c1) = ...                            ... = ... a(b1 i + c1) ...
   ...                                            ...
  ... = ... a(b2 i + c2) ...                    a(b2 i + c2) = ...
   ...                                            ...
```

Where I_1, I_2, b_1, b_2, c_1 and c_2 are constants and a is a one dimensional array. Assume that when c_1 is equal to c_2, b_1 is not equal to b_2. Our results can be generalized to nested *DO-loops* with nesting of more than 1, but we restrict ourselves in this paper to singly-nested *DO-loops*. Our results also apply to the case where the dependence equation is non-linear [19], however, for the rest of this paper we will restrict ourselves to linear dependence equation. Treatment of non-linear dependence can be found elsewhere [19].

For parallelizing a *Do-loop* we need to consider whether there exist two different iterations that access the same location in the array a, namely, we have to check if the following linear diophantine equation has an (integer) solution.

$$b_1x_1 + c_1 = b_2x_2 + c_2$$

In other words, we have to consider whether an equation of the form:

$$a_1x_1 + a_2x_2 = c \tag{4.1}$$

has any (integer) solution (x_2, x_1) in the rectangular region $[I_1 : I_2,\ I_1 : I_2]$.

To test whether (4.1) has any solution, we can use any kind of test such as the diagonal test introduced in section 3, the GCD test, or the Banerjee test. Obviously, if there is no solution, we can trivially parallelize this single *DO-loop*. On the other hand, if there is any solution, we may use some of the techniques that have been developed, such as partitioning, recurrence breaking, cycle shrinking, loop spreading, subscript blocking, etc. (See Chapter 2 of [12], also [6] and [20]) for transforming this loop and then parallelizing it. Most of the program transformation techniques are, however, applicable only when the *dependence distance* [4] between two iteration varies in a regular, well-defined manner. In the following we develop a loop-partitioning technique that allows us to exploit parallelism in loops with any kind of data-dependencies (i.e. the Banerjee test succeeds on them). This loop-partitioning technique is inspired by our Diagonal test, i.e., the geometric relationship between the line corresponding to equation (4.1) and the rectangle $[I_1 : I_2,\ I_1 : I_2]$.

The main idea is that we partition a single loop with loop limits I_1 and I_2 into two single loops, the first one having loop limits I_1 and $\lfloor \frac{I_1+I_2}{2} \rfloor$ and the second loop having loop limits $\lfloor \frac{I_1+I_2}{2} \rfloor + 1$ and I_2. There are a number of possbilities regarding execution of the partitioned loops that are explained next. One can continue the partitioning and partition the two half-loops further until a certain grain size is reached going below which parallel execution will not pay.

Intuitively, as shown in Figure 3, R_1 is the region $[I_1 : \lfloor \frac{I_1+I_2}{2} \rfloor,\ I_1 : \lfloor \frac{I_1+I_2}{2} \rfloor]$, R_3 is the region $[\lfloor \frac{I_1+I_2}{2} \rfloor + 1 : I_2,\ \lfloor \frac{I_1+I_2}{2} \rfloor + 1 : I_2]$, R_2 is the region $[\lfloor \frac{I_1+I_2}{2} \rfloor + 1 : I_2,\ I_1 : \lfloor \frac{I_1+I_2}{2} \rfloor]$, and R_4 is the region, $[I_1 : \lfloor \frac{I_1+I_2}{2} \rfloor,\ \lfloor \frac{I_1+I_2}{2} \rfloor + 1 : I_2]$. We can see that the regions R_1 and R_3 correspond to the first and the second partitioned loops, the first one having the loop limits I_1 and $\lfloor \frac{I_1+I_2}{2} \rfloor$, the second one having the loop limits $\lfloor \frac{I_1+I_2}{2} \rfloor + 1$ and I_2.

Depending on the relative positions of the line with respect to regions R_1 through R_4, the following conclusions can be drawn:

- If the line crosses region R_1, then there may be a dependency between two iterations of the first partition of the loop and the various iterations of this partition cannot be executed in parallel, otherwise they can be executed in parallel.
- If the line crosses region R_2 or R_4, then there may be a dependency between an iteration of the first partition and an iteration of the the second partition of the loop. This means that the two partitions cannot be executed in parallel with each other (i.e. the second partition should be executed only after the first one has finished), otherwise they can be executed in parallel with each other.
- If the line crosses region R_3, then there may be a dependency between two iterations of the second partition of the loop and the various iterations of this partition cannot be executed in parallel, otherwise they can be executed in parallel.

Note that if the line does not cross any of the region R_1 through R_4 then there is no dependency between any two iterations of the loop.

The line may pass through more than one regions R_1 through R_4, in which case the conditions for different individual regions have to be combined. This is shown in Figure 3.

In Figures 3.(I) and 3.(II), the line does not pass through any of the regions R_1 through R_4, hence all the iterations of the loop can be executed in parallel. If the line crosses the region R_4 or R_2 (but not R_1 or R_3) as shown in Figures 3.(III), 3.(IV), and 3.(V), then the iterations of each of the two partitions can be executed in parallel. However, the partitions have to be executed one after the other. That is, iterations of the first partition are executed in parallel first, followed by the iterations of the second partition executed in parallel next. If the line passes through only R_1 (R_3)

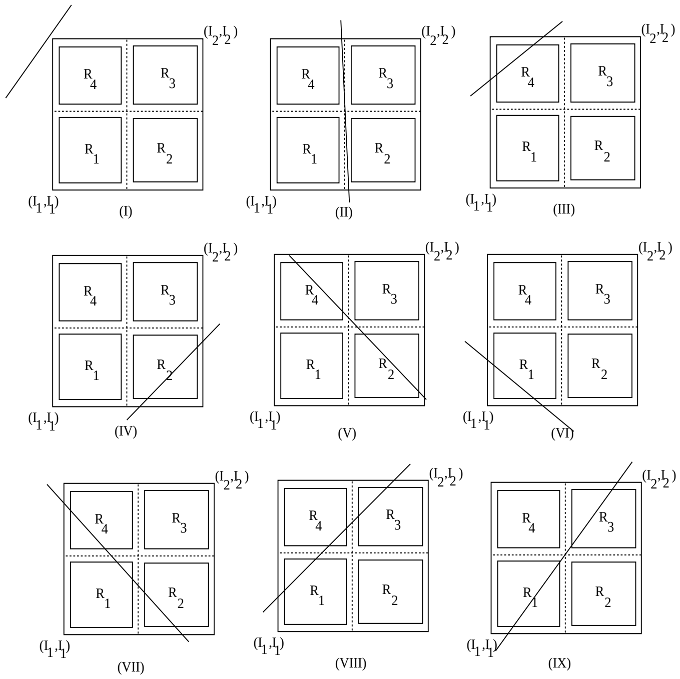


Figure 3: The Geometrical Relationship Between a Line and a Square

as shown in Figure 3.(VI), then iterations of the first (second) partition cannot be parallelized, however, those of the second (first) partition can be. Also, the first (second) partition, with its iterations executed sequentially, can be executed in parallel with the second (first) one, with its iterations executed in parallel. If the line passes through R_1 (R_3) and also through R_2 or R_4 as shown in Figure 3.(VII), then only the iterations in second (first) partition can be executed in parallel; those of the first (second) cannot be. Moreover, the second partition should be executed only after all iterations of the first partition have finished. If the line passes through R_1 and R_3 and one of R_2 or R_4 as shown in Figure 3.(VIII), then no parallelization is possible at all. If the line passes through R_1 and R_3 but not R_2 and not R_4, as shown in Figure 3.(IX), then the iteration in neither of the two partitions can be executed in parallel. However, the two partitions (with individual iterations of each executed sequentially) can be executed in parallel with each other.

As mentioned, each of the loop partitions can be partitioned themselves further and the analysis repeated. The loop-partitioning method is condensed in the algorithm that follows.

Algorithm *for Parallelizing a single Do-loop*
Input: a line $l:\ a_1x_1 + a_2x_2 = c$, an interval $[I_1, I_2]$, and a grain size *chunk*.
Output: the partitioned (parallelized) loops.

BEGIN

(1) **Test** whether the line l passes through the rectangle $[I_1 : I_2,\ I_1 : I_2]$ using the Banerjee test or the diagonal test. If the answer is no, then the loop can be fully parallelized. **Output** the parallelized loop using *doall* notation and exit. Otherwise, **goto** step 2.

(2) **Test** whether the line crosses the region R_1, if not, then the first half of the loop can be parallelized. Record this first parallelized loop using *doall* notation. Otherwise, it means that the first half of the loop can not be parallelized, if the length of the side of R_1 is not larger than *chunk*, then **output** the sequential loop using *do* notation, otherwise, call the parallelizing algorithm with input l, the interval from the left end to the right end of one side of the region R_1, and the grain size *chunk*.

(3) **Repeat** (2) for the region R_3.

(4) **Test** whether l crosses the region R_2 or R_4, if yes, **output** sequential code composed of the recorded code in (2) and then the recorded code in (3). If not, then **output** the parallel code composed of the recorded code in (2) and the recorded code in (3).

END

Note that before we first apply this algorithm, we can test whether there is any integer point in the line l using the GCD test. If the answer is negative, then we know that the original loop can be fully parallelized, thus we can directly output the fully parallelized code without applying the algorithm for partitioning. Moreover, for the sake of flexibility of scheduling on a multiprocessor, two adjacent partitions, both of whose iterations can be executed in parallel, can be combined into a single partition. Two adjacent partitions, both of whose iterations are to be executed sequentially, can be likewise combined into a single partition.

The output of the above algorithm can be represented as a binary tree. The left branch represents the code corresponding to the parallelized first partition of the loop, while the right branch represents the code corresponding to the parallelized second partition of the loop. Each non-leaf node is marked with either *seq* or *par* indicating whether the two partitions are to be executed in parallel with each other or sequentially. Each leaf-node represents a sequential loop with the notation *do* or parallel loop with notation *doall*. Once the *par-seq* tree has been output, it can be traversed bottom-up and adjacent partitions, if possible, combined. This combining can be carried out only if the two adjacent *do-all* loops have a least common ancestor node that is a *par* node, or two adjacent *do* loops that have a least common ancestor that is a *seq* node. Let us now consider the following examples and see how we can apply the above technique for parallelizing single *DO-loops*.

Example 4.1: Assume that we have the following *Do-loop*.

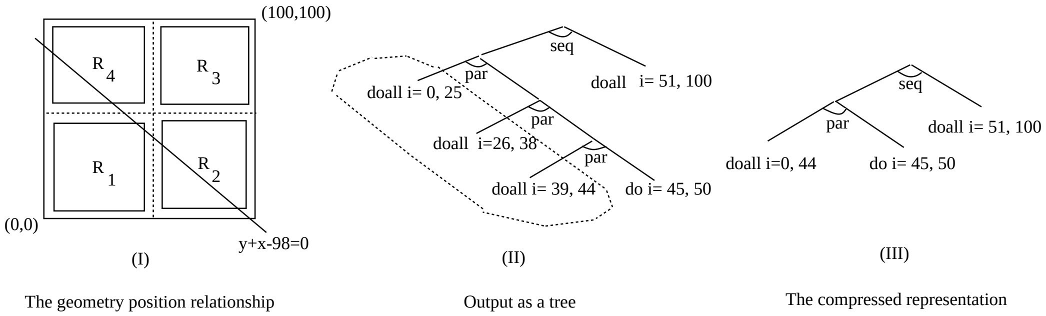


Figure 4: Geometrical Representation of Example 4.1

```
for i=0, 100
      ...
  a(i+2) = ···
     ...
  ··· = ··· a(100-i) ···
     ...
```

Let us apply the above algorithm with the grain size *chunk* 10 and check the parallelized results.

We first find that the corresponding line is $y + x - 98 = 0$ and the interval is $[0, 100]$. The position relationship is represented in Figure 4.(I).

In fact, it is easy to check that both GCD test and Banerjee test are positive. Moreover, the dependence distance is not constant. Thus, most techniques can not be successfully applied for this example. However, if we apply the above algorithm, we can find that we can successfully parallelize this single *Do-loop*. The parallelized code is represented in Figure 4.(II). After combining all adjacent loops in the dashed circle in (II), the final parallelized code is represented in Figure 4.(III). □

Example 4.2: Assume that we have the following *Do-loop*.

```
for i=0, 50
     ...
  a(3i+100) = ···
     ...
  ··· = ··· a(13i-20) ···
     ...
```

Let us apply the above parallelizing algorithm with the grain size *chunk* 10 and check the parallelized results.

We first find that the corresponding line is $3y - 13x + 120 = 0$ and the interval is $[0, 50]$. The position relationship is represented in Figure 5.(I).

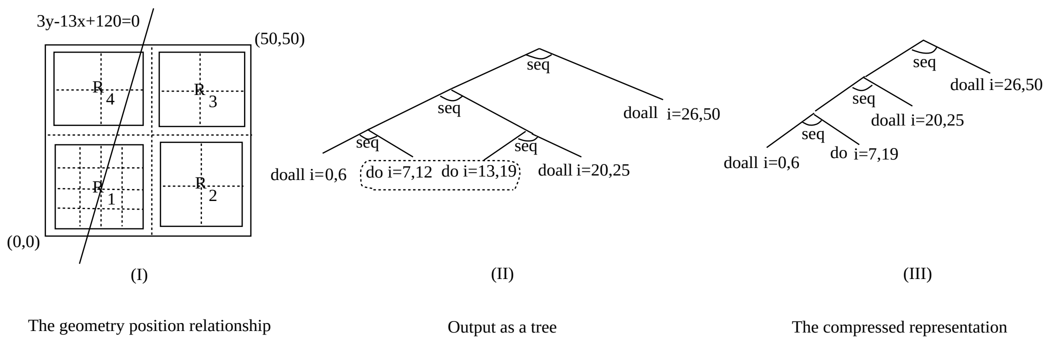


Figure 5: Geometrical Representation of Example 4.2

In fact, it is easy to check that both GCD test and Banerjee test are positive. Moreover, the dependence distance is not constant. Thus, most techniques cannot be successfully applied. However, if we apply the above parallelizing algorithm, we can find that we can successfully parallelize this single *Do-loop*.

The parallelized code is represented in Figure 5.(II), After combining all adjacent loops in the dashed circle in (II), the final parallelized code is represented in Figure 5.4.3 (III). □

Let's next consider an extreme situation in example 4.3 and to see how our parallelizing algorithm works.

Example 4.3: Assume that we have the following *Do-loop*.

```
for i=0, 10
    ...
  a(i+5) = ···
    ...
  ··· = ··· a(20i-100) ···
    ...
```

Let us apply the above parallelizing algorithm with the grain size *chunk* 1 and check the parallelized results.

We first find that the corresponding line is $20x - y - 105 = 0$ and the interval is $[0, 10]$. The position relationship is represented in Figure 6.(I).

In fact, it is easy to check that both GCD test and Banerjee test are positive. Moreover, the dependence distance is not constant. Thus, most techniques (e.g. those described in Chapter 2 of [12]) can not be successfully applied. However, if we apply the above parallelizing algorithm, we can find that we can successfully parallelize this single *DO-loop*.

The parallelized code is represented in Figure 6.(II). After combining all adjacent loops in the dashed circle in (II), the final parallelized code is represented in the figure 6.(III), that is the fully parallelized loop of the original single loop. □

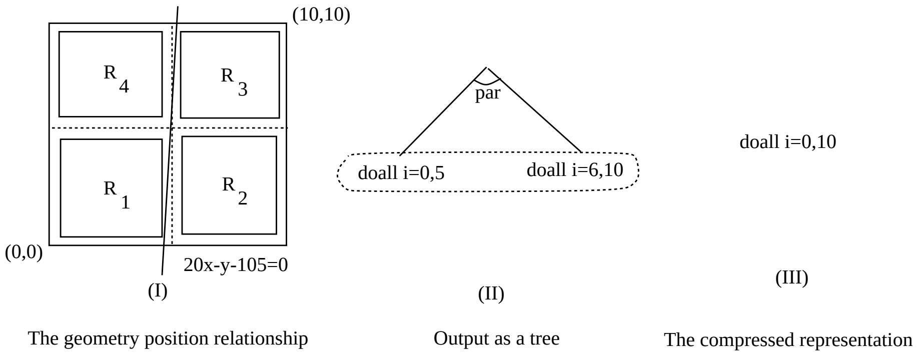


Figure 6: Geometrical Representation of Example 4.3

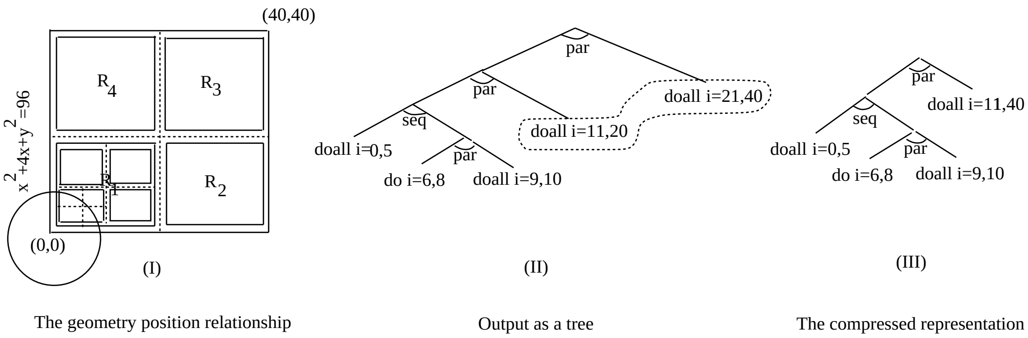


Figure 7: Geometrical Representation of Example 4.4

Finally, let's consider an application of our partitioning technique for nonlinear subscript cases.

Example 4.4 Assume that we have the following *Do-loop*.

```
for i=0, 40
     ...
  a(i*i+4i) = ···
     ...
  ··· = ··· a(96-i*i) ···
     ...
```

Let us apply our parallelizing algorithm assuming grain size *chunk* of 3 and check the parallelized results.

We first can find that the data dependency equation is $x^2+4x = 96-y^2$, that is, $(x+2)^2+y^2 = 100$, and the interval is [0, 40] as in the linear case. The position relationship is represented in Figure 7.(I).

For non-linear dependence our diagonal test is really more powerful than Banerjee's test which is specifically designed for cases where dependency is linear (if dependency is non-linear the use a^+

and a^- will not make any sense, and in fact will be incorrect). The *Do-loop* above will not be parallelized by most supercompliers due to the nonlinearity in data dependency. (However, it is also possible to extend Banerjee's test for non-linear cases.) Also it should be noted that in certain cases (when the dependency curve does not intersect any of the diagonals), the diagonal test will not work (the extended Banerjee test will not work either for such cases) [19].

Using our technique the section of the loop which causes this dependency is easily identified, and the independent parts of the loop parallelized as shown in Figure 7(I). The parallelized code is represented in Figure 7.(II). After combining all adjacent loops in the dashed circle in (II), the final parallelized code is represented in the Figure 7.(III). □
It should be noted that while the diagonal test has the same power as Banerjee test, it still has two advantages:

1. It works for non-linear dependencies as well, without any extension, unlike Banerjee's test.

2. It is much more intuitive for the most commonly occurring case of single subscript arrays. Although one might argue that this intuition is of little value because once the test is built into the compiler the user of the compiler doesn't have to know about the test. However, one could conceivably build a graphical tool that shows the data-dependency to the programmer in a way similar to the figures above. For those cases where the compiler cannot transform a loop in order to extract parallelism, this graphical tool can help the programmer visualize the data-dependency and modify or reprogram their loop in such a way that more parallelism can be exploited by the compiler. In other words, when the Banerjee test fails, and the compiler cannot figure out how to automatically transform the program so that more parallelism can be exploited, the compiler cannot give any information to the user as to what to do next (i.e., the compiler cannot tell the user in any intuitive way as to why the test failed). With the diagonal test, if no parallelism can be exploited, the compiler can put up a graphical representation (which is intuitively meaningful to the user) and seek his/her help in uncovering parallelism if it exists.

Finally, in this section we will show that the algorithm for parallelizing a single *DO-loop* needs at most linear time data dependency tests w.r.t. the loop index range (from I_1 to I_2).

Since in each iteration, we basically partition a square into four regions from R_1 to R_4. For both R_1 and R_3, we have to test whether a line passes through these regions. For R_2 or R_4, we just need to test whether the line crosses any of these two regions. Thus, after partitioning the square with the side length of $(I_2 - I_1)$ into four regions, we need at most four data dependency tests using the Banerjee test, or the diagonal test. Again, after partitioning two squares R_1 and R_3 whose side lengths are $(I_2 - I_1)/2$, we further need 2×4 tests. Repeating the above partitioning and test procedure until the side length is no greater than the grain size $chunk$, we totally need $4+2*4+2^2*4+\cdots+4*2^{k-1}$ data dependency tests, where $k = \lceil log_2 \frac{n}{chunk} \rceil$. Thus, the total number of tests equals the above summation, namely, $4(2^{log_2(\frac{n}{chunk})}) = \Theta(n/chunk)$ data-dependency tests. Moreover, if a test gives negative answer in step (1), (2) or (3), then we will not partition any more. Therefore, we can see that the parallelizing algorithm needs $O(n/chunk)$ tests. This fact can also be verified from examples above.

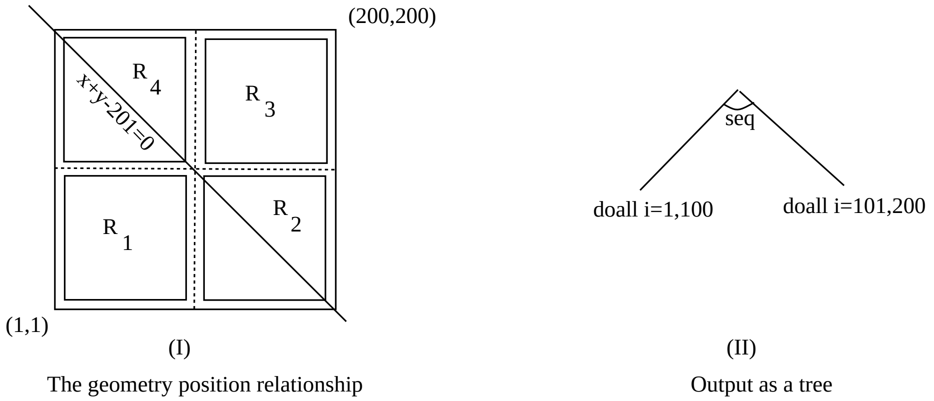


Figure 8: Geometrical Representation of $x + y = 201$

5. Related Work

Banerjee has proposed a transformation called index-set splitting [21, 3] to parallelize loops that have statements S and S' such that both $S\delta^f S'$ and $S\delta^a S'$ hold. His technique finds the point where flow-dependence changes to anti-dependence (or vice versa), which is the point where the loop is split. Consider the following example:

```
for i=1, 200
       ...
    S:  b(i) = ··· a(201 - i) ···
       ...
    S':  a(i) = ···
       ...
```

Notice the nature of dependences between S and S' across different iterations. $S(i)\delta^a S'(j)$ for $1 \leq i \leq 100, 101 \leq j \leq 200$ and $S'(i)\delta^f S(j)$ for $1 \leq i \leq 100, 101 \leq j \leq 200$. If this loop is split in the middle then iterations from 1 to 100 can be executed in parallel, followed by iterations 101 to 200 executed in parallel.

Note that our technique will yield a similar result. The geometric representation of the dependency line $x + y = 201$ and the generated output is shown in Figure 8.

The index-set splitting technique is more restricted than ours since it looks for the point where dependence changes direction, whereas our technique does not depend on such direction changes. Moreover, our partitioning technique can be used even when the dependence equation is non-linear (i.e. for loops in which dependency direction can change arbitrarily many times) as shown in example 4.4, and, in addition, has a nice intuitive geometric representation.

6. Conclusions

To conclude, in this paper we introduced a new geometric test, called the Diagonal test, that is equivalent to the commonly used Banerjee test. This test is more intuitive, easier to understand and use, especially for single *DO-loop*. For deeply nested loop, however, the Diagonal test may be more complex to use than the Banerjee test since in the worst case the Diagonal test may have to try more than one diagonals. We also introduced a new technique for parallelizing and scheduling a loop that has dependencies among its iterations that are of ill-defined and irregular nature. With this technique, we can successfully parallelize a loop, that may not be parallelizable by other techniques described in the literature. Moreover, our parallelizing technique is also applicable when the dependency equation is non-linear. Our technique has some resemblance to Banerjee's Index-set Splitting technique, but our technique is arguably more general, has much wider applicability, and is also applicable to cases where dependence equation is non-linear.

References

[1] R. Allen and K. Kennedy. Automatic Translation of Fortran Programs to Vector Form. In *ACM TOPLAS*, Vol 9, No. 4, 1987.

[2] U. Banerjee. Data Dependence in Ordinary Programs. M.S. Thesis, University of Illinois at Urbana-Champain, 1976.

[3] U. Banerjee. Speedup of Ordinary Programs. Ph.D. Thesis, University of Illinois at Urbana-Champain, 1979.

[4] U. Banerjee. Dependence Analysis for Supercomputing. Kluwer Academic Publishers, Norwell, Mass., 1988.

[5] R. G. Cytron. Compile-time Scheduling and Optimization for Asynchronous Machines. Ph.D. Thesis, Department of Computer Science, University of Illinois. 1984.

[6] E. H. D'Hollander. Partitioning and Labeling of Index Sets in Do Loops with Constant Dependence. International Conference on Parallel Processing 1989. Vol. II, pp. 139-144.

[7] M. Gupta and P. Banerjee. Demonstration of Automatic data Partitioning Techniques for Parallelizing Compilers on Multiprocessors. IEEE Transactions on Parallel And Distributed Systems, Vol. 3, No. 2, pp. 179-193.

[8] C-T King and L. M. Ni. Grouping in nested loops for parallel execution on multicomputers. International Conference on Parallel Processing 1989, Vol II, pp. 31-38.

[9] X. Kong, D. Klappholz, and K. Psarris. The I-test: An Improved Dependence Test for Automatic Parallelization and Vectorization. IEEE Transactions on Parallel and Distributed Systems, Vol. 2, no. 3, pp. 342-349, July 1991.

[10] Li and M. Chen. Index Domain Alignment: Minimizing cost of cross-referencing between distributed arrays, In Frontiers 90: 3rd Symp. Frontiers of Massively Parallel Computation, College Park, MD, Oct. 1990.

[11] Z. Li, P-C Yew, and C-Q Zhu. An Efficient Data Dependence Analysis for parallelizing Compilers. IEEE Transactions on Parallel and Distributed Systems, Vol. 1, NO. 1, Jan. 1990. pp. 26-34.

[12] C. D. Polychronopoulos. Parallel Programming and Compilers. Kluwer Academic Publishers. 1988.

[13] C. D. Polychronopoulos, M. Girkar, et al. Parafrase-2: An Environment for parallelizing, Partitioning, Synchronizing, and Scheduling Programs on Multiprocessors. International Conference on Parallel Processing 1989, Vol. II, pp. 39-47.

[14] K. Psarris, D. Klappholz, and X. Kong, On the Accuracy of the Banerjee Test. Journal of Parallel and Distributed Computing 12, pp. 152-157, 1991.

[15] K. Psarris. Exact Data Dependence Analysis. International Conference on Supercomputing, July, pp. 19-23, 1992.

[16] V. Sarkar. PTRAN: The IBM Parallel Translation System. In Parallel Functional Languages and Compilers. B. L. Szymanski (ed), pp. 309-392, 1989.

[17] Z. Shen, Z. Li, and P-C Yew. An Empirical Study of Fortran Programs for Parallelizing Compilers. IEEE Transaction on Parallel and Distributed Systems, Vol. 1. NO. 3, July 1990. pp. 356-364.

[18] D. Tang and G. Gupta. Geometric Techniques for Parallelizing and Scheduling DO-loops. Technical Report, Laboratory for Logic, Databases, and Advanced Programming, New Mexico State University. Jan. 1994.

[19] D. Tang and G. Gupta. Geometric Techniques for Parallelizing and Scheduling DO-loops with Nonlinear Dependence. Technical Report, Laboratory for Logic, Databases, and Advanced Programming, New Mexico State University.

[20] Michael Wolfe. Optimizing Supercompilers for Supercomputers. The MIT Press, Cambridge, Massachusetts.

[21] H. Zima and B. Chapman. Supercompilers for Parallel and Vector Computers. ACM Press. 1991.