To appear in the Journal of Parallel Algorithms and Architecture

Compile Time Partitioning of Nested Loop Iteration Spaces with Non-uniform Dependences*

Swamy Punyamurtula[†], Vipin Chaudhary[‡], Jialin Ju and Sumit Roy
Parallel and Distributed Computing Laboratory
Dept. of Electrical and Computer Engineering
Wayne State University
Detroit, MI 48202

Abstract

In this paper we address the problem of partitioning nested loops with non-uniform *(irregular) dependence vectors. Parallelizing and partitioning of nested loops requires efficient inter-iteration dependence analysis. Although many methods exist for nested loop partitioning, most of these perform poorly when parallelizing nested loops with irregular dependences. Unlike the case of nested loops with* uniform dependences *these will have a complicated dependence pattern which forms a* non-uniform dependence vector *set. We apply the results of classical convex theory and principles of linear programming to iteration spaces and show the correspondence between minimum dependence distance computation and iteration space tiling. Cross-iteration dependences are analyzed by forming an* Integer Dependence Convex Hull *(IDCH). Every integer point in this IDCH corresponds to a dependence vector in the iteration space of the nested loops. A simple way to compute minimum dependence distances from the dependence distance vectors of the extreme points of the IDCH is presented. Using these minimum dependence distances the iteration space can be tiled. Iterations within a tile can be executed in parallel and the different tiles can then be executed with proper synchronization. We demonstrate that our technique gives much better speedup and extracts more parallelism than the existing techniques.*

1 Introduction

In the past few years there has been a significant progress in the field of *Parallelizing Compilers*. Many new methodologies and techniques to parallelize sequential code have been developed and tested. Of

*This work has been supported in part by NSF MIP-9309489, US Army Contract DAEA-32-93-D-004 and Ford Motor Company Grant #0000952185. A preliminary version of this paper appeared in the Proceedings of the Symposium on Parallel and Distributed Processing 1994.

[†]AMD, Austin TX-78741,swamy@beast.amd.com
[‡]vipin@eng.wayne.edu

particular importance in this area is compile time partitioning of program and data for parallel execution. Partitioning of programs requires efficient and exact *Data Dependence* analysis. A precise dependence analysis helps in identifying dependent/independent partitions of a program. In general, nested loop program segments have a lot of scope for parallelization, since independent iterations of these loops can be distributed among the processing elements. So, it is important that appropriate dependence analysis be applied to extract maximum parallelism from these recurrent computations.

Although many dependence analysis methods exist for identifying cross-iteration dependences in nested loops, most of these fail in detecting the dependence in nested loops with coupled subscripts. These techniques are based on numerical methods which solve a set of *Diophantine Equations*. Even though these methods are generally efficient and detect dependences in many practical cases, in nested loops with coupled subscripts these techniques become computationally expensive. According to an empirical study reported by Shen et. al. [1], coupled subscripts appear quite frequently in real programs. They observed that nearly 45% of two-dimensional array references are coupled. Coupled array subscripts in nested loops generate non-uniform dependence vectors. Example 1(a) and Example 1(b) show nested loop program segments with uniform dependences and non-uniform dependences respectively. Example 1(a) has a uniform set of dependences $\{(1,0),(0,1)\}$ and its iteration space is shown in Fig.1(a). Array A in Example 1(b) has coupled subscripts and has a non-uniform dependence vector set. Figure 1(b) shows its iteration space and the irregularity of its dependences.

Example 1(a):

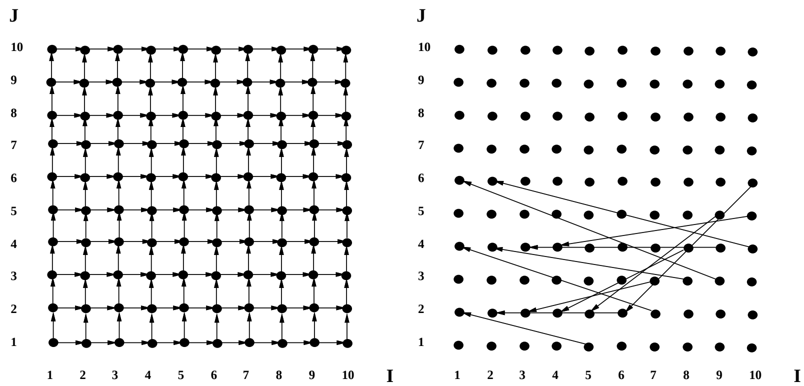
```
for I = 1, 10
    for J = 1, 10
        A(I,J) = ......
        ...... = A(I-1,J) + A(I, J-1)
    endfor
endfor
```

Example 1(b):

```
for I = 1,10
    for J = 1, 10
        A(2*J+3,I+1) = ......
        ...... = A(2*I+J+1,I+J+3)
    endfor
endfor
```

When the array subscripts are linear functions of loop indices (i.e., subscripts are coupled), some of the dependences among the iterations could be very complex and irregular. This irregularity in the dependence pattern is what makes the dependence analysis very difficult for these loops. A number of methods based on integer and linear programming techniques have been presented in the literature[2]. A serious disadvantage with these techniques is their high time complexity (NP - complete for integer programming methods). To analyze the cross-iteration dependences for these loops, we apply results from classical convex theory and present simple schemes to compute the dependence information.

Once the dependence analysis is carried out, the task now is to analyze and characterize the coupled dependences. These dependences can be characterized by *Dependence Direction Vectors* and *Dependence Distance Vectors* [3]. Computing these dependence vectors for loops with uniform dependences is simple and straight forward [4, 5]. But for nested loops with non-uniform dependences the dependence vector computation is an interesting problem. In such cases, it is very difficult to extract parallelism from the loops. Many approaches based on *vector decomposition* techniques have been presented in the literature [6, 7, 8]. These techniques represent the dependence vector set using a set



(a) Uniform Dependences (b) Nonuniform Dependences

Figure 1: Kinds of Iteration spaces

of basic dependence vectors. With the help of these basic dependence vectors, the iteration space can be partitioned for parallel execution. Normally iterations are aggregated into *groups* or *tiles* or *supernodes*. These aggregations are then executed with proper synchronization primitives to enforce the dependences. Most of these vector decomposition techniques consider nested loops with uniform dependences and they perform poorly in parallelizing nested loops with irregular dependences. Zaafrani and Ito [9] divide the iteration space of the loop into parallel and serial regions. All the iterations in the parallel region can be fully executed in parallel. Loop uniformization can then be applied to the serial region to find and exploit additional parallelism. In this paper we present partitioning schemes which extract maximum parallelism from nested loops.

Our approach to this problem is based on the theory of convex spaces. A set of *Diophantine equations* is formed from the array subscripts of the nested loops. These Diophantine equations are solved for integer solutions [4]. The loop bounds are applied to these solutions to obtain a set of inequalities. These inequalities are then used to form a dependence convex hull as an intersection of a set of half-spaces. We use an extended version of the algorithm presented by Tzen and Ni [8] to construct an integer dependence convex hull. An empty convex hull indicates absence of any cross-iteration dependence among the multi-dimensional array references of the nested loops considered. Every integer point in the convex hull corresponds to a dependence vector of the iteration space. The corner points of this convex hull form the set of extreme points for the convex solution space. These extreme points have the property that any point in the convex space can be represented as a convex combination of these extreme points [10]. The dependence vectors of these extreme points form a set of extreme vectors for the dependence vector set [11]. We compute the minimum dependence distances from these extreme vectors. Using these minimum dependence distances we tile the iteration space. For parallel execution of these tiles, parallel code with appropriate synchronization primitives is given. Our technique extracts maximum parallelism from the nested loops and can be easily extended to multiple dimensions.

The rest of the paper is organized as follows. In section two, we introduce the program model considered and review the related work previously done on tiling. Dependence analysis for tiling is also presented. Section three discusses dependence convex hull computation. Algorithms to compute the minimum dependence distances and our minimum dependence distance tiling schemes are given in section four. In section five a comparative performance analysis with Zafraani and Ito's techniques is presented to demonstrate the effectiveness of our scheme. Finally, we conclude in section six by summarizing our results and suggesting directions for future research.

2 Program Model and Background

We consider nested loop program segments of the form shown in Figure 2. For simplicity of notation and technique presentation we consider only doubly nested loops. However our method applies to multi-dimensional nested loops also. We consider tightly coupled nested loops only. The dimension of the nested loop segment is equal to the number of nested loops in it. For loop I(J), $L_I(L_J)$ and $U_I(U_J)$ indicate the lower and upper bounds respectively. We also assume that the program statements inside these nested loops are simple assignment statements of arrays. The dimensionality of these arrays is assumed to be equal to the nested loop depth. To characterize the coupled array subscripts, we assume the array subscripts to be linear functions of the loop index variables.

```
        for I = L_I, U_I
            for J = L_J, U_J
S_d:            A(f_1(I,J),f_2(I,J)) = ...
S_u:                ... = A(f_3(I,J),f_4(I,J))
            endfor
        endfor
```

Figure 2: Program Model

In our program model statement S_d defines elements of array A and statement S_u uses them. Dependence exists between S_d and S_u whenever both refer to the same element of array A. If the element defined by S_d is used by S_u in a subsequent iteration, then a *flow dependence* exists between S_d and S_u and is denoted by $S_d \delta^f S_u$. On the other hand if the element used in S_u is defined by S_d at a later iteration, this dependence is called *anti dependence* denoted by $S_d \delta^a S_u$. Other types of dependences like *output dependence* and *input dependence* can also exist but these can be eliminated or converted to flow or anti dependences.

An *iteration vector* $\vec{\imath}$ represents a set of statements that are executed for a specific value of (I,J) = (i,j). It forms an iteration space $\vec{I}$ defined as:

$$\vec{\imath} \in \vec{I} \;=\; \{(i,j) \mid L_I \leq i \leq U_I, L_J \leq j \leq U_J; i,j \in Z\}$$

where Z denotes the set of integers. For any given iteration vector $\vec{\imath}$ if there exists a dependence between S_d and S_u it is called *intra-iteration dependence*. These dependences can be taken care of by considering an iteration vector as the unit of work allotted to a processor. The dependence between S_d and S_u for two different iteration vectors $\vec{\imath}_1$ and $\vec{\imath}_2$ is defined as *cross-iteration dependence* and is represented by the dependence distance vector $\vec{d} = \vec{\imath}_2 - \vec{\imath}_1$. These dependences have to be honored while partitioning the iteration space for parallel execution. A dependence vector set $\vec{D}$ is a collection of all such distinct dependence vectors in the iteration space and can be defined as

$$\vec{D} = \{\vec{d} \mid \vec{d} = \vec{\imath}_2 - \vec{\imath}_1; \vec{\imath}_1, \vec{\imath}_2 \in \vec{I}\}$$

If all the iteration vectors in the iteration space are associated with the same set of constant dependence vectors, then such a dependence vector set is called a *uniform dependence vector set* and the nested loops are called *shift-invariant nested loops*. Otherwise, the dependence vector set is called a *non-uniform dependence vector set* and the loops are called *shift-variant nested loops*. Normally, coupled array subscripts in nested loops generate such non-uniform dependence vector sets and irregular dependences. The dependence pattern shown in Figure 1(b) is an example of such patterns. Because of the irregularity of these dependences, it is extremely difficult to characterize this dependence vector set. By characterizing the dependence vector set we mean representing or approximating it by a set of basic dependence vector set. The advantage with such characterization is that this dependence information can be used to tile the iteration space. In the following subsection, we review the work previously done to compute these basic dependence vectors and tiling. We also point out the deficiencies and disadvantages of those methods.

2.1 Related Work on Extreme Vector Computation and Tiling

Irogoin and Triolet [7] presented a partitioning scheme for hierarchical shared memory systems. They characterize dependences by convex polyhedron and form dependence cones. Based on the generating systems theory [12], they compute *extreme rays* for dependence cones and use a *hyperplane* technique [13] to partition the iteration space into *supernodes*. The supernodes are then executed with proper synchronization primitives. However the extreme rays provide only a dependence direction and not a distance. Also, their paper does not discuss any automatic procedure to form the rays and choose the supernodes. Our approach presents algorithms to form the extreme vectors which also give information about the dependence distance.

Ramanujam and Sadayappan [14, 6] proposed a technique which finds *extreme vectors* for tightly coupled nested loops. Using these extreme vectors they *tile* the iteration spaces. They derive expressions for optimum tile size which minimizes inter-tile communications. While their technique applies to distributed memory multiprocessors, it works only for nested loops with uniform dependence vectors. For parallel execution both tiles and supernodes need barrier synchronization which degrades the performance due to hotspot conditions [8].

Tzen and Ni [8] proposed the *dependence uniformization* technique. This technique computes a set of basic dependence vectors using the dependence slope theory and adds them to every iteration in the iteration space. This uniformization helps in applying existing partitioning and scheduling techniques, but it imposes too many dependences to the iteration space which otherwise has only a few of them. We provide algorithms to compute more precise dependence information and use better partitioning schemes to extract more parallelism from the nested loops.

Ding-Kai Chen and Pen-Chung Yew [15] presented a scheme which computes *Basic Dependence Vector Set* and schedules the iterations using *Static Strip Scheduling*. They extend the dependence uniformization work of Tzen and Ni and give algorithms to compute better basic dependence vector sets which extract more parallelism from the nested loops. While this technique is definitely an improvement over [8], it also imposes too many dependences on the iteration space, thereby reducing the extractable parallelism. Moreover this uniformization needs lot of synchronization. We present better parallelizing techniques to extract the available parallelism.

The *three region execution* of loops with a single dependence model by Zaafrani and Ito [9] divides the iteration space into three regions of execution. The first area consists of parts of the iterations space where the destination iteration comes lexically before the source ie. the direction vector satisfies $(>, *)$ or $(=, >)$ using conventional direction vector definition [4]. The next area represents the part of the iteration space where the destination iteration comes lexically after the source iteration and the source iteration is in the first area. Thus, iterations in these two areas can be executed in parallel as long as the two areas execute one after the other. The third area consists of the rest of the iteration space which should be executed serially or dependence uniformization techniques can be applied. However, as already pointed out, this method would introduce too many dependences. Moreover, as we will show our technique uses a better partitioning technique.

2.2 Dependence Analysis for Tiling

For the nested loop segment shown in Figure 2, a dependence exists between statements S_d and S_u if they both refer to the same element of array A. This happens when the subscripts in each dimension are equal. In other words, if $f_1(i_1, j_1) = f_3(i_2, j_2)$ and $f_2(i_1, j_1) = f_4(i_2, j_2)$ then a cross iteration dependence exists between S_d and S_u. We can restate the above condition as "cross-iteration dependence exists between S_d and S_u *iff* there is a set of integer solutions (i_1, j_1, i_2, j_2) to the system of Diophantine equations (1) and the system of linear inequalities (2)".

$$\begin{aligned} f_1(i_1, j_1) &= f_3(i_2, j_2) \\ f_2(i_1, j_1) &= f_4(i_2, j_2) \end{aligned} \tag{1}$$

$$\begin{aligned} L_I &\le i_1 \le U_I \\ L_J &\le j_1 \le U_J \\ L_I &\le i_2 \le U_I \\ L_J &\le j_2 \le U_J \end{aligned} \tag{2}$$

We use algorithms given by Banerjee [4] to compute the general solution to these Diophantine equations. This general solution can be expressed in terms of two integer variables x and y, except when $f_1(i_1, j_1) = f_3(i_2, j_2)$ is parallel to $f_2(i_2, j_2) = f_4(i_4, j_4)$, in which case the solution is in terms of three integer variables [8]. Here, we consider only those cases for which we can express the general solution in terms of two integer variables. We have (i_1, j_1, i_2, j_2) as functions of two variables x, y, which can be written as

$$(i_1, j_1, i_2, j_2) = (s_1(x,y), s_2(x,y), s_3(x,y), s_4(x,y))$$

Here s_i are functions with integer coefficients. We can define a solution set **S** which contains all the ordered integer sets (i_1, j_1, i_2, j_2) satisfying (1) ie.

$$\mathbf{S} = \{(i_1, j_1, i_2, j_2) \quad | f_1(i_1, j_1) = f_3(i_2, j_2) \cap f_2(i_1, j_1) = f_4(i_2, j_2) \quad \}$$

For every valid element (i_1, j_1, i_2, j_2) there exists a dependence between the statements S_d and S_u for iterations (i_1, j_1) and (i_2, j_2). The dependence distance vector $\vec{d}$ is given as $\vec{d} = (i_2 - i_1, j_2 - j_1)$ with dependence distances $d_i = i_2 - i_1$ and $d_j = j_2 - j_1$ in the i and j dimensions, respectively. So, from the general solution the dependence vector function D(x,y) can be written as

$$D(x,y) = \{(s_3(x,y) - s_1(x,y)), (s_4(x,y) - s_2(x,y))\}$$

The dependence distance functions in i, j dimensions can be given as $d_i(x,y) = s_3(x,y) - s_1(x,y)$ and $d_j(x,y) = s_4(x,y) - s_2(x,y)$. The dependence distance vector set $\vec{d}$ is the set of vectors $\vec{d} = \{(d_i(x,y), d_j(x,y))\}$. The two integer variables x,y span a solution space Γ given by

$$\Gamma = \{(x,y) \quad | \quad s_i(x,y) \quad satisfies \quad (1)\}$$

Any integer point (x,y) in this solution space causes a dependence between statements S_d and S_u, provided the system of inequalities given by (2) are satisfied. In terms of the general solution, the system of inequalities can be written as

$$\begin{aligned} L_I &\le s_1(x,y) \le U_I \\ L_J &\le s_2(x,y) \le U_J \\ L_I &\le s_3(x,y) \le U_I \\ L_J &\le s_4(x,y) \le U_J \end{aligned} \tag{3}$$

These inequalities bound the solution space and form a *convex polyhedron*, which can also be termed as *Dependence Convex Hull* (DCH) [8]. In the following section we give a brief introduction to convex set theory and explain how we apply well known results of convex spaces to iteration spaces.

3 Dependence Convex Hull

To extract useful information from the solution space Γ, the inequalities in (3) have to be applied. This bounded space gives information on the cross-iteration dependences. Tzen and Ni [8] proposed an elegant method to analyze these cross-iteration dependences. They formed a DCH from the solution space Γ and the set of inequalities (3). We extend their algorithm to compute more precise dependence information by forming an Integer Dependence Convex Hull as explained in the following paragraphs. In this section, we first present a review of basics from convex set theory. Our algorithm to form the integer dependence convex hull is given in later subsections.

3.1 Preliminaries

Definition 1 *The set of points specified by means of a linear inequality is called a* **half space** *or* **solution space** *of the inequality.*

For example $L_I \leq s_1(x,y)$ is one such inequality and a set $\mathbf{s} = \{(x,y)|s_1(x,y) \geq L_I\}$ is its half space. The inequalities given by (2) are weak inequalities, so the half spaces defined by them are closed sets. From (3) we have eight half spaces. The intersection of these half spaces forms a convex set.

Definition 2 *A* **convex set** *X can be defined as a set of points X_i which satisfy the convexity constraint that for any two points X_1 and X_2, $\lambda X_1 + (1-\lambda)X_2 \in X$, where $\lambda \in [0,1]$.*

Geometrically, a set is convex if, given any two points in the set, the straight line segment joining the points lies entirely within the set. The corner points of this convex set are called extreme points.

Definition 3 *A point in a convex set which does not lie on a line segment joining two other points in the set is called an* **extreme point**.

Every point in the convex set can be represented as a convex combination of its extreme points. Clearly any convex set can be generated from its extreme points.

Definition 4 *A* **convex hull** *of any set X is defined as the set of all convex combinations of the points of X.*

The convex hull formed by the intersection of the half spaces defined by the inequalities (3) is called a *Dependence Convex Hull*. This DCH can be mathematically represented as

$$\begin{aligned} \mathbf{D} &= \{(x,y)|L_I \leq s_1(x,y) \leq U_I\} \\ &\cap \{(x,y)|L_J \leq s_2(x,y) \leq U_J\} \\ &\cap \{(x,y)|L_I \leq s_3(x,y) \leq U_I\} \\ &\cap \{(x,y)|L_J \leq s_4(x,y) \leq U_J\} \end{aligned} \tag{4}$$

This DCH is a convex polyhedron and is a subspace of the solution space Γ. If the DCH is empty then there are no integer solutions (i_1, j_1, i_2, j_2) satisfying (2). That means there is no dependence between statements S_d and S_u in the program model. Otherwise, every integer point in this DCH represents a dependence vector in the iteration space. So, constructing this DCH serves two purposes viz., it gives precise information on the dependences and it also verifies whether there are any points in **S** within the region bounded by (2).

3.2 Computation of Integer Dependence Convex Hull

We use the algorithm given by Tzen and Ni [8] to form the DCH. Their algorithm forms the convex hull as a ring connecting the extreme points (nodes of the convex hull). The algorithm starts with a large solution space and applies each half space from the set defined by (3). The nodes are tested to find whether they lie inside this half space. It is done by assigning a *zoom* value to each node. If *zoom* = 1 then the node is outside the half space. Otherwise, if *zoom* = 0 then it is within the half space. If for any node the *zoom* value is different from its previous node then an intersection point is computed and is inserted into the ring between the previous node and the current node. When all the nodes in the DCH ring are tested, those nodes with *zoom* = 1 are removed from the ring. This procedure is repeated for every half space and the final ring contains the extreme points of the DCH. The extreme points of this convex hull can have real coordinates, because these points are just intersections of a set of hyperplanes. We extend this algorithm to convert these extreme points with real coordinates to extreme points with integer coordinates. The main reason for doing this is that we use the dependence vectors of these extreme points to compute the minimum and maximum dependence distances. Also, it can be easily proved that the dependence vectors of these extreme points form extreme vectors for the dependence vector set [11]. This information is otherwise not available for non-uniform dependence vector sets. We will explain how we obtain this dependence distance information in the following sections. We refer to the convex hull with all integer extreme points as *Integer Dependence Convex Hull* (IDCH).

The IDCH contains more accurate dependence information as explained later. So, constructing such an IDCH for a given DCH is perfectly valid as long as no *useful* dependence information is lost [11]. After constructing the initial DCH, our algorithm checks if there are any real extreme points for the DCH. If there are none, then IDCH is itself the DCH. Otherwise we construct an IDCH by computing integer extreme points. For every real extreme point, this algorithm computes two closest integer points on either side of the real extreme point. As the DCH is formed as a ring, for every node (real_node) there is a previous node (prev_node) and a next node (next_node). Two lines, prev_line joining prev_node and real_node, next_line joining real_node and next_node are formed. Now, we are looking for integer points on or around these two lines closest to the real_node but within the DCH. Figure 3 schematically explains the method. We have a simple algorithm to find these integer points [11]. The worst case complexity of this algorithm is bounded by O(A) where A is the area of the DCH. It should be emphasized here that considering the nature of the DCH, in most of the cases the integer extreme points are computed without much computation. The kind of speedup we get with our partitioning techniques based on this conversion, makes it affordable.

We demonstrate the construction of the DCH and IDCH with an example. Consider example 1(b) whose iteration space is shown in Figure 1(b). Two Diophantine equations can be formed from the subscripts of Array A.

$$\begin{aligned} 2*j+3 &= 2*i+j+1 \\ i+1 &= i+j+3 \end{aligned} \tag{5}$$

By applying the algorithm given by Banerjee [4], we can solve these equations. We can obtain the general solution $(s_1(x,y), s_2(x,y), s_3(x,y), s_4(x,y))$ to be $(x, y, -x+2y+4, 2x-2y-6)$. So the dependence vector function can be given as $D(x,y) = (-2x+2y+4, 2x-3y-6)$. The solution

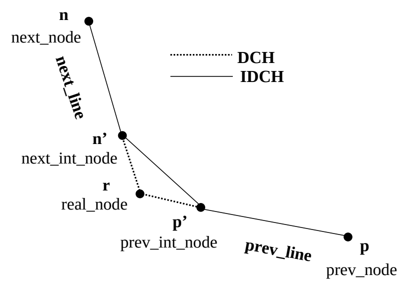


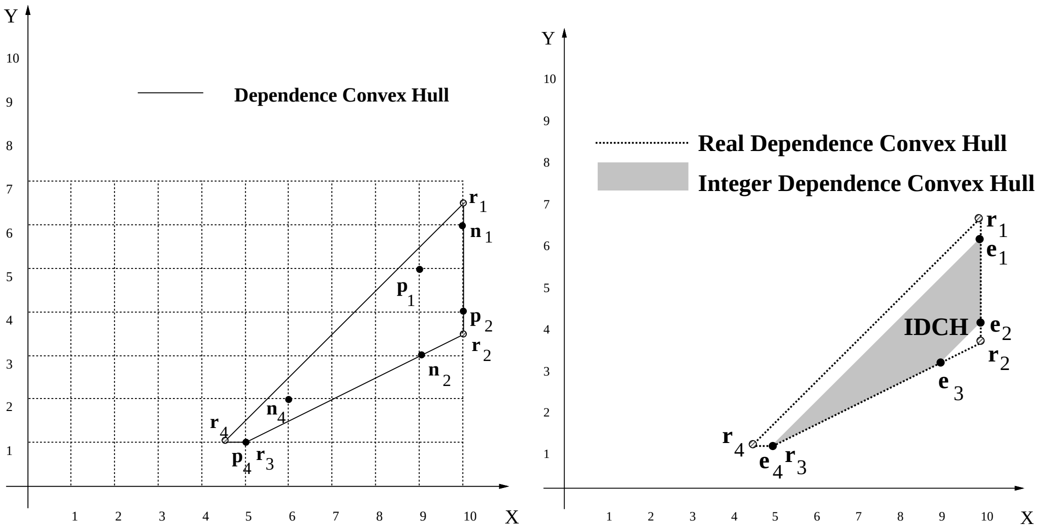
Figure 3: Computation of integer intersection points

space **S** is the set of points (x, y) satisfying the solution given above. Now, the set of inequalities can be given as

$$\begin{array}{rcl} 1 \leq & x & \leq 10 \\ 1 \leq & y & \leq 10 \\ 1 \leq & -x + 2y + 4 & \leq 10 \\ 1 \leq & 2x - 2y - 6 & \leq 10 \end{array} \tag{6}$$

Figure 4(a) shows the dependence convex hull DCH constructed from (6). This DCH is bounded by four nodes $\mathbf{r_1}$=(10,6.5), $\mathbf{r_2}$=(10,3.5), $\mathbf{r_3}$=(5,1), $\mathbf{r_4}$=(4.5,1). Because there are three real extreme points $(\mathbf{r_1}, \mathbf{r_2}, \mathbf{r_4})$, our Algorithm IDCH converts these real extreme points to integer extreme points. The DCH is scanned to find if there are any real extreme points. For a real extreme point, as explained previously, it forms two lines prev_line and next_line. For example, consider the node $\mathbf{r_2}$=(10,3.5). The node $\mathbf{r_1}$ is $\mathbf{r_2}$'s prev_node and $\mathbf{r_3}$ is its next_node. So, a prev_line joining $\mathbf{r_2}$ and $\mathbf{r_1}$, and a next line joining $\mathbf{r_2}$ and $\mathbf{r_3}$ are formed. As shown in Figure 4(a) the integer points closest to $\mathbf{r_2}$ and lying on the prev_line and next_line are $\mathbf{p_2}$ and $\mathbf{n_2}$. So, these two nodes are inserted into the ring. Similarly for the other real nodes $\mathbf{r_1}$ and $\mathbf{r_4}$ also, the integer nodes are computed as $\mathbf{p_1}$, $\mathbf{n_1}$ and $\mathbf{p_4}$, $\mathbf{n_4}$, respectively. So, the new DCH, i.e., the IDCH is formed by the nodes $\mathbf{n_1}$, $\mathbf{p_2}$, $\mathbf{n_2}$, $\mathbf{p_4}$ (here $\mathbf{p_4}$ and $\mathbf{r_3}$ coincide), $\mathbf{n_4}$ and $\mathbf{p_1}$. But, to represent the convex hull its extreme points (corner points) are enough. So, $\mathbf{n_4}$ and $\mathbf{p_1}$ can be removed from the ring. The resulting IDCH with four extreme points $(\mathbf{e_1}, \mathbf{e_2}, \mathbf{e_3}, \mathbf{e_4})$ is shown in Figure 4(b). While joining these extreme points our algorithm takes care to preserve the convex shape of the IDCH.

As can be seen from Figure 4(b) the IDCH is a subspace of DCH. So it gives more precise dependence information. We are interested only in the integer points inside the DCH. No useful information is lost while changing the DCH to IDCH [11]. In the following section we demonstrate how these extreme points are helpful in obtaining the minimum dependence distance information.



(a) IDCH Computation

(b) DH and IDCH

Figure 4: IDCH technique for Example 1(b)

3.3 Extreme Vectors and Extreme Rays

Every integer point in the Integer Dependence Convex Hull corresponds to a dependence vector in the dependence vector set $\vec{D}$. As per Definition 3 the corner points of a convex hull are called extreme points and any point in the convex hull can be represented as a convex combination of these extreme points. Using this we formulate a theorem which help us determine the extreme vector set for the dependence vector set $\vec{D}$.

Definition 5 *A set of vectors $\vec{E} \subset \vec{D}$, is called an* **extreme vector** *set for the dependence vector set $\vec{D}$ if every vector in $\vec{D}$ can be represented as a convex combination of the vectors in the set $\vec{E}$.*

Theorem 1 *The dependence vectors of extreme points of the* IDCH *form extreme vectors of the dependence vector set.*

Proof: Let us consider an IDCH with a set of extreme points $\mathbf{e}_1$, $\mathbf{e}_2$, ...$\mathbf{e}_n$. Also let D be the dependence vector set of the dependence space defined by the IDCH. From the property of extreme points of a convex space we know that any point $\mathbf{x}_1$ in the IDCH can be represented as a convex combination of the extreme points, $\mathbf{e}_i$. Therefore, we have

$$\mathbf{x}_1 = \lambda_1 * \mathbf{e}_1 + \lambda_2 * \mathbf{e}_2 + .. + \lambda_i * \mathbf{e}_i \quad \sum_i \lambda_i = 1 \tag{7}$$

Now, suppose $d_{\mathbf{x}_1}$ is the dependence vector associated with the point $\mathbf{x}_1$ and $d_{\mathbf{e}_i}$ be the dependence vector of the extreme point $\mathbf{e}_i$. We know that the dependence vector function $d(x, y)$ is a linear function of the form $A\mathbf{x} + c$. Hence, we can write $d_{\mathbf{x}_1} = A\mathbf{x}_1 + c$. Similarly, we have $d_{\mathbf{e}_i} = A\mathbf{e}_i + c$. Now, from (7) we can write

$$d_{\mathbf{x}_1} = A[\lambda_1 * \mathbf{e}_1 + \lambda_2 * \mathbf{e}_2 + .. + \lambda_i * \mathbf{e}_i] + c$$

which can be rewritten as

$$d_{\mathbf{x}_1} = \lambda_1[A\mathbf{e}_1 + c] + \lambda_2[A\mathbf{e}_2 + c] + ... + \lambda_i[A\mathbf{e}_i + c]$$

So, we have

$$d_{\mathbf{x}_1} = \lambda_1 * d_{\mathbf{e}_1} + \lambda_2 * d_{\mathbf{e}_2} + ... + \lambda_i * d_{\mathbf{e}_i}$$

By denoting the dependence vectors of the extreme points $d_{\mathbf{e}_i}$ as $\vec{\mathbf{e}}_i$ we can write

$$d_{\mathbf{x}_1} = \lambda_1 * \vec{\mathbf{e}}_1 + \lambda_2 * \vec{\mathbf{e}}_2 + ... + \lambda_i * \vec{\mathbf{e}}_i \tag{8}$$

Hence the result. □

Ramanujam and Sadayappan [6] formulated an approach to find extreme vectors for uniform dependence vector sets. Their extreme vectors need not be a subset of the dependence vector set, whereas our extreme vector set $\vec{E}$ is a subset of the dependence vector set $\vec{D}$. This gives a more precise characterization of the dependence vector set.

Definition 6 *A* **Convex Cone** $\mathcal{C}$ *is defined as a convex set in which* $\lambda x \in \mathcal{C}$ *for each* $x \in \mathcal{C}$ *and for each* $\lambda \geq 0$.

So, this convex cone $\mathcal{C}$ consists entirely of rays emanating from origin. The dependence vector set $\vec{D}$ spans a similar convex cone called Dependence Cone $\mathcal{C}$. Since, the extreme vector set $\vec{E}$ represents the set $\vec{D}$, we can form $\mathcal{C}$ from $\vec{E}$ itself. Therefore we can write

$$\mathcal{C} = \sum_{j=1}^{n} \gamma_j \vec{\mathbf{e}}_j \ ; \ \ \gamma_j \geq 0 \ and \ \vec{\mathbf{e}}_j \in \vec{E} \tag{9}$$

Definition 7 *An* **extreme direction** *of a convex set is a direction of the set that cannot be represented as a positive combination of two distinct directions of the set.*

Definition 8 *Any ray that is contained in the convex set and whose direction is an extreme direction is called an* **extreme ray**.

Each one of the extreme vectors $\vec{\mathbf{e}}_j$ spanning the dependence cone is a ray of this dependence cone. Hence the dependence cone can be characterized by its rays. From Definition 7 the non-extreme directions can be represented as a positive combination of the extreme directions. So, the extreme directions or extreme rays are enough to characterize the cone $\mathcal{C}$. Therefore the dependence cone $\mathcal{C}$ can be completely characterized by a set of extreme rays $\vec{\mathcal{R}}$.

$$\mathcal{C} = \sum_{j=1}^{n} \alpha_j \mathbf{r}_j \ \ \alpha_j \geq 0 \tag{10}$$

Irigoin and Triolet [7] also formed a set of extreme rays based on the generating systems theory. But, the distinct feature of our extreme rays is that these provide dependence distance information also. As shown in the later sections, this dependence distance information is used to partition the iteration space.

Theorem 2 *The set of extreme rays $\vec{\mathcal{R}}$ of the dependence cone $\mathcal{C}$ is a subset of the extreme vector set $\vec{E}$.*

Proof: From Theorem 1 we know that any dependence vector in the set $\vec{D}$ can be represented as a convex combination of the extreme vectors $\vec{\mathbf{e}}_j$. Now, from definition the extreme rays of the cone are those vectors which cannot be represented as a positive combination of two distinct vectors. So, if $\vec{\mathcal{R}}$ is not a subset of $\vec{E}$ then Theorem 1 is false. So, it implies $\vec{\mathcal{R}} \subseteq \vec{D}$. □

Characterizing the dependence vector set with the extreme rays has an advantage in that the actual dependence cone size can be computed. Zhigang Chen and Weijia Shang [16] defined the *dependence cone size* as the length of the intersection curve $x_1^2 + x_2^2 = 1$ with the dependence cone.

$$\mathcal{C}_{size} = \int_{x \in \mathcal{C}, x_1^2 + x_2^2 = 1} dx \tag{11}$$

Since, the set of extreme rays is a subset of the dependence vector set we can easily see that the dependence cone size spanned by the rays $\vec{\mathcal{R}}$ gives the actual dependence cone size. This dependence cone size gives us a measure of the amount of parallelism.

As shown in Figure 4(b) the IDCH of Example 1(b) has four extreme points, $\mathbf{e}_1$ = (12,8), $\mathbf{e}_2$ = (12,5), $\mathbf{e}_3$ = (11,4) and $\mathbf{e}_4$ = (5,1). Correspondingly the dependence vector set has four extreme vectors $\vec{\mathbf{e}}_1$ = (-4,-6), $\vec{\mathbf{e}}_2$ = (-10,3), $\vec{\mathbf{e}}_3$ = (-10,4) and $\vec{\mathbf{e}}_4$ = (-4,1). These extreme vectors span a dependence cone shown in Figure 5. This dependence cone has two extreme rays $\vec{\mathbf{r}}_1$ = (-10,4) and $\vec{\mathbf{r}}_2$ = (-4,-6). The unit radius circle at the origin intersects the dependence cone and the length of the intersection curve is the dependence cone size. The larger the dependence cone the wider is the range of dependence slope. So, it is important that to estimate the available parallelism the actual size of the dependence cone is measured. We show in the later sections that most of the existing techniques fail to detect the available parallelism completely.

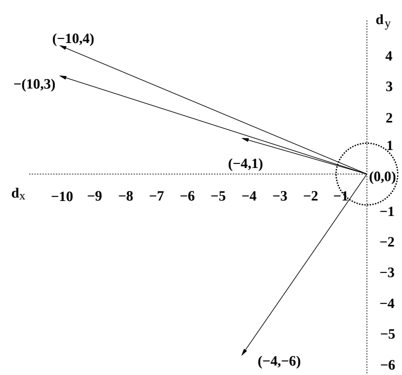


Figure 5: Dependence cone of Example 1(b)

In the following section we demonstrate how the extreme points and extreme vectors are helpful in obtaining the minimum dependence distance information.

4 Tiling with Minimum Dependence Distance

4.1 Minimum Dependence Distance Computation

The dependence distance vector function $D(x, y)$ gives the dependence distances d_i and d_j in dimensions i and j, respectively. For uniform dependence vector sets these distances are constant. But for the non-uniform dependence sets, these distances are linear functions of the loop indices. So we can write these dependence distance functions in a general form as

$$d_i(x, y) = a_1x + b_1y + c_1; \quad d_j(x, y) = a_2x + b_2y + c_2$$

where a_i, b_i, and c_i are integers and x, y are integer variables of the Diophantine solution space. These distance functions generate non-uniform dependence distances. Because there are unknown number of dependences at compile time, it is very difficult to know exactly what are the minimum and maximum dependence distances. For this we have to study the behavior of the dependence distance functions. The dependence convex hull contains integer points which correspond to dependence vectors of the iteration space. We can compute these minimum and maximum dependence distances by observing the behavior of these distance functions in the dependence convex hull. In this subsection, we present conditions and theorems through which we can find the minimum and maximum dependence distances.

We use a theorem from linear programming that states "*For any linear function which is valid over a bounded and closed convex space, its maximum and minimum values occur at the extreme points of the convex space*" [10, 17]. Theorem 1 is based on the above principle. Since both $d_i(x, y)$ and $d_j(x, y)$ are linear functions and are valid over the IDCH, we use this theorem to compute the minimum and maximum dependence distances in both i and j dimensions.

Theorem 3 *: The minimum and maximum values of the dependence distance function $d(x, y)$ occur at the extreme points of the* IDCH.

Proof: The extreme points of the IDCH are nothing but it's corner points. The general expression for dependence distance function can be given as $d(x, y) = ax + by + c$. If this function is valid over the IDCH, then the line $ax + by + c = k$ is a line passing through it. Now, suppose the minimum and maximum values of $d(x, y)$ are d_{min} and d_{max} respectively. The lines $ax + by + c = d_{min}$ and $ax + by + c = d_{max}$ are parallel to the line $ax + by + c = k$. Since the function $d(x, y)$ is linear, it is monotonic over the IDCH. Therefore, we have $d_{min} \leq k \leq d_{max}$ for any value of k, the function $d(x, y)$ assumes in the IDCH. Thus, the lines $ax+by+c = d_{min}$ and $ax+by+c = d_{max}$ are tangential to the IDCH and hence pass through the extreme points as shown in Figure 6. So, the function $d(x, y)$ assumes its maximum and minimum values at the extreme points. □

Hence, we can find the minimum dependence distance from the extreme vector list (dependence distance vectors of the extreme points). But as these minimum distances can be negative (for anti dependences) we have to find the absolute minimum dependence distance. For this we use Theorem 2 which states that if the distance functions $d_i(x, y)$ and $d_j(x, y)$ are such that the lines $d_i(x, y)$=0 and $d_j(x, y)$=0 do not pass through the IDCH, then the absolute minimum and absolute maximum values of these functions can be obtained from the extreme vectors. With the help of these theorems we can compute the minimum dependence distance.

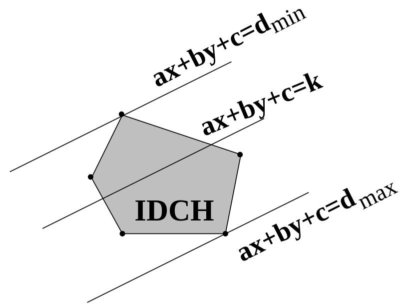
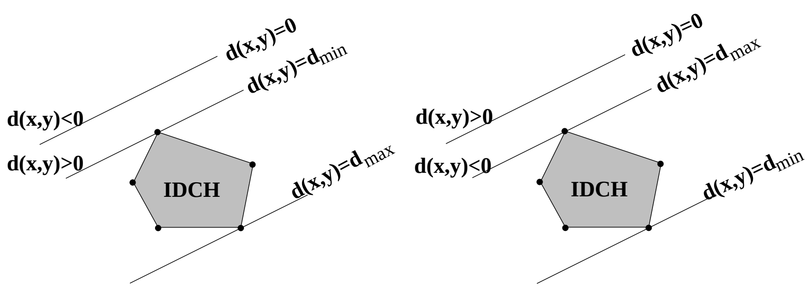


Figure 6: Minimum and maximum values of $d(x, y)$

Theorem 4 *: If $d(x, y)$ = 0 does not pass through the* IDCH *then the absolute minimum and absolute maximum values of $d(x, y)$ appear on the extreme points.*

Proof: If $d(x, y) = 0$ does not pass through the IDCH, then the IDCH is either in the $d(x, y) > 0$ or $d(x, y) < 0$ side. Let us consider the case where IDCH lies on the $d(x, y) > 0$ side as shown in Figure 7(a). By Theorem 1, the minimum and maximum values of $d(x, y)$ occur at the extreme points. The lines $d(x, y) = d_{min}$ and $d(x, y) = d_{max}$ are tangential to the IDCH. Since both d_{min} and d_{max} are positive, the absolute minimum and absolute maximum values are the minimum and maximum values of $d(x, y)$. For the case where IDCH lies on the $d(x, y) < 0$ side (Figure 7(b)), the minimum and maximum values of $d(x, y)$ are negative. So, the absolute minimum and absolute maximum values are the maximum and minimum values, respectively. □



(a) IDCH $\in d(x, y) > 0$ (b) IDCH $\in d(x, y) < 0$

Figure 7: Computation of abs(min) and abs(max) values of $d(x, y)$

For cases which do not satisfy theorem 2, we assume an absolute minimum dependence distance of 1. Using the minimum dependence distances computed above, we can tile the iteration space. In the following subsection, we present algorithms for tiling and show that our partitioning techniques give better speed up than the existing parallelizing techniques [8, 9].

4.2 Tiling and Tile Synchronization

In this subsection, we present algorithms to identify partitions (tiles) of the iteration space which can be executed in parallel. We also present synchronization schemes to order the execution of these tiles satisfying the inter-tile dependences.

The tiles are rectangular shaped, uniform partitions. Each tile consists of a set of iterations which can be executed in parallel. The minimum dependence distances d_{imin} and d_{jmin} can be used to determine the tile size. We first determine whether $d_i(x,y)$=0 passes through the IDCH. If it does not, then d_{imin} can be obtained by selecting the minimum dependence distance in dimension i of the set of extreme vectors. Otherwise, if $d_j(x,y)$=0 does not pass through the IDCH we can determine d_{jmin}. We consider these cases separately and propose suitable partitioning algorithms. With the help of examples we demonstrate the tiling and synchronization schemes.

Case I: $d_i(x,y)$*=0 does not pass through the* IDCH

In this case, as the $d_i(x,y)$=0 does not pass through the IDCH, the IDCH is either on the $d_i(x,y) > 0$ side or $d_i(x,y) < 0$ side. From theorem 2, the absolute minimum of d_i occurs at one of the extreme points. Suppose this minimum value of $d_i(x,y)$ is given by d_{imin}. Then, we can group the iterations along the dimension i into tiles of width d_{imin}. All the iterations in this tile can be executed in parallel as there are no dependences between these iterations (no dependence vector exists with $d_i < d_{imin}$). The height of these tiles can be as large as N where $N = U_J - L_J + 1$. Inter-iteration dependences can be preserved by executing these tiles sequentially. No other synchronization is necessary here. If the tiles are too large, they can be divided into sub-tiles without loss of any parallelism.

We can now apply this method to the nested loop program segment given in example 1(b). Its IDCH is shown in Fig. 4(b). Here $d_i(x,y)$=0 does not pass through the convex hull. So from theorem 2, the absolute value of the minimum dependence distance can be found to be d_{imin}=abs(-4)=4. This occurs at the extreme points (5,1) and (10,6). So, we can tile the iteration space of size $M * N$ with d_{imin}=4 as shown in Fig. 8. The number of tiles in the iteration space can be given as $T_n = \lceil \frac{N}{d_{imin}} \rceil$ except near the boundaries of the iteration space, where the tiles are of uniform size $M * d_{imin}$. Parallel code for example 1(b) can be given as in Figure 9. This parallel code applies to any nested loop segment that satisfies case 1 and of the form as given in 2 with L_I = 1, $U_I = N$, L_J = 1, $U_J = M$.

Theoretical speedup for this case can be computed as follows. Ignoring the synchronization and scheduling overheads, each tile can be executed in one time step. So, the total time of execution equals the number of tiles T_n. Speedup can be calculated as the ratio of total sequential execution time to the parallel execution time.

$$Speedup = \frac{M * N}{T_n}$$

Minimum speedup with our technique for this case is M, when $T_n = N$ (i.e., d_{imin}=1).

Case II: $d_j(x,y) = 0$ *does not pass through the* IDCH

Here, since $d_j(x,y)$=0 does not pass through the IDCH, we have d_{jmin} at one of the extreme points. As $d_i(x,y)$=0 goes through the IDCH, we take the absolute value of d_{imin} to be 1. So, we tile the

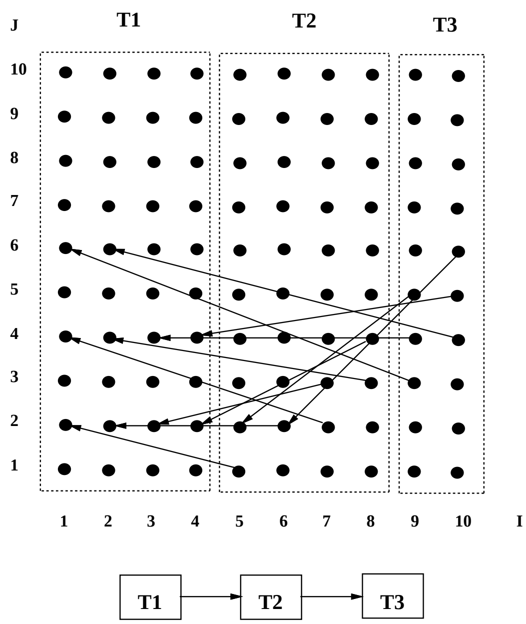


Figure 8: Tiling with minimum dependence distance d_i

Tile num T_n = $\lceil \frac{N}{d_{imin}} \rceil$

```
DOserial K = 1, T_n
    DOparallel I = (K-1)*d_imin+1, min(K*d_imin, N)
        DOparallel J = 1, M
            A(2*J+3,I+1) = ...... ;
            ...... = A(2*I+J+1,I+J+3);
        ENDDOparallel
    ENDDOparallel
ENDDOserial
```

Figure 9: Parallel code for scheme 1

iteration space into tiles with width=1 and height=d_{jmin}. This means, the iteration space of size $M * N$ can be divided into N groups with $T_n = \lceil \frac{M}{d_{jmin}} \rceil$ tiles in each group. Iterations in a tile can be executed in parallel. Tiles in a group can be executed in sequence and the dependence slope information of Tzen and Ni [8] can be used to synchronize the execution of inter-group tiles.

Tzen and Ni [8] presented a number of lemmas and theorems to find the maximum and minimum values of the *Dependence Slope Function* defined as DSF = $\frac{d_j(x,y)}{d_i(x,y)}$. This DSF is non-monotonic when $d_i(x, y)$=0 passes through the IDCH. Under this condition we have three different cases :

- If $d_j(x, y)$ is parallel to $d_i(x, y)$, i.e., if $d_j(x, y)$ is of the form $a * d_i(x, y) + b$, then the slope range is $(a - b, \infty)$ if $b > 0$ and $(a + b, a - b)$ if $b \leq 0$.

- If $d_j(x, y)$=constant b, then the dependence slope range is $(-b, \infty)$ if $b > 0$ and $(b, -b)$ if $b \leq 0$.

- $d_j(x, y)$ is not parallel to $d_i(x, y)$, the range of the dependence slope is $(-min(M - 1, P), \infty)$ where P is the maximal absolute value of $d_j(x, y)$ on the extreme points and can be obtained from the extreme vectors.

These minimum or maximum dependence slopes are then used to enforce the dependence constraints among the iterations. Thus the execution of the inter-group tiles can be ordered by applying a basic dependence vector with min(max) slope. Consider the nested loop given in Example 2. Figure 10 shows its IDCH. Note that $d_i(x, y)$=0 passes through the IDCH while $d_j(x, y)$=0 does not pass through the IDCH. The d_{jmin} can be computed to be 4 and the iteration space can be tiled as shown in Figure 11(a).

Example 2:

```
for I = 1, 10
    for J = 1, 10
        A(2*I+3,J+1) = ......
        ...... = A(2*J+I+1,I+J+3)
    endfor
endfor
```

We can apply the dependence slope theory explained above and find the minimum dependence slope as $-min(M - 1, P)$, where P=10 and M=11. Therefore, DSF_{min}=-10. Applying this to the iteration space, we find that an iteration i of any group (except the first one) can be executed as soon as the previous group finishes the $(i + 10)^{th}$ iteration. As we tile these iterations, we can compute the inter-group tile dependence slope as $T_s = \lceil \frac{DSF_{imin}}{d_{jmin}} \rceil$. So, we can synchronize the tile execution with a inter-group tile dependence vector (1,T_s). If T_s is negative, then this dependence vector forces a tile i of j^{th} group to be executed after the tile $i + |T_s|$ of group $j - 1$. Otherwise, if T_s is positive then a tile i of group j

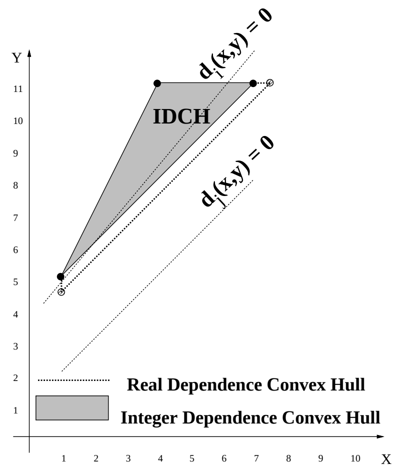
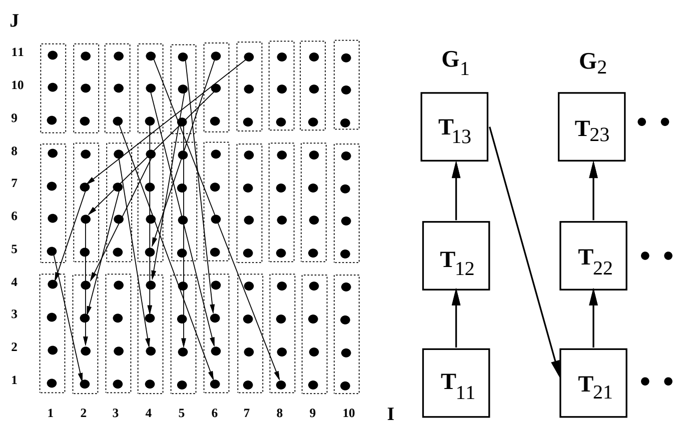


Figure 10: IDCH of Example 2

can be executed as soon as $(i - T_s)^{th}$ tile in group $j - 1$ is executed. Figure 11(b) shows the tile space graph for this example. In this figure G_i denotes a group and T_{ij} denotes $j^t h$ tile of group i. Parallel code for this example is given in Figure 12. This parallel code also applies to any nested loop segment of the form in Figure 2.



(a) Iteration Space

(b) Tile Synchronization

Figure 11: Tiling with minimum dependence distance d_j

Speedup for this case can be computed as follows. The total serial execution time is $M * N$. The parallel execution time is $T_n + (N-1) * T_s$. Hence, the speedup is given as

$$Speedup = \frac{M * N}{T_n + (N-1)T_s}$$

For the case where both $d_i(x, y)$=0 and $d_j(x, y)$=0 pass through the IDCH, we assume both d_{imin} and d_{jmin} to be 1. So, each tile corresponds to a single iteration. The synchronization scheme given in Figure 12 is also valid for this case. In the next section, we compare the performance of our technique with existing techniques and analyze the improvement in speedup.

Tile num $T_n = \lceil \frac{M}{d_{jmin}} \rceil$

Tile slope $T_s = \lceil \frac{DSF_{min}}{d_{jmin}} \rceil$

```
DOacross I = 1, N
    Shared integer J[N]
    DOserial J[I] = 1, T_n
        if (I > 1) then
            while (J(I-1) < (J(I)+T_s))
              wait;
        DOparallel K = (J[I]-1)*d_jmin+1, J[I]*d_jmin
            A(2I+3, K+1) = ...... ;
            ...... = A(I+2K+1,I+K+3);
        ENDDOparallel
    ENDDOserial
ENDDOacross
```

Figure 12: Parallel code for scheme 2

5 Performance Analysis

5.1 Comparison with Uniformization Techniques

The dependence uniformization method presented by Tzen and Ni [8] computes dependence slope ranges in the DCH and forms a *Basic Dependence Vector* (BDV) set which is applied to every iteration in the iteration space. The iteration space is divided into groups of one column each. Index synchronization is then applied to order the execution of the iterations in different groups. Our argument is that this method imposes too many dependences on the iteration space, thereby limiting the amount of extractable parallelism. Consider example 1(b). If we apply the dependence uniformization technique, a BDV set can be formed as $\{(0,1), (1,-1)\}$. The uniformized iteration space is shown in Figure 13. As we can see from this figure the uniformization imposes too many dependences. If index synchronization is applied, the maximum speedup that can be achieved by this technique is $Speedup_{unif} = \frac{M}{\phi}$, where ϕ = 1-t is the delay and t = $\lfloor DSF_{min} \rfloor$ or $\lceil DSF_{max} \rceil$. This speedup is significantly affected by the range of dependence slopes. If the dependence slopes vary over a wide range, in the worst case this method would result in serial execution. For the example under consideration (Example 1(b)) the speedup with uniformization technique is 5. Ding-Kai Chen and Pen-Chung Yew [15] proposed a similar but improved uniformization technique. Using their technique, the basic dependence vector set can be formed as {(1,0), (1,1), (2,-1)}. The iteration space is uniformized and static strip scheduling is used to schedule the iterations which gives a speedup of 12. The disadvantage with the above uniformization techniques is that by applying these BDVs at every iteration increases the synchronization overhead. Figure 8 shows the tiled iteration space obtained by applying our minimum dependence distance tiling method. From the analysis given in the previous section the speedup with our method is $\frac{M*N}{T_n}$, which is more than 30. So, our method gives a significant speedup compared to other techniques. Even for the case d_{imin}=1 our technique gives a speedup of 10 (M) which is better than Tzen and Ni's technique and same as Chen and Yew's. An important feature of our method is that the speedup does not depend on the range of dependence slopes.

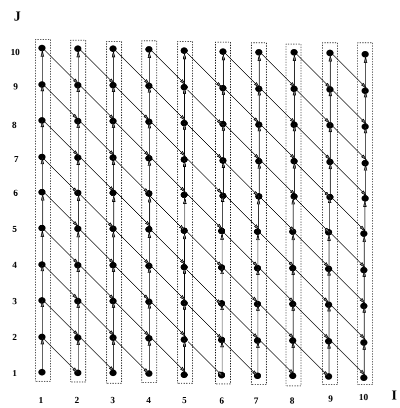


Figure 13: Uniformized Iteration space of Example 1(b)

For Example 2, $d_i(x,y) = 0$ passes through its IDCH and $d_j(x,y) = 0$ does not. So, we follow our second approach to tile its iteration space. For this example, the dependence uniformization technique forms the BDV set as {(0,1), (1,-10)} and the speedup can be calculated as $\frac{M}{\phi}$ = $\frac{11}{10} \simeq 1$. Our method gives a speedup of $\frac{M*N}{T_n+(N-1)*T_s} \simeq 3$. So, we have a significant speedup improvement in this case too. For the case where $d_{imin} = d_{jmin}$=1 our speedup is as good as the speedup with their technique.

Example 3:

```
for I = 1, 10
    for J = 1, 10
        A(J+2,2*I+4) = ......
        ...... = A(I-J+1,2*I-J+2)
    endfor
endfor
```

We can show that with IDCH we get more precise dependence information. Consider Example 3. The dependence vector functions can be computed as $d_i(x,y) = (x-y+1)$ and $d_j(x,y) = (2*x-3*y)$. So, the dependence slope function DSF is $\frac{2*x-3*y}{x-y+1}$. The DCH and IDCH for this example are shown in Figure 14. The extreme points of DCH are $\mathbf{r_1}$=(1.5,1), $\mathbf{r_2}$=(5,1) and $\mathbf{r_3}$=(8.5,8). After converting these real extreme points to integer extreme points, the IDCH has $\mathbf{e_1}$=(2,1), $\mathbf{e_2}$=(5,1) and $\mathbf{e_3}$=(8,7). Since $d_i(x,y)$=0 does not pass through the IDCH, the maximum and minimum values of the DSF occur at the extreme points of DCH and IDCH. For DCH, the DSF_{max}=1.4 and DSF_{min}=-4 which occur at $\mathbf{r_2}$ and $\mathbf{r_3}$, respectively. Similarly for IDCH, DSF_{max}=1.4 is at $\mathbf{e_2}$ and DSF_{min}=-2.5 is at $\mathbf{e_3}$. Clearly, the IDCH gives more accurate values. Since the speedup depends on the delay ϕ and as ϕ=1-t where t=$\lfloor DSF_{min} \rfloor$ or $\lceil DSF_{max} \rceil$, the speedup reduces when we use inaccurate dependence slope information. For this example, with DCH the delay ϕ=5. If we use IDCH then the delay reduces to $\phi = 1 - \lfloor -2.5 \rfloor$=4. Hence the accurate dependence slope information obtained with IDCH helps us to extract more parallelism from the nested loops.

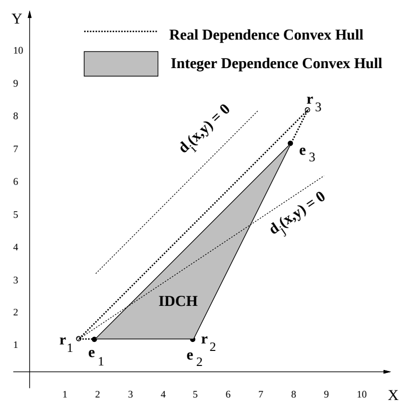


Figure 14: IDCH of Example 3

5.2 Experimental Comparison with the Three Region Approach

We ran our experiments on a Cray J916 with 8 processors and used the *Autotasking Expert System (Atexpert)* to analyze the performance. *Atexpert* is a tool developed by *Cray Research Inc. (CRI)* for accurately measuring and graphically displaying tasking performance from a job run on an arbitrarily loaded *CRI* system. The graphs shown indicate the speed-up expected by Amdahl's law, ignoring all multitasking overhead as well as the expected speedup on a dedicated system using the given program. A linear speed-up reference line is also plotted.

We used *User-directed tasking* directives to construct our fully parallelizable area in the iteration space. The format is as below.

```
#pragma _CRI parallel defaults
#pragma _CRI taskloop
    loop
#pragma _CRI endparallel
```

The first nested loop is as follows:

Experiment 1:

do $i = 1$, N

  do $j = 1$, N

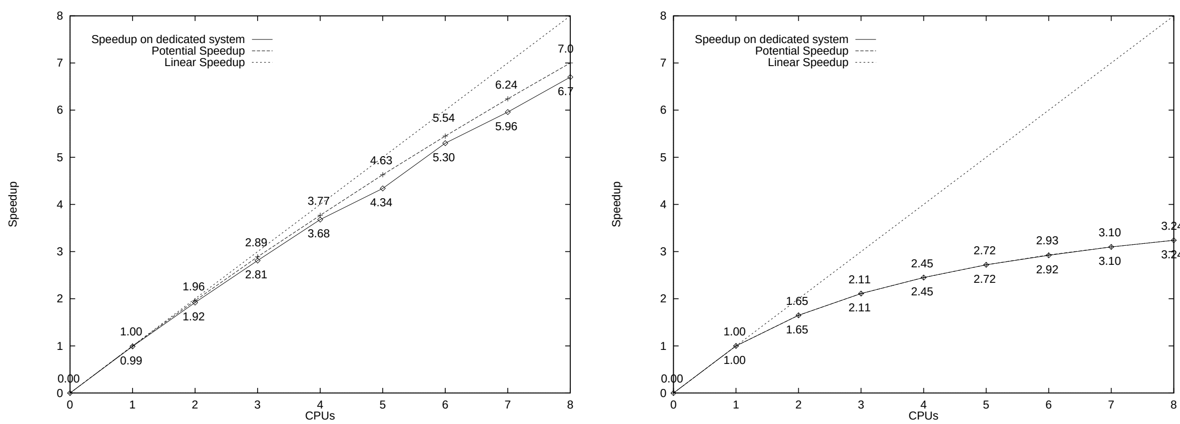
$$A(i + j + 50, 3 * i) = \cdots$$

$$\cdots = A(i, j)$$

  enddo

enddo

This loop has coupled subscripts in the array definition and the minimum distance is 50. Figure 15 shows the speedup of our technique versus Zaafrani and Ito's technique. For this loop, our technique got a better performance, since the minimum distance for this loop is large compared to the number of processors. Moreover our partitioning is more regular than theirs.



(a) IDCH method (b) Zaafrani and Ito's method

Figure 15: Speedups for Experiment 1

For the second experiment the array reference has coupled subscripts and the minimum distance is 9. The second experiment is:

Experiment 2:

do $i = 1$, N

  do $j = 1$, N

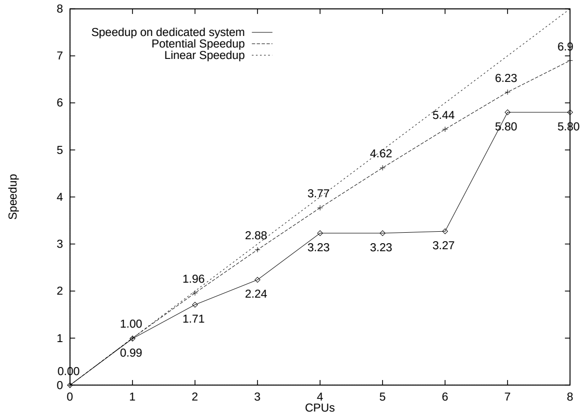
$$A(2 * i, j) = \cdots$$

$$\cdots = A(i - 9, i - j + 1)$$

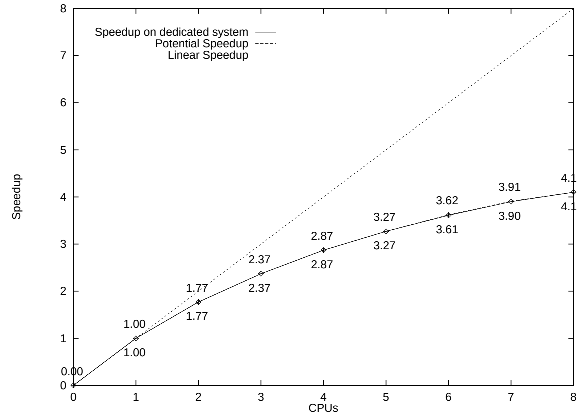
  enddo

enddo

The performance of our technique and Zaafrani's technique for this loop is shown on Figure 16. We can see that for our technique the speedup stayed almost the same for 4, 5 and 6 processors. The reason for this is that there is a load imbalance since the minimum distance does not fit the number of



(a) IDCH method



(b) Zaafrani and Ito's method

Figure 16: Speedups for Experiment 2

processors quite well for these cases. In practice, once we know the exact number of processors we want to use for some specific system and the minimum distance, we can adjust the partitioning distance number to match the system.

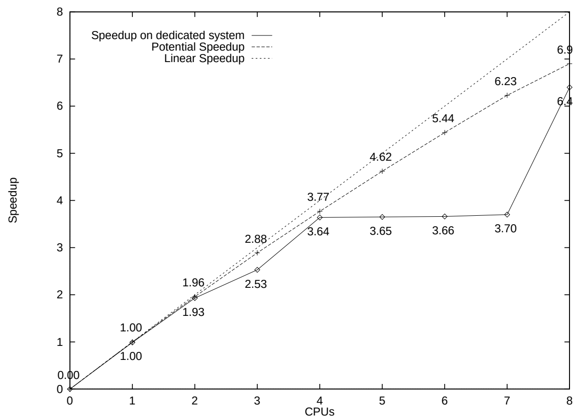
For the last experimental loop the coupling of the subscripts is relatively complicated. The minimum distance is 24.

Experiment 3:
do $i = 1, N$
  do $j = 1, N$
    $A(2*i+j-1, i-2*j+4) = \cdots$
    $\cdots = A(i-24, j)$
  enddo
enddo

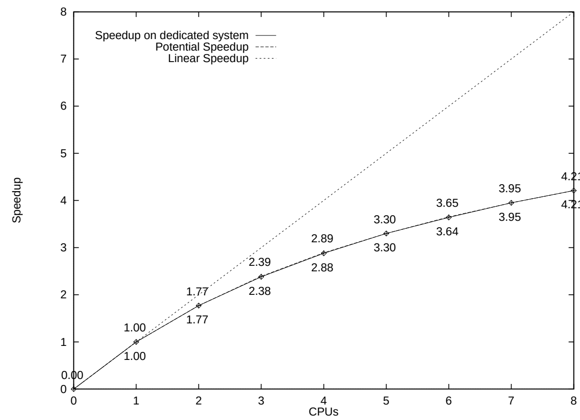
As before our performance is better than that of Zaafrani's as shown on Figure 17. For some of the experiments, we observe that our speedups are not increasing as evenly as they do with Zaafrani's technique due to load imbalance. This can be taken care of by choosing an appropriate distance which is less than the minimum distance. Moreover the Amdahl's law curves for our technique are consistently better than that for Zaafrani and Ito's method. This shows that the *IDCH* method tends to minimize sequential code in the transformed loop.

6 Conclusion

In this paper we have presented simple and computationally efficient partitioning and tiling techniques to extract maximum parallelism from nested loops with irregular dependences. The cross-iteration dependences of nested loops with non-uniform dependences are analyzed by forming an Integer Dependence Convex Hull. Minimum dependence distances are computed from the dependence vectors



(a) IDCH method



(b) Zaafrani and Ito's method

Figure 17: Speedups for Experiment 3

of the IDCH extreme points. These minimum dependence distances are used to partition the iteration space into tiles of uniform size and shape. Dependence slope information is used to enforce the inter-iteration dependences. We have shown, both theoretically and experimentally that our method gives much better speedup than existing techniques and exploits the inherent parallelism in the nested loops with non-uniform dependences.

Since the convex hull encloses all the dependence iteration vectors but not all the iteration vectors in the convex hull are dependences it is possible that some of the extreme points may not have a dependence. Our minimum distances are evaluated using these extreme points and thus we might underestimate the minimum distance. Our future research work is to eliminate such cases. We also plan to test this method for higher dimensional nested loops.

Acknowledgments

We would like to thank Dr. Chengzhong Xu for his many useful suggestions and the anonymous reviewers in making this paper a lot more readable.

References

[1] Z. Shen, Z. Li, and P.-C. Yew, "An empirical study on array subscripts and data dependencies," in *Proceedings of the International Conference on Parallel Processing*, pp. II–145 to II–152, 1989.

[2] W. Pugh, "A practical algorithm for exact array dependence analysis," *Communications of the A.C.M.*, vol. 35, pp. 102–114, Aug 1992.

[3] M. Wolfe, *Optimizing Supercompilers for Supercomputers*. The MIT Press Cambridge: Pitman Publishing, 1989.

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

[5] H. B. Ribas, "Obtaining dependence vectors for nested loop computations," in *Proceedings of the International Conference on Parallel Processing*, vol. II, pp. II–212 to 219, 1990.

[6] J. Ramanujam and P. Sadayappan, "Tiling of multidimensional iteration spaces for multicomputers," *Journal of Parallel and Distributed Computing*, vol. 16, p. 108 to 120, Oct 1992.

[7] F. Irigoin and R. Triolet, "Supernode partitioning," in *Conference Record of the 15th ACM Symposium on Principles of Programming Languages*, (San Deigo, CA), p. 319 to 329, 1988.

[8] T. H. Tzen and L. M. Ni, "Dependence uniformization: A loop parallelization technique," *IEEE Transactions on Parallel and Distributed Systems*, vol. 4, p. 547 to 558, May 1993.

[9] A. Zaafrani and M. R. Ito, "Parallel region execution of loops with irregular dependencies," in *Proceedings of the International Conference on Parallel Processing*, vol. II, pp. II–11 to II–19, 1994.

[10] M. S. Bazaraa, J. J. Jarvis, and H. D. Sherali, *Linear Programming and Network Flows*. John Wiley & sons, 1990.

[11] S. Punyamurtula, "Compile-time partitioning of nested loop iteration spaces with non-uniform dependence vectors," Master's thesis, Dept. of E.C.E, Wayne State University, 1994.

[12] A. Schrijver, *Theory of Linear and Integer Programming*. John Wiley & sons, 1986.

[13] L. Lamport, "The parallel execution of do loops," *Communications of the A.C.M.*, vol. 17, pp. 83–93, Feb 1974.

[14] J. Ramanujam and P. Sadayappan, "Tiling of iteration spaces for multicomputers," in *Proceedings of the International Conference on Parallel Processing*, pp. II–179 to II–186, Aug 1990.

[15] D.-K. Chen and P.-C. Yew, "A scheme for efeective execution of irregular doacross loops," in *Proceedings of the International Conference on Parallel Processing*, vol. II, pp. II–285 to II–292, 1992.

[16] Z. Chen and W. Shang, "On uniformization of affine dependence algorithms," in *Proceedings of the Fourth IEEE Symposium on Parallel and Distributed Processing*, (Arlington, Texas), p. 128 to 137, IEEE Computer Society Press, December 1992.

[17] W. A. Spivey, *Linear Programming, An Introduction*. The Macmillan company, 1967.