Minimum Dependence Distance Tiling of Nested Loops with Non-uniform Dependences

Swamy Punyamurtula and Vipin Chaudhary*

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. Although many methods exist for nested loop partitioning, most of these perform poorly when parallelizing nested loops with irregular dependences. 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. The cross-iteration dependences are analyzed by forming an Integer Dependence Convex Hull (IDCH). 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 in a tile can be executed in parallel and the tiles can 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 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. In general, nested loop program segments give a lot of scope for parallelization. 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.

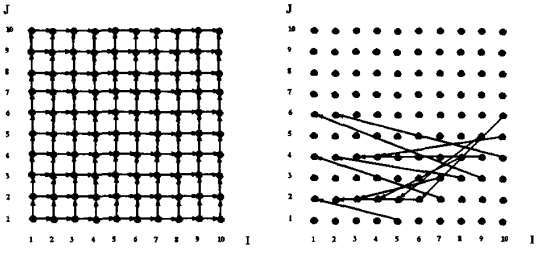
Although many dependence analysis methods exist for identifying cross-iteration dependences in nested loops, most of these fail in detecting the dependence in nested loops with coupled subscripts (i.e., subscripts are linear functions of loop indices). According to an empirical study reported by Shen 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
```



(a) (b)

Figure 1: *Iteration spaces with (a) Uniform dependences and (b) Non-uniform dependences*

Irregularity in the dependence pattern makes the dependence analysis very difficult for nested loops. A number of methods based on integer and linear programming techniques have been presented in the literature. A serious disadvantage with these techniques is their high time complexity. 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

*The author's work has been supported in part by NSF MIP-9309489 and Wayne State University Faculty Research award. e-mail: vipin@eng.wayne.edu

0-8186-6427-4/94 $04.00 © 1994 IEEE

task now is to analyze and characterize the coupled dependences. These dependences can be characterized by *Dependence Direction Vectors* and *Dependence Distance Vectors* [2]. Computing these dependence vectors for loops with uniform dependences is simple and straight forward [3]. But for nested loops with non-uniform dependences, the dependence vector computation is an interesting problem. Many approaches based on *vector decomposition* techniques have been presented in the literature [4, 5, 6]. These techniques represent the dependence vector set using a set of basic dependence vectors. With the help of these basic dependence vectors, the iteration space is 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. In this paper we present partitioning schemes which extract maximum parallelism from these 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 [3]. 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 halfspaces. We use the algorithm presented by Tzen and Ni [6] to construct this dependence convex hull. Every integer point in the convex hull corresponds to a dependence vector of the iteration space. If there are no integer points within the convex hull, then there are no cross-iteration dependences among the nested loop iterations. The corner points of this convex hull form the set of extreme points for the convex hull. These extreme points have the property that any point in the convex hull can be represented as a convex combination of these extreme points [7]. Since the extreme points of a dependence convex hull could be real and therefore are not valid iterations, we propose an algorithm (algorithm IDCH) to convert the dependence convex hull to an *integer dependence convex hull* with integer extreme points. The dependence vectors of the extreme points form a set of extreme vectors for the dependence vector set [8]. 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.

The rest of the paper is organized as follows. In section II, we introduce the program model considered and review the related work previously done on tiling. Dependence analysis for tiling is also presented. Section III discusses dependence convex hull computation. In section IV, with the help of theorems based on linear programming principles we propose simple methods to compute the minimum dependence distances and present minimum dependence distance tiling schemes. Finally in section V a comparative performance analysis with some of the existing methods is presented to demonstrate the effectiveness of our scheme.

2 PROGRAM MODEL

We consider nested loop program segments of the form shown in Figure 2. For the simplicity of notation and technique presentation we consider only tightly coupled doubly nested loops. However our method applies to multi-dimensional nested loops also. 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 dimensions of these arrays are assumed to be equal to the nested loop dimension. 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(f1(I,J),f2(I,J)) = ...
S_u:          ... = A(f3(I,J),f4(I,J))
        endfor
      endfor
```

Figure 2: *Program Model*

In our program model shown in Figure 2, 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 other types like flow and anti dependences.

An *iteration vector* $\vec{i}$ represents a set of statements that are executed for a specific value of (I,J) = (i, j). It can be represented mathematically by $\vec{i} = \{(i,j) \mid L_I \leq i \leq U_I, L_J \leq j \leq U_J; i, j \in Z\}$ where Z denotes the set of integers. For any given iteration vector $\vec{i}$ 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 $\vec{i}_1$ and $\vec{i}_2$ is defined as *cross-iteration dependence* and is represented by the dependence distance vector $\vec{d} = \vec{i}_2 - \vec{i}_1$. 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 $\vec{D} = \{\vec{d} \mid \vec{d} = \vec{i}_2 - \vec{i}_1; \vec{i}_1, \vec{i}_2 \in Z^2\}$.

If all the iteration vectors in the iteration space have the same set of 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 Review of Related Work

Irigoin and Triolet [5] 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 [9], they compute *extreme rays* for dependence cones and use *hyperplane* technique 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. We present simple schemes to compute minimum dependence distance and partition the iteration space into tiles.

Ramanujam and Sadayappan [4] 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.

Tzen and Ni [6] 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. In our work we extend their cross-iteration dependence analysis methods to compute more accurate dependence information by forming an integer dependence convex hull. We tile the iteration space by computing minimum dependence distances and use tile synchronization methods to synchronize parallel execution of tiles.

Based on integer programming techniques, Tseng et. al. [10] form a *minimum dependence vector* set. Using this minimum dependence vector set, the iterations are grouped. But the method they used to compute the minimum dependence vector set may not always give *minimum dependence distances*. We show how accurate minimum dependence distance can be computed using our approach. We have also observed that their grouping method does not work for some cases [8]. We present better tiling techniques which work with tile synchronization.

2.2 Dependence Analysis for Tiling

For the nested loop program segment shown in Figure 2, 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 [3] 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 [6]. Here, we consider only those cases for which we can express the general solution in terms of two integer variables. So, we have (i_1, j_1, i_2, j_2) as functions of 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. For every valid set of integers (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 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 $\Gamma = \{(x, y) \mid s_i(x, y) \ satisfies \ (1)\}$. 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 Γ and form a *convex polyhedron*, which can also be termed as *Dependence Convex Hull* (DCH) [6]. In the following section we give a brief introduction to convex set theory and explain how we apply the well known results of convex spaces to iteration spaces.

3 DEPENDENCE CONVEX HULL

To extract useful dependence 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 [6] 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.

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. 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 satements S_d and S_u in the program model. Otherwise, every integer point in this DCH represents a dependence vector in the iteration space.

3.2 Integer Dependence Convex Hull

We use the algorithm given by Tzen and Ni [6] 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) and cuts the solution space to form a bounded dependence convex hull. The extreme points of this convex hull can have real coordinates, because these points are just intersections of a set of hyperplanes. We propose an 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 a set of extreme vectors for the dependence vector set [8]. This information is otherwise not available for non-uniform dependence vector sets. We refer to the convex hull with all integer extreme points as *Integer Dependence Convex Hull* (IDCH). Our algorithm IDCH is given in Figure 3.

```
Algorithm IDCH:
 input: The Dependence Convex Hull (DCH);
 output: An Integer Dependence Convex Hull (IDCH);
 begin
  Initialize IDCH = DCH;
  Scan the IDCH ring
   if(coordinates of the node or node→next are real)
     1. Form a line equation next_line connecting node (r)
     and node→next (n);
     2. Find an integer point i closest to r in
     the direction towards n:
       insert i in the ring between r and n
       if i lies on the next_line
         if n is an integer point
           goto next node in the ring;
     3. Compute all the integer points along the boundary
     of the next_line but within the IDCH.
   endif
  Rescan the IDCH ring:
   remove any redundant and collinear nodes, maintaining
   the convex shape of the IDCH;
 end
```

Figure 3: *Algorithm to compute a two dimensional Integer Dependence Convex Hull*

The IDCH contains more accurate dependence information as explained later. 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. As the DCH is formed as a ring, for every node (real_node) there is a prevous node (prev_node) and a next node (next_node). Our algorithm traverses the IDCH once, converting all the real extreme points to integer form. A line (next_line) joining the real_node and next_node is formed. Now a nearest integer point in the direction of this line but within the convex hull is computed. This node is inserted into the ring. Similarly, all the integer points that are along the boundary of the line next_line and within the convex hull are computed. Once all the extreme points are converted to integer points, our algorithm traverses the IDCH again to remove any redundant, collinear nodes from the ring. While doing this, our algorithm preserves the convex shape of the IDCH. We have a simple algorithm to compute the integer points [8]. The worst case complexity of this algorithm is bounded by $O(\mathcal{N})$ where $\mathcal{N}$ is the number of integer points along the perimeter 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.

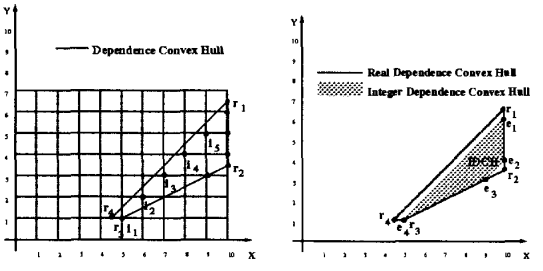
We can demonstrate the construction of the DCH and IDCH with an example. Consider example 1(b) whose iteration space is shown in Figure 1(b). Two diophantine equations can be formed from the subscripts of array A.

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

By applying the algorithm given by Banerjee [3], 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 $D(x,y) = (-2x+2y+4, 2x-3y-6)$. Now, the set of inequalities can be given as

$$\begin{aligned} 1 \leq \quad & x & \leq 10 \\ 1 \leq \quad & y & \leq 10 \\ 1 \leq \quad & -x+2y+4 & \leq 10 \\ 1 \leq \quad & 2x-2y-6 & \leq 10 \end{aligned} \quad (6)$$

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 by scanning the DCH ring. For a real extreme point, as explained previously, it forms a line next_line. For example, consider the node $\mathbf{r}_4$=(4.5,1). The node $\mathbf{r}_1$ is $\mathbf{r}_4$'s next_node. So, a next line joining $\mathbf{r}_4$ and $\mathbf{r}_1$ is formed. As shown in Figure 4(a) the integer point closest to $\mathbf{r}_4$ and along the next_line is $\mathbf{i}_1$. This node is inserted into the ring between the real_node and the next_node. Similarly points $\mathbf{i}_2$, $\mathbf{i}_3$, $\mathbf{i}_4$ and $\mathbf{i}_5$ are computed. Same steps are repeated for the real node $\mathbf{r}_1$ whose next_node is $\mathbf{r}_2$. Once all the real extreme points are converted to integer points, we can eliminate redundant or collinear nodes from the IDCH ring. For example, nodes $\mathbf{i}_1$, $\mathbf{i}_2$, $\mathbf{i}_3$, and $\mathbf{i}_4$ 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.



(a) (b)

Figure 4: (a) IDCH *Computation* (b) DCH *and* IDCH *for Example* 1(b)

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 dependence information is lost while changing the DCH to IDCH [8]. In the following section we demonstrate how these extreme points 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 $d_i(x,y) = a_1x+b_1y+c_1$; $d_j(x,y) = a_2x+b_2y+c_2$ 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*" [7, 11]. 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 1 : *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 $ax+by+c = k$ passes 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 5. So, the function $d(x,y)$ assumes its maximum and minimum values at the extreme points. □

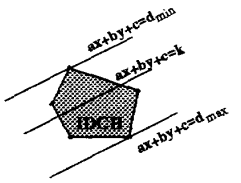


Figure 5: *Minimum and maximum values of* $d(x, y)$

Hence, we can find the minimum dependence distance from the extreme vector list. 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.

Theorem 2 *: 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 6(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 6(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. □

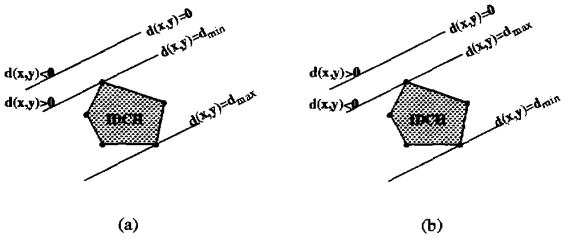


Figure 6: *Computation of abs(min) and abs(max) values of* $d(x, y)$ *when (a)* IDCH $\in d(x, y) > 0$ *(b)* IDCH $\in d(x, y) < 0$

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.

4.2 Tiling and Tile Synchronization

In this subsection, we show how to identify partitions (tiles) of the iteration space. 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 methods. 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 subtiles without loss of any parallelism.

We can now apply this method to the nested loop program segment given in example 1(b). It's 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. 7. 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 8. This parallel code applies to any nested loop segment that satisfies case 1 and of the form as given in 2 with $L_I = 1$, $U_I = N$, $L_J = 1$, $U_J = M$.

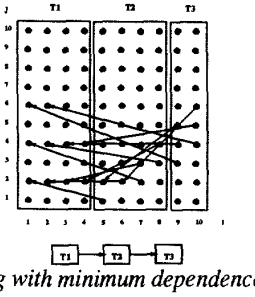


Figure 7: *Tiling with minimum dependence distance* d_i

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).

```
Tile num T_n = ⌈N / d_imin⌉

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 8: *Parallel code for scheme 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 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 [6] can be used to synchronize the execution of inter-group tiles.

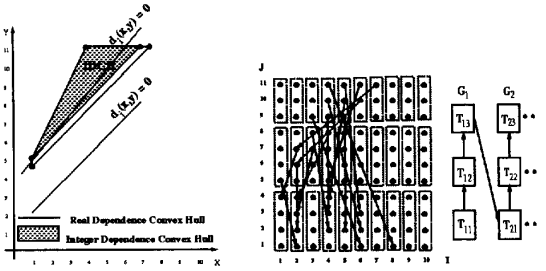
Tzen and Ni [6] presented a number of lemmas and theorems to find the maximum and minimum values of the *Dependence Slope Function* defined as DSF = $\frac{d_j(x,y)}{d_i(x,y)}$. These minimum or maximum dependence slopes can be used to enforce the dependence constraints among the iterations. The execution 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 9(a) 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 9(b).

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
```

For this example, we can find the minimum dependence slope to be $-min(M-1, P)$, 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 can be executed as soon as $(i - T_s)^{th}$ tile in group $j-1$ is executed. Figure 9(b) shows the tile space graph for this example. In this figure G_i denotes a group and T_{ij} denotes j^th tile of group i. Parallel code for this example is given in Figure 10. Speedup for this case can be computed as follows. The total serial execution time is $M * N$. Since the parallel execution time is $T_n + (N-1) * T_s$. Hence, the speedup is given as

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



(a) (b)

Figure 9: *(a) IDCH of Example 2 (b) Tiling with minimum dependence distance* d_j

Case III: $d_i(x,y)$=0 *and* $d_j(x,y)$=0 *pass through the* IDCH

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 10 is also valid for this case. For this case our technique performs as good as the dependence uniformization technique.

In the next section, we compare the performance of our technique with existing techniques and analyze the improvement in speedup.

```
Tile num T_n = ⌈M / d_imin⌉
Tile slope T_s = ⌈DSF_min / d_jmin⌉

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 10: *Parallel code for scheme 2*

5 PERFORMANCE ANALYSIS

Many existing techniques cannot parallelize the type of nested loops considered in this paper because of the irregularity of the de-

pendences. Though some advances have been made to solve this problem, the amount of parallelism that the existing techniques extract is very low compared to the available parallelism. In the previous sections we have presented simple schemes to compute minimum dependence distance and used it to partition the nested loop iteration spaces with irregular dependences. Now, we compare our algorithms with some existing methods.

The dependence uniformization method presented by Tzen and Ni [6] 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 iteration space is uniformized and with index synchronization, 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. Figure 7 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 the dependence uniformization technique. Even for the case d_{imin}=1 our technique gives a speedup of 10 (M) which is much higher compared to the speedup with their technique. An important feature of our method is that the speedup does not depend on the range of dependence slopes.

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. Moreover the IDCH formed by our method gives more precise dependence slope information.

Though the minimum dependence vector set of Tseng et.al., [10] can be used to form somewhat similar rectangular partitions, their grouping techniques do not consider all the cases. Also, the method they used to compute the minimum dependence vector set may not always give minimum dependence distances. Moreover, they use integer programming techniques to compute the minimum dependence vector set which definitely is time consuming. Their method does not work for certain cases [8].

6 CONCLUSION

In this paper we have presented simple and computationally efficient 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 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. Pseudo code for parallel execution of the tiles is given. We have shown that our method gives much better speedup than the existing techniques and exploits the inherent parallelism in the nested loops with non-uniform dependences.

ACKNOWLEDGEMENTS

We would like to thank members of our Parallel and Distributed Computing Laboratory for their useful suggestions.

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

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

[4] 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.

[5] 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.

[6] 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.

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

[8] S. Punyamurtula and V. Chaudhary, "On tiling nested loop iteration spaces with irregualr dependence vectors," Tech. Rep. TR-94-02-22, Parallel and Distributed Computing Laboratory, Wayne State University, Detroit, Mar 1994.

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

[10] S.-Y. Tseng, C.-T. King, and C.-Y. Tang, "Minimum dependence vector set: A new compiler technique for enhancing loop parallelism," in *Proceedings of 1992 International Conference on Parallel and Distributed Systems*, (Hsinchu, Taiwan, R.O.C.), p. 340 to 346, Dec 1992.

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