Unique Sets Oriented Parallelization of Loops with Non-uniform Dependences

JIALIN JU AND VIPIN CHAUDHARY

Parallel and Distributed Computing Laboratory, Wayne State University, Detroit, MI 48202, USA
Email: vipin@eng.wayne.edu

Although many methods exist for nested loop partitioning, most of them perform poorly when parallelizing loops with non-uniform dependences. This paper addresses the issue of automatic parallelization of loops with non-uniform dependences. Such loops normally are not parallelized by existing parallelizing compilers and transformations. Even when parallelized in rare instances, the performance is very poor. Our approach is based on the Convex Hull theory which has adequate information to handle non-uniform dependences. We introduce the concept of Complete Dependence Convex Hull, Unique Head and Tail Sets and abstract the dependence information into these sets. These sets form the basis of the iteration space partitions. The properties of the unique head and tail sets are derived. Depending on the relative placement of these unique sets, partitioning schemes are suggested for implementation of our technique. Implementation results of our scheme on the Cray J916 and comparison with other schemes show the superiority of our technique.

Received November 4, 1996; revised July 15, 1997

1. INTRODUCTION

Given a sequential program, a challenging problem for parallelizing compilers is to detect maximum parallelism. It is generally agreed upon and shown in the study by Kuck et. al. 1 that most of the computation time is spent in loops. Current parallelizing compilers concentrate on *loop parallelization* 2. A loop can be easily parallelized if there are no *cross-iteration dependences*. However, loops with cross-iteration dependences are very common. Parallelizing loops with cross-iteration dependences is a major concern facing parallelizing compilers today.

Loops with cross-iteration dependences can be roughly divided into two groups. One is loops with static regular dependences, which can be analyzed during compile time. *Example 1, 2* in Figure 1 belong to this group. The other group is loops with dynamic irregular dependences, which have indirect access patterns. *Example 3* shows a typical irregular loop, which is used for edge-oriented representation of sparse matrices. These kind of loops cannot be parallelized at compile time, for lack of sufficient information. To execute such loop efficiently in parallel, runtime support must be provided. The major job of parallelizing compilers is to parallelize loops with static regular dependences.

Static regular loops can be further divided into two sub-groups. One is with uniform dependences and the other is with non-uniform dependences. The dependences are uniform only when the patterns of dependence vectors are uniform. In other words, the dependence vectors can be expressed by constants, *i.e., distance vectors*. *Example 1* illustrates a uniform dependence loop. Its dependence vectors are (1, 0) and (1, -1). Figure 2 (a) shows the dependence patterns of *Example 1* in the iteration space. In the same fashion, we call some dependences non-uniform when dependence vectors are in irregular patterns which cannot be expressed by *distance vectors*. Figure 2 (b) shows the dependence patterns of *Example 2* in the iteration space.

A lot of research has been done in parallelizing loops with uniform dependences, from dependence analysis to loop transformation, such as *loop interchange, loop permutation, skew, reversal, wavefront, tiling*, etc. But little research been done for the loops with non-uniform dependences.

The existing commercial parallelizing compilers and research parallelizing compilers, such as Stanford's SUIF 3, CSRD's Parafrase-2 4, and University of Maryland's Omega Project 5, can parallelize most of the loops with uniform dependences. But they do not satisfactorily handle loops with non-uniform dependences. Most of the time, the compiler treats such loops as un-

Example 1:
```
do i = 1, 12
   do j = 1, 12
      A(i + 1, j) = ···
      ··· = A(i, j) + A(i, j + 1)
   enddo
enddo
```

Example 2:
```
do i = 1, 12
   do j = 1, 12
      A(2i + 3, j + 1) = ···
      ··· = A(2j + i + 1, i + j + 3)
   enddo
enddo
```

Example 3:
```
do i = 1, 12
   do j = 1, 12
      A(B(i), C(j)) = ···
      ··· = A(B(i − 2), C(j + 5))
   enddo
enddo
```

FIGURE 1. *Examples of loops with different kinds of dependences*

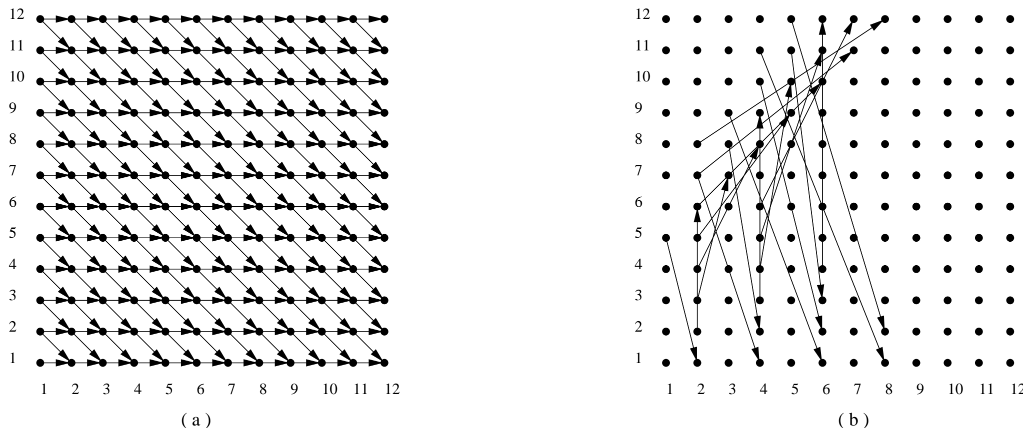


FIGURE 2. *Iteration spaces with (a) Uniform dependences and (b) Non-uniform dependences*

parallelizable and leaves them running sequentially. For instance, neither SUIF nor Parafrase-2 can parallelize the loop in *Example 2*. Unfortunately, loops with non-uniform dependences are not so uncommon in the real world. In an empirical study, Shen *et al.* 6 observed that nearly 45% of two dimensional array references are coupled, which means array subscripts are linear combinations of loop indices. These coupled subscripts lead to non-uniform dependence. Hence, it is imperative to give loops with non-uniform dependence a serious consideration, even though they are more difficult to parallelize.

This paper focuses on parallelization of perfectly nested loops with non-uniform dependences. The rest of this paper is organized as follows. Section two surveys the research in parallelization of non-uniform dependence loops. Section three reviews the Dependence Convex Hull theory and introduces the Complete Dependence Convex Hull. Section four gives the definition of unique sets and the techniques to find them. Section five presents our unique set oriented partitioning approach. Section six extends our technique to a general program model with multiple nestings. Section seven confirms the superiority of our technique with an implementation on Cray J916 and comparison with previously proposed techniques. Finally, we conclude in section eight.

2. SURVEY OF RELATED RESEARCH

The convex hull created by solving the linear Diophantine equations is required for detecting parallelism in non-uniform loops since it is the least abstraction to have adequate information to accomplish the detection of parallelism in non-uniform loops 7. Thus, most of the techniques proposed for parallelizing loops with non-uniform dependences are based on dependence convex hull theory. These can be classified into four categories: uniformization, uniform partitioning, non-uniform partitioning, and integer programming based partitioning.

2.1. Uniformization

Tzen and Ni8 proposed the *dependence uniformization* technique. Based on solving a system of Diophantine equations and a system of inequalities, they compute the maximal and minimal dependence slopes of any uniform and non-uniform dependence pattern in a two-dimensional iteration space. Then, by applying the idea of vector decomposition, a set of basic dependences is chosen to replace all original dependence constraints in every iteration so that the dependence pattern becomes uniform. They also proved that any doubly nested loop could always be uniformized to a uniform dependence loop with two dependence vectors. They proposed an *index synchronization* method to reduce the synchronization, in which synchronization could be systematically inserted. 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.

Chen and Yew9 presented a scheme which computes a *Basic Dependence Vector Set* and schedules the iterations using *Static Strip Scheduling*. They extended the dependence uniformization technique of Tzen and Ni8 and presented algorithms to compute better basic dependence vector sets which extract more parallelism from the nested loops. The program model is more general, including non-perfect nested loops. While this technique is definitely an improvement over Tzen and Ni's work, it also imposes too many dependences on the iteration space, thereby reducing the extractable parallelism. Moreover, this uniformization needs a lot of synchronization.

Chen and Shang10 proposed another uniformization technique. They form the set of basic dependence vectors and improve this set using certain objective functions. They select those basic dependence vectors which are time-optimal and cone-optimal. After uniformizing the iteration space, they use optimal linear schedules11 to order the execution of the iterations. This technique like both the previous uniformization techniques impose too many dependences.

2.2. Uniform Partitioning

Punyamurtula and Chaudhary12 extended the theory of Convex Hull to the *Integer Dependence Convex Hull*(IDCH) and proposed a *Minimum Dependence Distance Tiling* technique. Every integer point in the IDCH corresponds to a dependence vector in the iteration space of the nested loops. They showed that the minimum and maximum values of the dependence distance function occur at the extreme points of the IDCH. Therefore, it is only necessary to calculate the dependence distance at the extreme points and compare all the values of the distance to get the minimum dependence distance. These minimum dependence distances are used to partition the iteration space into tiles of uniform size and shape. The width of tiles is less than or equal to the minimum dependence distance in at least one direction. This would guarantee that for any dependence vector, its head and tail would fall into different tiles. Iterations in a tile would be executed in parallel. Tiles in a group would be executed in sequence and the dependence slope information of Tzen and Ni8 can be used to synchronize the execution of inter-group tiles. This technique works very well for cases when the minimum distance in one direction is large. It does not work as well for the case when the dependence distances are small as it would involve too much synchronization overhead.

2.3. Non-uniform Partitioning

Zaafrani and Ito13 proposed the three-region technique. This technique divides the iteration space into two parallel regions and one sequential region. The iterations in the parallel regions can be executed fully in parallel while the iterations in the sequential region can only be executed sequentially. Two parallel regions are called $Area1$ and $Area2$, respectively, and the sequential region is called $Area3$. $Area1$ represents the part of the iteration space where the destination iteration comes lexically before the source iteration. The iterations in $Area1$ can be fully executed in parallel provided that variable renaming is performed. $Area1$ corresponds to the region where the direction vector is equal to $(<, *)$ or equal to $(=, <)$. $Area2$ represents the part of the iteration space where the destination iteration comes lexically after the source iteration and the source iteration is in $Area1$. If $Area1$ is executed first, then the nodes in $Area2$ can be executed in parallel. $Area3$ represents the rest of the iteration space (iteration space - $(Area1 \cup Area2)$). Once $Area1$ and $Area2$ are executed, then the nodes in $Area3$ should be executed sequentially. Zaafrani and Ito apply their technique to the entire iteration space, though it will suffice to applying it only to the DCH or IDCH. The nodes that are not in the DCH can be executed in parallel because of the nonexistence of dependences for these nodes. This is equivalent to dividing the iteration space into four regions ($Area1$, $Area2$, $Area3$, and non-DCH). Again this technique has its disadvantages. The sequential part of the iteration space is the bottleneck for the performance. If the sequential part of iteration space is small, this technique is fine. Otherwise the sequential part can be a serious drawback in performance.

2.4. Integer Programming Based Approach

Tseng et. al.14 proposed a partitioning scheme using Integer Programming techniques. They start with an original dependence vector set and divide it into eight groups. They find the minimum dependence vector set by solving integer programming formulations. Then they use minimum dependence vector set to represent the dependence vectors of nested loops and partition the iterations of loops into groups. All iterations in the same group can be executed at the same time. They also proposed a group synchronization method for arranging synchronization. But the method they used to compute the minimum dependence vector set may not always give minimum dependence distances. Besides, integer programming approach is time-consuming.

Pugh and Wonnacott 15 construct several sets of constraints that describe, for each statement, which iterations of that statement can be executed concurrently. By constructing constraints that correspond to different assumptions about which dependences might be eliminated through additional analysis, transformations, and user assertions, they determine whether they can expose parallelism by elimination dependences. Then they look for conditional parallelism, and try to identify the kinds of iteration-reordering transformations that could be used to produce parallel loops. However, their method

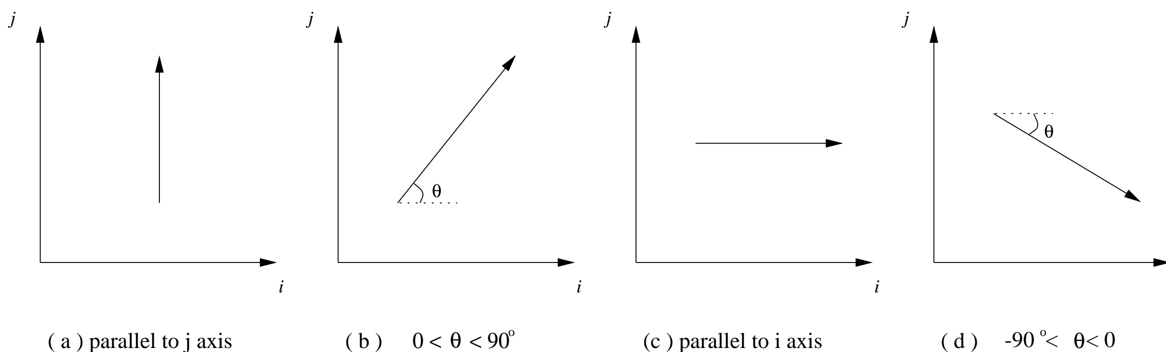


FIGURE 3. *Possible dependence directions in lexicographic order*

may produce false dependences.

3. DEPENDENCE ANALYSIS

Cross-iteration dependence is the major concern that may keep the program from running in parallel. For the four types of data dependences, *flow, anti, output,* and *input* dependence, *input* dependence imposes no ordering constraints, so we only look at the other three types. We won't consider output dependences as real dependences either. We can always use the storage replication technique to allow the statements which have output dependences to execute concurrently. This research will look at the cases of flow dependences and anti dependences.

Data dependence defines the execution order among iterations. The execution order can be expressed as *Lexicographic order*. Lexicographic order can be shown as an arrow in the iteration space, which also represents the dependence vector. All the arrows in Figure 2 are in lexicographic order. The iteration corresponding to the arrow head cannot be executed until the iteration corresponding to the tail has been executed. All the dependences discussed in this paper are put into lexicographic order. If there is a dependence from iteration **i** to iteration **j**, and **i** executes before **j**, we represent it by drawing an arrow $\mathbf{i} \rightarrow \mathbf{j}$.

Figure 3 shows all four possible directions if all the dependence vectors are put in lexicographic order with two level of loops, where i is the index for the outer loop and j is the index for the inner loop. The running order imposes that there cannot exist an arrow pointing to the left or an arrow parallel to j axis and pointing down. The arrows here are the dependence vectors.

3.1. Dependence and Convex Hull

Studies16, 6 show that most of the loops with complex array subscripts are two dimensional loops. We start with this typical case. We simplify our general program model to a normalized, doubly nested loop with coupled subscripts (*i.e.*, with subscripts being linear functions of loop indices) as shown in figure 4.

We wish to discover what cross-iteration dependences

do $i = L_1, U_1$
  do $j = L_2, U_2$
    $A(a_{11} * i + b_{11} * j + c_{11}, a_{12} * i + b_{12} * j + c_{12}) = \cdots$
    $\cdots = A(a_{21} * i + b_{21} * j + c_{21}, a_{22} * i + b_{22} * j + c_{22})$
  enddo
enddo

FIGURE 4. *Doubly Nested Loop Model*

exist between the two references to array A in the program model. There are a large variety of tests that can prove independence in some cases. It is infeasible to solve the problem directly, even for linear subscript expressions, because finding dependences is equivalent to the NP-complete problem of finding integer solutions to systems of linear Diophantine equations17. Two general and approximate tests are GCD18 and Banerjee's inequalities19. Recently, Subhlok and Kennedy 20 proposed a new search procedure that identifies an integer solution in a convex region, or prove that no integer solutions exist.

The most common methods to compute data dependence is to solve a set of linear Diophantine equations with a set of constraints which are the iteration boundaries. A dependence exists only if the equations have a solution.

We want to find a set of integer solutions (i_1, j_1, i_2, j_2) that satisfy the system of Diophantine equations (1) and the system of linear inequalities (2) .

$$\begin{aligned} a_{11}i_1 + b_{11}j_1 + c_{11} &= a_{21}i_2 + b_{21}j_2 + c_{21} \\ a_{12}i_1 + b_{12}j_1 + c_{12} &= a_{22}i_2 + b_{22}j_2 + c_{22} \end{aligned} \tag{1}$$

$$\begin{cases} L_1 \le i_1 \le U_1 \\ L_2 \le j_1 \le U_2 \\ L_1 \le i_2 \le U_1 \\ L_2 \le j_2 \le U_2 \end{cases} \tag{2}$$

Once the general solutions are found, dependence information can be represented by dependence vector. The dependence is uniform when dependence vectors are constants. Otherwise the dependence is non-uniform.

The data dependence analysis techniques do well on loops with uniform dependences since dependence *distance vectors* can be calculated precisely. A lot of research has been done for uniform dependence analysis and loop transformation techniques 21, 22, 23, 24. However, for the case of non-uniform dependences, Yang, Ancourt and Irigoin7 showed that direction vector alone does not have enough information for transforming non-uniform dependence. *Dependence Convex Hull* (DCH) 8 is the least requirement if we want to parallelize loops with non-uniform dependence. DCHs are convex polyhedrons and are subspace of the solution space. First of all, we show how to find DCHs.

There are two approaches to solve the system of Diophantine equations of (1). One way is to set i_1 to x_1 and j_1 to y_1 and get the solution to i_2 and j_2.

$$\begin{cases} a_{21}i_2 + b_{21}j_2 + c_{21} = a_{11}x_1 + b_{11}y_1 + c_{11} \\ a_{22}i_2 + b_{22}j_2 + c_{22} = a_{12}x_1 + b_{12}y_1 + c_{12} \end{cases}$$

We have the solution as

$$\begin{cases} i_2 = \alpha_{11}x_1 + \beta_{11}y_1 + \gamma_{11} \\ j_2 = \alpha_{12}x_1 + \beta_{12}y_1 + \gamma_{12} \end{cases}$$

where

$$\alpha_{11} = \frac{a_{11}b_{22} - a_{12}b_{21}}{a_{21}b_{22} - a_{22}b_{21}} \qquad \beta_{11} = \frac{b_{11}b_{22} - b_{12}b_{21}}{a_{21}b_{22} - a_{22}b_{21}}$$

$$\gamma_{11} = \frac{b_{22}c_{11} + b_{21}c_{22} - b_{22}c_{21} - b_{21}c_{12}}{a_{21}b_{22} - a_{22}b_{21}}$$

$$\alpha_{12} = \frac{a_{21}a_{12} - a_{11}b_{22}}{a_{21}b_{22} - a_{22}b_{21}} \qquad \beta_{12} = \frac{a_{21}b_{12} - a_{22}b_{11}}{a_{21}b_{22} - a_{22}b_{21}}$$

$$\gamma_{12} = \frac{a_{21}c_{12} + a_{22}c_{21} - a_{21}c_{22} - a_{22}c_{11}}{a_{21}b_{22} - a_{22}b_{21}}$$

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

$$\begin{cases} L_1 \le & x_1 & \le U_1 \\ L_2 \le & y_1 & \le U_2 \\ L_1 \le & \alpha_{11}x_1 + \beta_{11}y_1 + \gamma_{11} & \le U_1 \\ L_2 \le & \alpha_{12}x_1 + \beta_{12}y_1 + \gamma_{12} & \le U_2 \end{cases} \qquad (3)$$

where (3) defines a DCH denoted by **DCH1**.

Another approach is to set i_2 to x_2 and j_2 to y_2 and solve for the solution to i_1 and j_1.

$$\begin{cases} a_{11}i_1 + b_{11}j_1 + c_{11} = a_{21}x_2 + b_{21}y_2 + c_{21} \\ a_{12}i_1 + b_{12}j_1 + c_{12} = a_{22}x_2 + b_{22}y_2 + c_{22} \end{cases}$$

We have the solution as

$$\begin{cases} i_1 = \alpha_{21}x_2 + \beta_{21}y_2 + \gamma_{21} \\ j_1 = \alpha_{22}x_2 + \beta_{22}y_2 + \gamma_{22} \end{cases}$$

where

$$\alpha_{21} = \frac{a_{21}b_{12} - a_{22}b_{11}}{a_{11}b_{12} - a_{12}b_{11}} \qquad \beta_{21} = \frac{b_{12}b_{21} - b_{11}b_{22}}{a_{11}b_{12} - a_{12}b_{11}}$$

$$\gamma_{21} = \frac{b_{12}c_{21} + b_{11}c_{12} - b_{12}c_{11} - b_{11}c_{22}}{a_{11}b_{12} - a_{12}b_{11}}$$

$$\alpha_{22} = \frac{a_{11}a_{22} - a_{12}b_{21}}{a_{11}b_{12} - a_{12}b_{11}} \qquad \beta_{22} = \frac{a_{11}b_{22} - a_{12}b_{21}}{a_{11}b_{12} - a_{12}b_{11}}$$

$$\gamma_{22} = \frac{a_{11}c_{22} + a_{12}c_{11} - a_{11}c_{12} - a_{12}c_{21}}{a_{11}b_{12} - a_{12}b_{11}}$$

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

$$\begin{cases} L_1 \le & \alpha_{21}x_2 + \beta_{21}y_2 + \gamma_{21} & \le U_1 \\ L_2 \le & \alpha_{22}x_2 + \beta_{22}y_2 + \gamma_{22} & \le U_2 \\ L_1 \le & x_2 & \le U_1 \\ L_2 \le & y_2 & \le U_2 \end{cases} \qquad (4)$$

where (4) defines another DCH, denoted by **DCH2**.

Both sets of solutions are valid. Each of them has the dependence information on one extreme. For some simple cases, for instance, there is only one kind of dependence, either flow or anti dependence, one set of solutions(*i.e.* DCH) should be enough. Punyamurtula and Chaudhary used constraints (3) for their technique 12, while Zaafrani and Ito used (4) for their technique 13. For those more complicated cases, where both flow and anti dependences are involved and dependence patterns are irregular, we need to use both sets of solutions. We will introduce a new term *Complete Dependence Convex Hull* to summarize these two DCHs and we demonstrate that the Complete DCH contains complete information about dependences.

3.2. Complete Dependence Convex Hull (CDCH)

DEFINITION 3.1 (COMPLETE DCH (CDCH)). Complete DCH is the union of two closed sets of integer points in the iteration space, which satisfy (3) or (4).

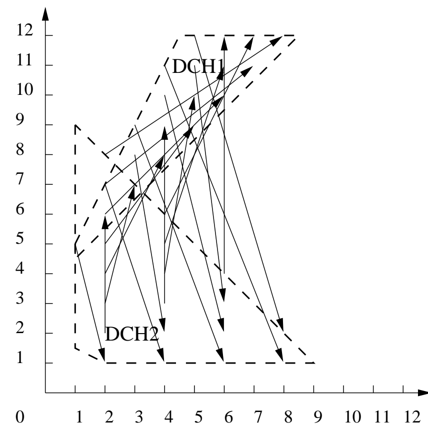


FIGURE 5. *CDCH of Example 2*

Figure 5 shows the CDCH of *Example 2*. We use an arrow to represent a dependence in the iteration space. We call the arrow's head *the dependence head* and the arrow's tail *the dependence tail*.

THEOREM 3.1. All the dependence heads and tails lie within the CDCH. The head and tail of any particular dependence lie in the two DCHs of the CDCH.

Proof. Let us assume that (i_2, j_2) is dependent on (i_1, j_1). In the iteration space graph we can have an arrow from (i_1, j_1) to (i_2, j_2). Here (i_1, j_1) is the arrow tail and (i_2, j_2) is the arrow head. Because of the existing dependence, (i_1, j_1) and (i_2, j_2) must satisfy the system of linear Diophantine equations (1) and the system of linear inequalities (2). There are four unknown variables. We can reduce two unknown variables by setting $i_1 = x$ and $j_1 = y$ and solve for i_2 and j_2. Then i_1 and j_1 must satisfy (3). Hence (i_1, j_1) lies in the area defined by (3) which is one of the DCH of the CDCH. In the same way, we reduce i_1 and j_1 by setting $i_2 = x$ and $j_2 = y$ and solve for i_1 and j_1. Here (i_2, j_2) lies in the area defined by (4) which is another DCH of the CDCH. Therefore, both (i_1, j_1) and (i_2, j_2) fall into different DCHs of the CDCH. 2

If iteration (i_2, j_2) is dependent on (i_1, j_1), then dependence vector D(x, y) is expressed as:

$$d_i(x, y) = i_2 - i_1$$
$$d_j(x, y) = j_2 - j_1$$

So, for DCH1, we have

$$\begin{aligned} d_i(x_1, y_1) &= (\alpha_{11} - 1)x_1 + \beta_{11}y_1 + \gamma_{11} \\ d_j(x_1, y_1) &= \alpha_{12}x_1 + (\beta_{12} - 1)y_1 + \gamma_{12} \end{aligned} \quad (5)$$

For DCH2, we have

$$\begin{aligned} d_i(x_2, y_2) &== (1 - \alpha_{21})x_2 - \beta_{21}y_2 - \gamma_{21} \\ d_j(x_2, y_2) &= -\alpha_{22}x_2 + (1 - \beta_{22})y_2 - \gamma_{22} \end{aligned} \quad (6)$$

Clearly if there is a solution (x_1, y_1) in DCH1, there must be a solution (x_2, y_2) in DCH2, because they have been solved from the same set of linear Diophantine equations (1).

Given the dependence vectors above, there must exist a minimum and a maximum value of $D(x, y)$. It was shown by Punyamurtula and Chaudhary 12 that the minimum and maximum values of the dependence $D(x, y)$ occur at the extreme points of the DCH.

4. UNIQUE SETS IN THE ITERATION SPACE

If a loop has cross-iteration dependences, we can construct its CDCH (comprising of DCH1 and DCH2). As we have proved earlier, all dependences lie within the CDCH. In other words, the iterations lying outside the CDCH can be executed in parallel. Punyamurtula and Chaudhary proposed the concept of minimum dependence distance tiling 12, which gives an excellent partitioning of iteration space for the case when $\vec{d}(x, y) = \vec{0}$ does not pass through any DCH. However, minimum dependence distance cannot be calculated when $\vec{d}(x, y) = \vec{0}$ passes through the DCH. Our technique works well for both the cases.

Suppose all dependence tails fall into DCH1 and all dependence heads fall into DCH2 (Figure 6(a)) and the two DCHs do not overlap. Partition can be done by drawing a line between the two DCHs. The area containing the DCH of tail will execute first followed by the area containing the DCH of heads. Figure 6(b) illustrates this fact by first executing area 1 followed by area 2. The iterations within the two areas are fully parallelizable.

The idea behind the above example is to find separate sets that contain the dependence heads and tails. We want to minimize these sets and then partition the iteration space by drawing lines separating these sets in the iteration space. The execution order is determined by whether the set contains heads or tails.

The next problem how is to find unique sets. The problem is compounded if these sets overlap.

4.1. Unique Head and Unique Tail Sets

There are only two DCHs given the program model in Figure 4. All the dependence heads and tails will lie within these two DCHs. These areas are our primitive sets. For one particular set, it is quite possible that it contains both the dependence heads and tails. Because of the complexity of the problem, we have to

- distinguish between the flow and anti dependences, and
- partition the iteration space in a non-uniform way because the dependence itself is non-uniform.

Let us look at Figure 5 which shows the CDCH of *Example 2*. We note that DCH1 contains all anti dependence heads and all flow dependence tails. DCH2 contains all the flow dependence heads and anti dependence's tails. Figure 7 separates the flow and anti dependences to give a clearer picture. It can be found out that DCH1 is the union of flow dependence tail set and anti dependence head set, and DCH2 is the union of flow dependence head set and anti dependence tail set. Hence, the following definition is derived to distinguish the sets.

DEFINITION 4.1 (UNIQUE HEAD(TAIL) SET). Unique head(tail) set is a set of integer points in the iteration space that satisfies the following conditions:

1. it is subset of one of the DCH (or is the DCH itself).
2. it contains all the dependence arrow's heads(tails), but does not contain any other dependence arrow's tails(heads).

Obviously the DCHs in Figure 7 are not the unique sets we are trying to find, because each DCH contains

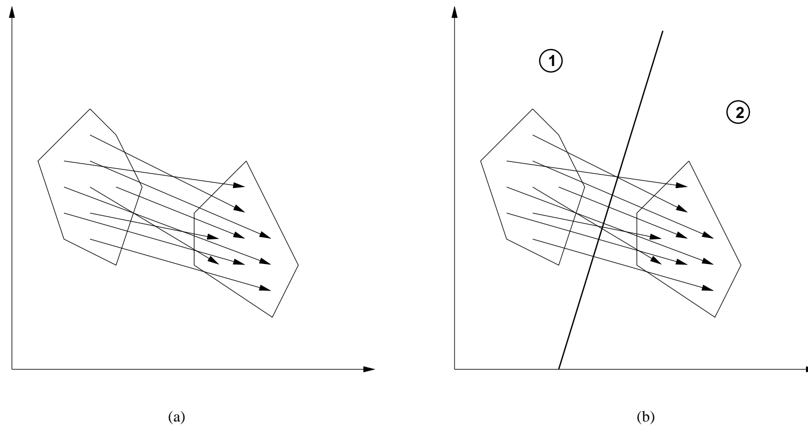


FIGURE 6. *Partitioning with two non-overlapping DCHs*

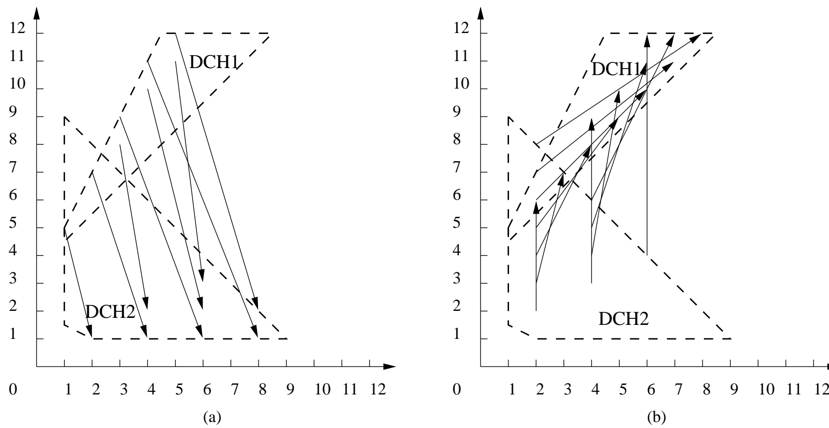


FIGURE 7. *(a) Flow dependence, (b) Anti dependence*

the dependence heads of one kind and the dependence tails of the other kind. Therefore, these DCHs must be further partitioned into smaller unique sets.

4.2. Finding Unique Head and Unique Tail Sets

First properties of DCH1 and DCH2 must be examined.

THEOREM 4.1. DCH1 contains all flow dependence tails and all anti dependence heads (if they exist) and DCH2 contains all anti dependence tails and all flow dependence heads (if they exist).

Proof. The system of inequalities in (3) defines DCH1 and

$$\begin{aligned} i_1 &= x_1 \\ j_1 &= y_1 \\ i_2 &= \alpha_{11}x_1 + \beta_{11}y_1 + \gamma_{11} \\ j_2 &= \alpha_{12}x_1 + \beta_{12}y_1 + \gamma_{12} \end{aligned}$$

If there exists a flow dependence, we can assume that (i_1, j_1, i_2, j_2) is a solution to the flow dependence. From the definition of flow dependence, (i_1, j_1) should be written somewhere in the iteration space before (i_2, j_2) is referenced. So we can draw an arrow from (i_1, j_1) to (i_2, j_2) in the iteration space to represent the dependence and execution order as $(i_1, j_1) \rightarrow (i_2, j_2)$ which is equivalent to $(x_1, y_1) \rightarrow (\alpha_{11}x_1 + \beta_{11}y_1 + \gamma_{11}, \alpha_{12}x_1 + \beta_{12}y_1 + \gamma_{12})$. Here (x_1, y_1) is the arrow tail. Since (x_1, y_1) satisfies (3) and we have assumed that (i_1, j_1, i_2, j_2) is a solution, DCH1 must contains all flow dependence tails.

If there exists an anti dependence, we can again assume that (i_1, j_1, i_2, j_2) is a solution to the anti dependence. From the definition of anti dependence, we have an arrow from (i_2, j_2) to (i_1, j_1), *i.e.*, $(\alpha_{11}x_1 + \beta_{11}y_1 + \gamma_{11}, \alpha_{12}x_1 + \beta_{12}y_1 + \gamma_{12}) \rightarrow (x_1, y_1)$. Since (x_1, y_1) is the arrow's head and (x_1, y_1) satisfies (3), DCH1 contains all anti dependence heads.

The proof that DCH2 contains all anti dependence tails and flow dependence heads (if they exist) is similar to the proof for DCH1. 2

The above theorem tells us that DCH1 and DCH2 are not unique head or unique tail sets if there are both flow and anti dependences. If there exist only flow or anti dependence, DCH1 either contains all the flow dependence tails or anti dependence heads, and DCH2 either contains all the flow dependence heads or anti dependence tails. Under these conditions, both DCH1 and

DCH2 are unique sets. The following theorem states the condition for DCH1 and DCH2 to be unique sets.

THEOREM 4.2. If $d_i(x, y) = 0$ does not pass through any DCH, then there is only one kind of dependence, either flow or anti dependence, and the DCH itself is the unique head set or the unique tail set.

Proof. [**Part 1**] $d_i(x_1, y_1)$ corresponds to DCH1 and $d_i(x_2, y_2)$ corresponds to DCH2. Suppose $d_i(x_1, y_1)$ does not pass through DCH1. Since $d_i(x_1, y_1) = i_2 - i_1 = (\alpha_{11} - 1)x_1 + \beta_{11}y_1 + \gamma_{11}$ and the iteration (x_1, y_1) that satisfies (3) must not satisfy $(\alpha_{11} - 1)x_1 + \beta_{11}y_1 + \gamma_{11} = 0$ ($d_i(x_1, y_1) = 0$ is a line in the iteration space), DCH1 must be on one side of $d_i(x_1, y_1) = 0$, *i.e.*, either $d_i(x_1, y_1) < 0$ or $d_i(x_1, y_1) > 0$. First let us look at the case when $d_i(x_1, y_1) < 0$. If $d_i(x_1, y_1) < 0$, then $\alpha_{11}x_1 + \beta_{11}y_1 + \gamma_{11}$ is always less than x_1. Thus, $i_1 > i_2$ is always true. Also, the array element corresponding to index i_1 is written and the array element corresponding to index i_2 is read. Clearly, only anti dependence can satisfy this condition. Therefore, DCH1 contains only anti dependences. Next, let us look at the case when $d_i(x_1, y_1) > 0$. Here $i_1 < i_2$. Clearly, only flow dependence can satisfy this condition. Therefore, DCH1 contains only anti dependence.

The proof for DCH2 follows similarly. Thus, if $d_i(x, y) = 0$ does not pass through any DCH, then there is only one kind of dependence.

[**Part 2**] We have already shown above that if $d_i(x, y) = 0$ does not pass through DCH1, then there is only one kind of dependence. If the dependence is flow dependence, then from theorem 2, DCH1 contains only the flow dependence tails or anti dependence heads, making DCH1 a unique tail or head set. Similarly, if the dependence is anti dependence, then from theorem 2, DCH2 contains only the anti dependence tails or flow dependence heads, making DCH2 a unique tail or head set. 2

DCH1 and DCH2 are constructed from the same system of linear Diophantine equations and system of inequalities. The following two theorems highlight the common attributes.

THEOREM 4.3. If $d_i(x_1, y_1) = 0$ does not pass through DCH1, then $d_i(x_2, y_2) = 0$ does not pass through DCH2.

Proof. If $d_i(x_1, y_1) = 0$ does not pass through DCH1, then either DCH1 lies on the side where $d_i(x_1, y_1) < 0$ or on the side where $d_i(x_1, y_1) > 0$. First let us consider the case when DCH1 is on same side of $d_i(x_1, y_1) < 0$. Since $d_i(x_1, y_1)$ is $i_2 - i_1$, we have that $i_2 < i_1$. We can find the same solution (i_1, j_1, i_2, j_2) for DCH2, because they are solved from the same set of linear Diophantine equations. $d_i(x_2, y_2)$ is also defined as $i_2 - i_1$. Hence, we can get $d_i(x_2, y_2) < 0$ which means $d_i(x_2, y_2) = 0$ does not pass through DCH2.

The second case when DCH1 is on the same side of $d_i(x_1, y_1) > 0$ can be proved similarly.. 2

COROLLARY 4.4. When $d_i(x_1, y_1) = 0$ does not pass through DCH1,

1. if $d_i(x_1, y_1) > 0$ in DCH1,
   (a) DCH1 is flow dependence unique tail set.
   (b) DCH2 is flow dependence unique head set.
2. if $d_i(x_1, y_1) < 0$ in DCH1,
   (a) DCH1 is anti dependence unique head set.
   (b) DCH2 is anti dependence unique tail set.

Proof. It follows from theorems 2 and 3. 2

COROLLARY 4.5. When $d_i(x_1, y_1) = 0$ does not pass through DCH1,

1. if $d_i(x_1, y_1) > 0$ in DCH1, then $d_i(x_2, y_2) > 0$ in DCH2.
2. if $d_i(x_1, y_1) < 0$ in DCH1, then $d_i(x_2, y_2) < 0$ in DCH2.

Proof. It is obvious from the above theorems and proofs given. 2

We have now established that if $d_i(x_1, y_1) = 0$ does not pass through DCH1, then both DCH1 and DCH2 are unique sets.

When $d_i(x, y) = 0$ passes through the CDCH, a DCH might contain both the dependence heads and tails (even if DCH1 and DCH2 do not overlap). This makes it harder to find the unique head and tail sets. The next theorem looks at some common attributes when $d_i(x, y) = 0$ passes through the CDCH.

THEOREM 4.6. If $d_i(x_1, y_1) = 0$ passes through DCH1, then $d_i(x_2, y_2) = 0$ must pass through DCH2.

Proof. Suppose $d_i(x_1, y_1) = 0$ passes through DCH1. Then we must be able to find (x_1', y_1') such that $d_i(x_1', y_1') < 0$ and (x_1'', y_1'') such that $d_j(x_1'', y_1'') > 0$ in DCH1. Correspondingly we can find (x_2', y_2') and (x_2'', y_2'') in DCH2 such that $d_i(x_1', y_1') = i_2' - i_1' = d_i(x_2', y_2')$ and $d_i(x_1'', y_1'') = i_2'' - i_1'' = d_i(x_2'', y_2'')$. Therefore, we have $d_i(x_2', y_2') < 0$ and $d_i(x_2'', y_2'') > 0$. Hence, $d_i(x_2, y_2) = 0$ must pass through DCH2. 2

Using the above theorem we can now deal with the case where a DCH contains all the dependence tails of one kind and all the dependence heads of another kind.

THEOREM 4.7. If $d_i(x, y) = 0$ passes through a DCH, then it will divide that DCH into a unique tail set and a unique head set. Furthermore, $d_j(x, y) = 0$ decides the inclusion of $d_i(x, y) = 0$ in one of the sets.

Proof. The proof for DCH1 and DCH2 are symmetric. Let us consider the case where $d_i(x_1, y_1) = 0$ passes through DCH1. First consider flow dependences. Without loss of generality, let (i_1, j_1) and (i_2, j_2) be the iterations which cause any flow dependence. Then, (i_1, j_1) and (i_2, j_2) satisfy (1). Thus, from the definition of

flow dependence, we have either $i_1 < i_2$ or $i_1 = i_2$ and $j_1 < j_2$. We can now solve (1) with

$$\begin{aligned} i_1 &= x_1 \\ j_1 &= y_1 \\ i_2 &= \alpha_{11}x_1 + \beta_{11}y_1 + \gamma_{11} \\ j_2 &= \alpha_{12}x_1 + \beta_{12}y_1 + \gamma_{12} \end{aligned} \qquad (7)$$

Since $x_1 < \alpha_{11}x_1 + \beta_{11}y_1 + \gamma_{11}$, we have $(\alpha_{11} - 1)x_1 + \beta_{11}y_1 + \gamma_{11} = d_i(x_1, y_1) > 0$. From the above equations we also have $x_1 = \alpha_{11}x_1 + \beta_{11}y_1 + \gamma_{11}$ and $y_1 < \alpha_{12}x_1 + \beta_{12}y_1 + \gamma_{12}$, which gives us $d_i(x_1, y_1) = 0$ and $d_j(x_1, y_1) > 0$.

Now let us consider anti dependence. We either have $i_1 > i_2$ or $i_1 = i_2$ and $j_1 > j_2$. Since $x_1 > \alpha_{11}x_1 + \beta_{11}y_1 + \gamma_{11}$, we have $(\alpha_{11} - 1)x_1 + \beta_{11}y_1 + \gamma_{11} = d_i(x_1, y_1) < 0$. From the set of equations (7) above we also have $x_1 = \alpha_{11}x_1 + \beta_{11}y_1 + \gamma_{11}$ and $y_1 > \alpha_{12}x_1 + \beta_{12}y_1 + \gamma_{12}$, which gives us $d_i(x_1, y_1) = 0$ and $d_j(x_1, y_1) < 0$.

$d_i(x_1, y_1) = 0$ divides DCH1 into two parts, $d_i(x_1, y_1) > 0$ and $d_i(x_1, y_1) < 0$. Flow dependences satisfy $d_i(x_1, y_1) > 0$. From theorem 2 we know that these are the flow dependence tails. Whether $d_i(x_1, y_1) = 0$ belongs to this set is dependent on whether $d_j(x_1, y_1) > 0$ or not. Therefore, $d_i(x_1, y_1) \geq 0$ decides the flow dependence unique tail set. Similarly $d_i(x_1, y_1) \leq 0$ decides the anti dependence unique head set. 2

Note that if $d_j(x_1, y_1) > 0$, then the line segment corresponding to $d_i(x_1, y_1) = 0$ belongs to the flow dependence unique tail set and if $d_j(x_1, y_1) < 0$, then the line segment corresponding to $d_i(x_1, y_1) = 0$ belongs to the anti dependence unique head set. The iteration corresponding to the intersection of $d_i(x_1, y_1) = 0$ and $d_j(x_1, y_1) = 0$, has no cross-iteration dependence. If the intersection point of $d_i(x_1, y_1) = 0$ and $d_j(x_1, y_1) = 0$ lies in DCH1, then one segment of the line $d_i(x_1, y_1) = 0$ inside DCH1 is a subset of the flow dependence unique tail set and the other segment of the line $d_i(x_1, y_1) = 0$ inside DCH1 is a subset of the anti dependence unique head set.

For DCH2, we have similar results as above. To summarize, the following corollary is derived.

COROLLARY 4.8. The flow dependence unique tail set is expressed by

$$\begin{cases} L_1 \leq & x_1 & \leq U_1 \\ L_2 \leq & y_1 & \leq U_2 \\ L_1 \leq & \alpha_{11}x_1 + \beta_{11}y_1 + \gamma_{11} & \leq U_1 \\ L_2 \leq & \alpha_{12}x_1 + \beta_{12}y_1 + \gamma_{12} & \leq U_2 \\ d_i(x_1, y_1) > 0 & \textit{and} & d_i(x_1, y_1) = 0 \\ & & d_j(x_1, y_1) > 0 \end{cases}$$

The anti dependence unique head set is expressed by

$$\begin{cases} L_1 \leq & x_1 & \leq U_1 \\ L_2 \leq & y_1 & \leq U_2 \\ L_1 \leq & \alpha_{11}x_1 + \beta_{11}y_1 + \gamma_{11} & \leq U_1 \\ L_2 \leq & \alpha_{12}x_1 + \beta_{12}y_1 + \gamma_{12} & \leq U_2 \\ d_i(x_1, y_1) < 0 & \textit{and} & d_i(x_1, y_1) = 0 \\ & & d_j(x_1, y_1) < 0 \end{cases}$$

The flow dependence unique head set is expressed by

$$\begin{cases} L_1 \leq & \alpha_{21}x_2 + \beta_{21}y_2 + \gamma_{21} & \leq U_1 \\ L_2 \leq & \alpha_{22}x_2 + \beta_{22}y_2 + \gamma_{22} & \leq U_2 \\ L_1 \leq & x_2 & \leq U_1 \\ L_2 \leq & y_2 & \leq U_2 \\ d_i(x_2, y_2) > 0 & \textit{and} & d_i(x_1, y_1) = 0 \\ & & d_j(x_2, y_2) > 0 \end{cases}$$

The ant- dependence unique tail set is expressed by

$$\begin{cases} L_1 \leq & \alpha_{21}x_2 + \beta_{21}y_2 + \gamma_{21} & \leq U_1 \\ L_2 \leq & \alpha_{22}x_2 + \beta_{22}y_2 + \gamma_{22} & \leq U_2 \\ L_1 \leq & x_2 & \leq U_1 \\ L_2 \leq & y_2 & \leq U_2 \\ d_i(x_2, y_2) < 0 & \textit{and} & d_i(x_1, y_1) = 0 \\ & & d_j(x_2, y_2) < 0 \end{cases}$$

Proof. It follows directly from Theorem 6. 2

COROLLARY 4.9. When $d_i(x_1, y_1) = 0$ passes through DCH1, then

1. DCH1 is the union of the flow dependence unique tail set and the anti dependence unique head set.
2. DCH2 is the union of the flow dependence unique head set and the anti dependence unique tail set.

Proof. It follows from Corollary 3. 2

Figure 8 illustrates the applications of our results to *Example 2*. Clearly $d_i(x_1, y_1) = 0$ divides DCH1 into two parts. The area on the left side of $d_i(x_1, y_1) = 0$ is the flow dependence unique tail set and the area on the right side of $d_i(x_1, y_1) = 0$ is the anti dependence unique head set. $d_i(x_1, y_1) = 0$ belongs to anti dependence unique head set. $d_i(x_2, y_2) = 0$ divides DCH2 into two parts too. The area below $d_i(x_2, y_2) = 0$ is the flow dependence unique head set and the area above $d_i(x_2, y_2) = 0$ is the anti dependence unique tail set. $d_i(x_2, y_2) = 0$ belongs to anti dependence unique tail set.

5. UNIQUE SETS ORIENTED PARTITIONING

In the previous sections we have grouped iterations based on their being unique head or tail sets. Clearly the unique head set will execute after the unique tail set. For our program model, there are at most four sets, *i.e.,* flow dependence unique tail set, flow dependence head set, anti dependence unique tail set, and anti dependence unique head set. The iterations outside these sets can be executed concurrently. Moreover, the iterations within each set can be executed concurrently. In order to maximize the parallelism, we want to partition the iteration space according to unique sets.

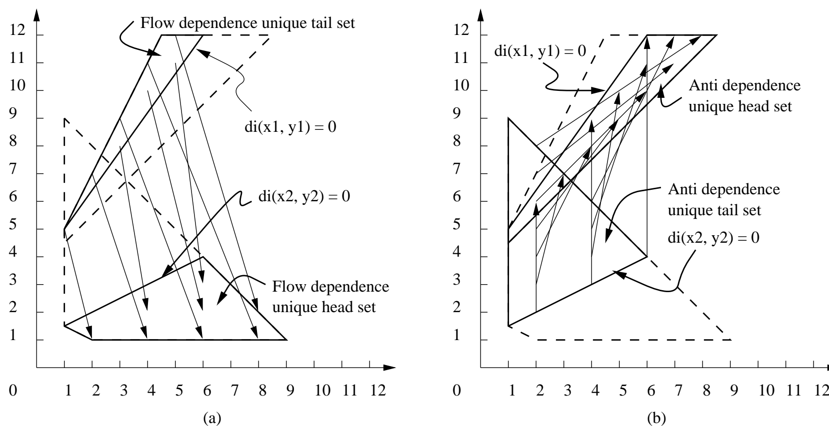


FIGURE 8. *Unique head sets and unique tail sets of (a) Flow dependence, (b) Anti dependence*

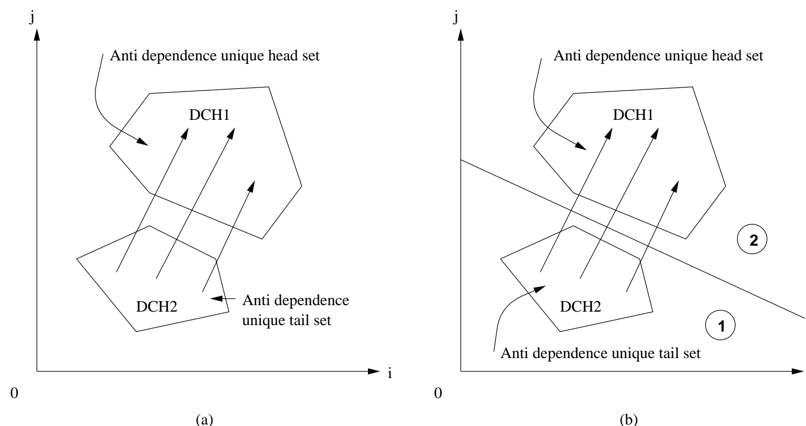


FIGURE 9. *One kind of dependence and DCH1 does not overlap with DCH2*

It is important, however, to note that the effectiveness of a partitioning scheme depends on the architecture of the parallel machine being used. In this paper we do not recommend partitions for particular architectures, rather, we explore the various partitions that can be generated from the available information. The suitability of a particular partition for a specific architecture is not studied.

Based on the unique head and tail sets that we can identify that there exist various combinations of overlaps (and/or disjointness) of these unique head and tail sets. We categorize these combinations as various cases starting from simpler cases and leading up to the more complicated ones.

Case1: There is only one kind of dependence and DCH1 does not overlap with DCH2.

Figure 9(a) illustrates this relatively easy case with an example. Any line drawn between DCH1 and DCH2 divides the iteration space into two areas. Inside each area, all iteration are independent. The DCHs in this case are unique head and unique tail sets. The iterations within each DCH can be executed concurrently. However, DCH2 needs to execute before DCH1 as shown by the partitioning in Figure 9(b). The execution order is given as $1 \to 2$.

From the implementation point of view, it is advisable to partition the iteration space along the i or j axis so that the partitioned areas can be easily represented as a loop. It is also advisable to partition the iteration space as evenly as possible. However, the final decision on partitioning will depend on the underlying architecture.

Case 2: There is only one kind of dependence and DCH1 overlaps with DCH2.

Figure 10(a) illustrates this case. DCH1 and DCH2 overlap to produce three distinct areas denoted by $Area1$, $Area2$, and $Area3$, respectively. $Area2$ and $Area3$ are either unique tail or unique head sets and thus iterations within each set can execute concurrently. $Area1$ contains both dependence heads and tails. We can apply the *Minimum Dependence Distance Tiling* technique proposed by Punyamurtula and Chaudhary 12 to $Area1$. Depending on the type of dependence there are two distinct execution orders possible. If DCH2 is a unique tail set, then the execution order is $Area3 \to Area1 \to Area2$. Otherwise the execution order is $Area2 \to Area1 \to Area3$.

From the implementation point of view, we want to use a straight line to partition the iteration space, so

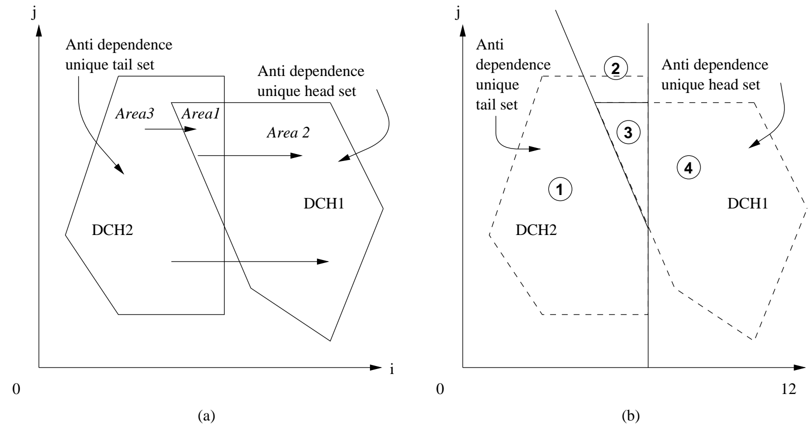


FIGURE 10. *One kind of dependence and DCH1 overlaps with DCH2*

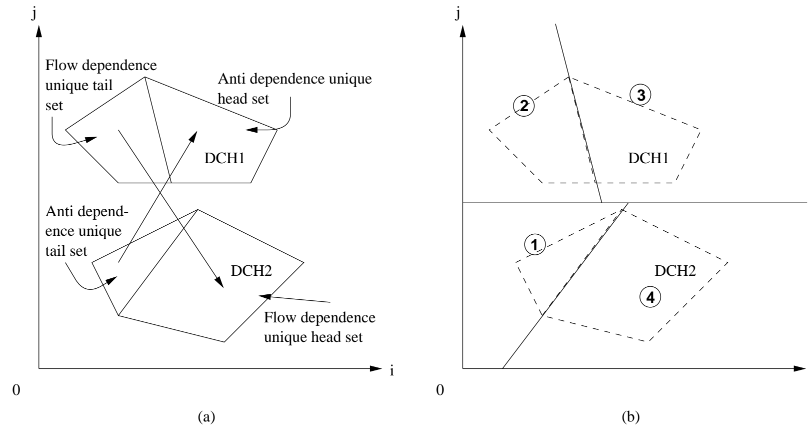


FIGURE 11. *Two kinds of dependence and DCH1 does not overlap with DCH2*

that the generated code will be much simpler. An example partitioning is shown in Figure 10(b) for the problem in Figure 10(a). The execution order is given as $1 \rightarrow 2 \rightarrow 3 \rightarrow 4$.

Another approach to parallelize the iteration space in this case is to apply the *Minimum Dependence Distance Tiling* technique 12 directly to the entire iteration space.

Case 3: There are two kinds of dependence and DCH1 does not overlap with DCH2.

Figure 11 illustrates this case. Since DCH1 and DCH2 are disjoint we can partition the iteration space into two, with DCH1 and DCH2 belonging to distinct partitions. From Theorem 6 we know that $d_i(x,y) = 0$ will divide the DCHs into unique tail and unique head sets. Next, we partition the area within DCH1 by the line $d_i(x_1, y_1) = 0$, and the area within DCH2 by the line $d_i(x_2, y_2) = 0$. So, we have four partitions, each of which is totally parallelizable. Figure 11(b) gives one possible partition with execution order as $1 \rightarrow 2 \rightarrow 3 \rightarrow 4$. Note that the unique head sets must execute after the unique tail sets.

Case 4: There are two kinds of dependence and DCH1 overlaps with DCH2, and there is at least one isolated unique set.

Figure 12 (a) and (c) illustrate this case. What we want to do is to separate this isolated unique set from the others. The line $d_i(x,y) = 0$ is the best candidate to do this. If $d_i(x,y) = 0$ does not intersect with any other unique set or another DCH, then it will divide the iteration space into two parts as shown in Figure 12(b). If $d_i(x,y) = 0$ does intersect with other unique sets or another DCH, we can add one edge of the other DCH as the boundary to partition the iteration space into two as shown in Figure 12(d). Let us denote the partition containing the isolated unique set by $Area2$. The other partition is denoted by $Area1$. If $Area2$ contains a unique tail set, then $Area2$ must execute before $Area1$, otherwise $Area2$ must execute after $Area1$. The next step is to partition $Area1$. Since $Area1$ has only one kind of dependence (as long as we maintain the execution order defined above) and DCH1 overlaps with DCH2, it falls under the category of case 2 and can be further partitioned.

Case 5: There are two kinds of dependence and all unique sets overlap each other.

Figure 13(a) illustrates this case. The CDCH can be partitioned into at most eight parts as shown in Figure 13(b). These partitions are areas that contain

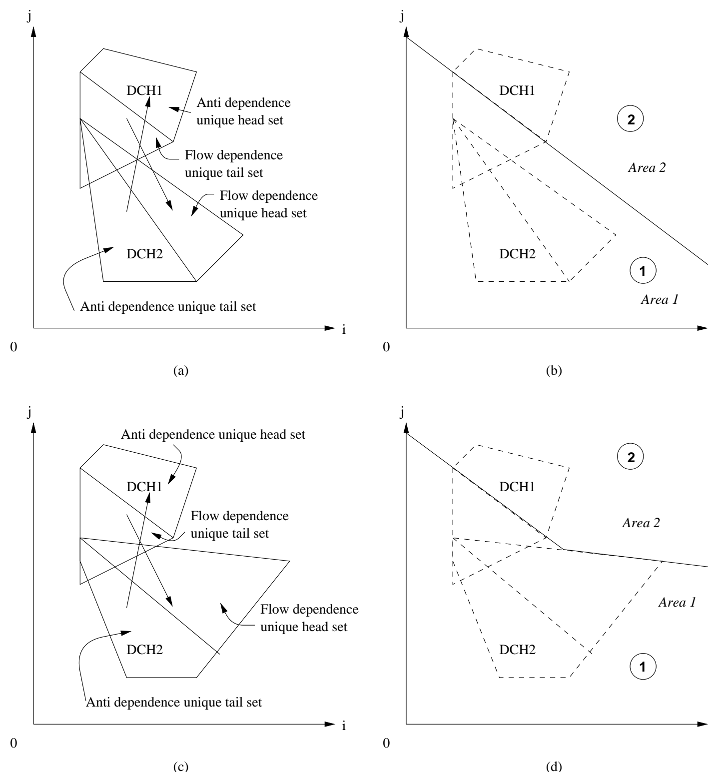


FIGURE 12. *Two kinds of dependence and one unique set isolated*

- only flow dependence tails, and we denote it by $Area1$.
- only anti dependence tails, and we denote it by $Area2$.
- only anti dependence heads,and we denote it by $Area3$.
- only flow dependence heads, and we denote it by $Area4$.
- flow dependence tails and anti dependence tails, and we denote it by $Area5$.
- flow dependence heads and anti dependence heads, and we denote it by $Area6$.
- flow dependence tails and flow dependence heads, and we denote it by $Area7$.
- anti dependence tails and anti dependence heads, and we denote it by $Area8$.

$Area1$, $Area2$, and $Area5$ can be combined together into a larger area, because they contain only the dependence tails. Let us denote this combined area by $AreaI$. In the same way, $Area3$, $Area4$, and $Area6$ can also be combined together, because they contain only the dependence heads. Let us denote this combined area by $AreaII$. $AreaI$ and $AreaII$ are fully parallelizable. The execution order becomes $AreaI \rightarrow Area7 \rightarrow Area8 \rightarrow AreaII$. Since $Area7$ and $Area8$ contain both dependence heads and tails, we can apply *Minimum Dependence Distance Tiling* technique to parallelize this area.

We may not always have all eight areas in this case. For example, if $d_i(x_1, y_1) = 0$ does not intersect $d_i(x_2, y_2) = 0$ inside the CDCH, then either $Area7$ or $Area8$ exists, but not both. However, the proposed partitioning and execution order still hold.

Now let us go back to *Example 2.* From Figure 8, we know that it fits in the category of Case 4.

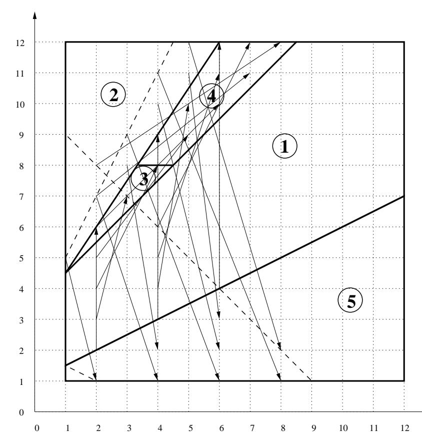


FIGURE 14. *Partitioning scheme for Example 2*

The partitioning scheme is shown in figure 14. There are five areas. All the iterations in each area are fully parallelizable. These area should be run in the order of $1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 5$. Area 3 is the overlapping area. *Minimum Dependence Distance Tiling* technique25 is adopted to partition along the j direction with minimum distance of 4. The parallelized code of *Example 2*

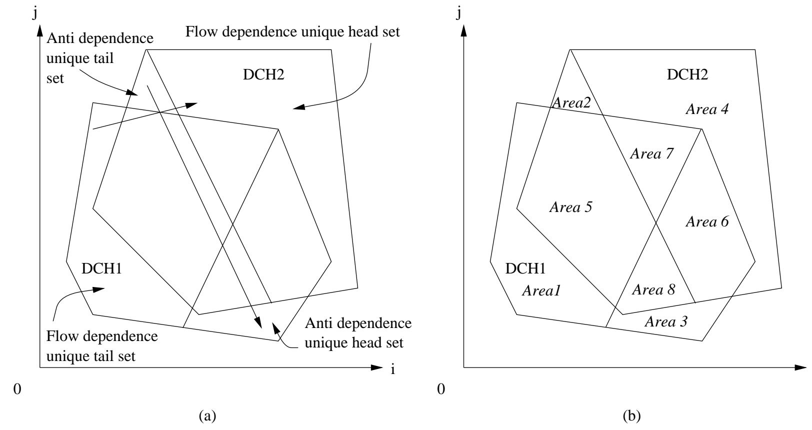


FIGURE 13. *Two kinds of dependence and all unique sets overlapped each other*

is shown below.

```
/* area 1 */
doparallel i = 1, 12
  doparallel j = ceil(i/2 + 1), min(floor(i + 3.5), 12)
    A(2 * i + 3, j + 1) = ···
    ··· = A(2 * j + i + 1, i + j + 3)
  enddo
enddo
/* area 2 */
doparallel i = 1, 6
  doparallel j = (floor((3i)/2 + 3) + 1), 12
    A(2 * i + 3, j + 1) = ···
    ··· = A(2 * j + i + 1, i + j + 3)
  enddo
enddo
/* area 3 */
doparallel i = floor((3i)/2 + 3), ceil(x + 7/2)
  doparallel j = 5, 8
    A(2 * i + 3, j + 1) = ···
    ··· = A(2 * j + i + 1, i + j + 3)
  enddo
enddo
/* area 4 */
doparallel i = floor((3i)/2 + 3), ceil(x + 7/2)
  doparallel j = 9, 12
    A(2 * i + 3, j + 1) = ···
    ··· = A(2 * j + i + 1, i + j + 3)
  enddo
enddo
/* area 5 */
doparallel i = 1, 12
  doparallel j = 1, (ceil(i/2 + 1) − 1)
    A(2 * i + 3, j + 1) = ···
    ··· = A(2 * j + i + 1, i + j + 3)
  enddo
enddo
```

This partitioning scheme seems to be worse than other techniques at first glance. This is because the loop upper bounds is only 12. As the loop upper bounds increase, this scheme will show the advantage. No matter how large the loop is, it synchronizes only five times. Synchronization overhead is always the major factor that affects the performance.

6. EXTENSION TO GENERAL NESTED LOOPS

We discussed the parallelization of two dimensional program model in the former sections. We now look at loops with n levels of nestings whose indices are i_1, i_2, $\cdots$, i_n. The array subscripts are linear functions of loop indices as shown in figure 15.

```
      do i_1 = L_1, U_1
        ···
        do i_n = L_n, U_n
S_1:      A[f_1(i_1, ..., i_n), ..., f_m(i_1, ..., i_n)] = ···
S_2:      ··· = A[g_1(i_1, ..., i_n), ..., g_m(i_1, ..., i_n)]
        enddo
        ···
      enddo
```

FIGURE 15. *General Program Model*

We want to find a set of integer solutions $(i_1, \ldots, i_n, i'_1, \ldots, i'_n)$ that satisfy the system of Diophantine equations (8) and the system of linear inequalities (9).

$$\begin{array}{rcl} f_1(i_1,\ldots,i_n) & = & g_1(i'_1,\ldots,i'_n) \\ & \vdots & \\ f_m(i_1,\ldots,i_n) & = & g_m(i'_1,\ldots,i'_n) \end{array} \quad (8)$$

$$\left\{ \begin{array}{ccccc} L_1 & \le & i_1 & \le & U_1 \\ L_1 & \le & i'_1 & \le & U_1 \\ & & \cdots & & \\ L_n & \le & i_n & \le & U_n \\ L_n & \le & i'_n & \le & U_n \end{array} \right. \quad (9)$$

To avoid lengthy repetition, we consider DCH2 as an example to illustrate how to get unique sets. From former sections, we know that DCH2 should contain

flow dependence unique head set and anti dependence unique tail set. Using the second approach to solve the set of Diophantine equations, we have integer solutions $(i_1, \cdots, i_n, i'_1, \cdots, i'_n)$ which are functions of $x_1, \cdots, x_n$. They can be written as:

$$(i_1, \cdots, i_n, i'_1, \cdots, i'_n) = (s_1(x_1, \cdots, x_n), \cdots, s_n(x_1, \cdots, x_n), s_{n+1}(x_1, \cdots, x_n), \cdots, s_{n+n}(x_1, \cdots, x_n))$$

From the general solution the dependence vector function $D(x_1, \cdots, x_n)$ can be written as

$$D(x_1, \cdots, x_n) = \{(s_{n+1}(x_1, \cdots, x_n) - s_1(x_1, \cdots, x_n)), \cdots, (s_{n+n}(x_1, \cdots, x_n) - s_n(x_1, \cdots, x_n))\}$$

Hence the dependence vectors are:

$$\begin{cases} d_1(x_1, \cdots, x_n) = (s_{n+1}(x_1, \cdots, x_n) - s_1(x_1, \cdots, x_n)) \\ \vdots \\ d_n(x_1, \cdots, x_n) = (s_{n+n}(x_1, \cdots, x_n) - s_n(x_1, \cdots, x_n)) \end{cases}$$

The dependence vector $D(x_1, \cdots, x_n)$ divides this DCH into two parts. One is flow dependence unique head set and the other is anti dependence unique tail set. The decision on the ownership of $D(x_1, \cdots, x_n)$ comes next.

The theorems proposed in section 4.2 are also valid for multi-dimensional loops. $d_1(x_1, \cdots, x_n) > 0$ belongs to flow dependence unique head set and $d_1(x_1, \cdots, x_n) < 0$ belongs to flow anti dependence unique tail set. When $d_1(x_1, \cdots, x_n) = 0$, $d_2(x_1, \cdots, x_n)$ has to be checked. If $d_2(x_1, \cdots, x_n) > 0$, then flow dependence unique head set contains $d_1(x_1, \cdots, x_n) = 0$ and $d_2(x_1, \cdots, x_n) > 0$. If $d_2(x_1, \cdots, x_n) < 0$, then anti dependence unique head tail contains $d_1(x_1, \cdots, x_n) = 0$ and $d_2(x_1, \cdots, x_n) < 0$. For $d_2(x_1, \cdots, x_n) = 0$, $d_3(x_1, \cdots, x_n)$ has to be checked. We continue in this fashion until $d_n(x_1, \cdots, x_n)$ is checked.

Using this method, we can get the unique sets for the given general program model. According to the positions of these sets, we can partition the iteration space. During the partitioning, the area containing unique tail set must be run before the area containing unique head set. The partitioning process is basically the same as for doubly nested loops, except that we now deal everything with multi-dimensional iteration space. The shape of the unique set is also multi-dimensional.

An alternative way to parallelize multi-dimensional loops is to parallelize only the two outer most loop nests, leaving inner loops running sequentially. The advantages of one approach over the other is left for future work. However, we feel that multi-dimensional unique set of partitioning will give us greater flexibility to transform the loops to adapt specific architectures.

7. EXPERIMENTAL RESULTS

We present results for two programs. The first program is similar to *Example 2* as shown in Figure 16. We tested the performance for varying loop sizes. The loop sizes ($SIZE$) used in the experiments are 50, 100, 500, and 1000.

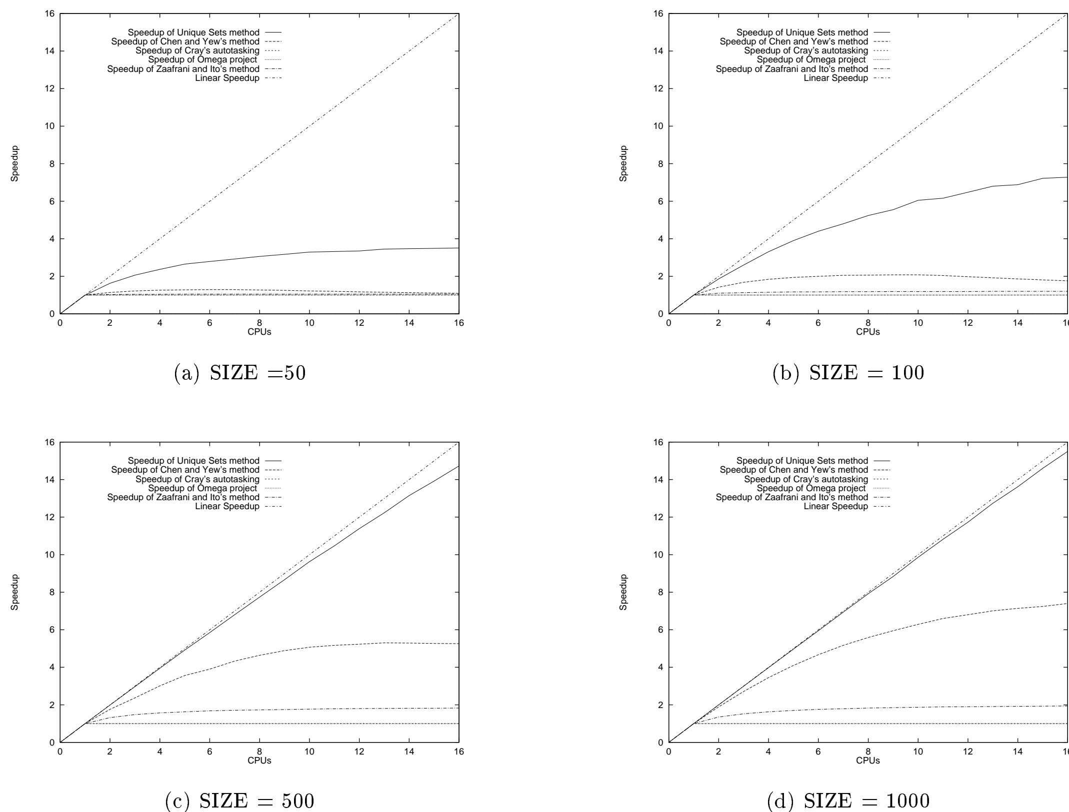
> do $i = 1$, $SIZE$
>   do $j = 1$, $SIZE$
> $A(2 * i + 3, j + 1) = \cdots$
> $\cdots = A(i + 2j + 1, i + j + 3)$
>   enddo
> enddo

FIGURE 16. *Program 1*

```
1          SUBROUTINE CHOLSKY (IDA, NMAT, M, N, A, NRHS, IDB, B)
2    C
3    C   CHOLESKY DECOMPOSITION/SUBSTITUTION SUBROUTINE.
4    C
5    C   11/28/84  D H BAILEY  MODIFIED FOR NAS KERNEL TEST
6    C
7          REAL A(0:IDA, -M:0, 0:N), B(0:NRHS, 0:IDB, 0:N), EPSS(0:256)
8          DATA EPS/1E-13/
9    C
10   C  CHOLESKY DECOMPOSITION
11   C
12         DO 1 J = 0, N
13           IO = MAX ( -M, -J )
14   C
15   C  OFF DIAGONAL ELEMENTS
16   C
17           DO 2 I = IO, -1
18             DO 3 JJ = IO - I, -1
19               DO 3 L = 0, NMAT
20   3             A(L,I,J) = A(L,I,J) - A(L,JJ,I+J) * A(L,I+JJ,J)
21             DO 2 L = 0, NMAT
22   2           A(L,I,J) = A(L,I,J) * A(L,0,I+J)
23   C
24   C  STORE INVERSE OF DIAGONAL ELEMENTS
25   C
26           DO 4 L = 0, NMAT
27   4         EPSS(L) = EPS * A(L,0,J)
28           DO 5 JJ = IO, -1
29             DO 5 L = 0, NMAT
30   5           A(L,0,J) = A(L,0,J) - A(L,JJ,J) ** 2
31           DO 1 L = 0, NMAT
32   1         A(L,0,J) = 1. / SQRT ( ABS (EPSS(L) + A(L,0,J)) )
33   C
34   C  SOLUTION
35   C
36         DO 6 I = 0, NRHS
37           DO 7 K = 0, N
38             DO 8 L = 0, NMAT
39   8           B(I,L,K) = B(I,L,K) * A(L,0,K)
40             DO 7 JJ = 1, MIN (M, N-K)
41               DO 7 L = 0, NMAT
42   7             B(I,L,K+JJ) = B(I,L,K+JJ) -  A(L,-JJ,K+JJ) * B(I,L,K)
43   C
44           DO 6 K = N, 0, -1
45             DO 9 L = 0, NMAT
46   9           B(I,L,K) = B(I,L,K) * A(L,0,K)
47             DO 6 JJ = 1, MIN (M, K)
48               DO 6 L = 0, NMAT
49   6             B(I,L,K-JJ) = B(I,L,K-JJ) - A(L,-JJ,K) * B(I,L,K)
50   C
51         RETURN
52         END
```

FIGURE 17. *Program 2*

The second program is shown in Figure 17. This is a subroutine taken from a benchmark test program which has been developed for use by the NAS program at NASA Ames Research Center to aid in the evaluation of supercomputer. This subroutine deals with the problem of Cholesky Decomposition and Substitution. We are more interested in the part from line 17 to line 22. Non-uniform dependences can be found in this part of the program. To illustrate the impact of non-uniform dependence and to make our experiment more comprehensive, we use the entire subroutine to evaluate the performance of our technique. In fact, the variable N and $NMAT$ decide the program size in this part of program. When we say the the program size is 50, both N and $NMAT$ are set to 50. We present results for



(a) SIZE =50 (b) SIZE = 100

(c) SIZE = 500 (d) SIZE = 1000

FIGURE 18. *Performance Results for Program 1 on Cray*

program sizes 50, 100, 200, and 300, respectively.

All the experiments are done on a Cray J916 with 16 processors. *Autotasking Expert System*(atexpert) are used to analyze the program. Atexpert is a tool developed by CRI (Cray Research, Inc.) for accurately measuring and graphically displaying tasking performance from a job run on an arbitrarily loaded CRI system. It can predict speedups on a dedicated system from data collected from a single run on a non-dedicated system. It shows where a program is spending most of its time and whether those areas are executed sequentially or in parallel.

User-Directed Tasking directives are used to construct parallelizable areas in the iteration space. Synchronizations are implemented with the help of *guarded region*. The format is as below.

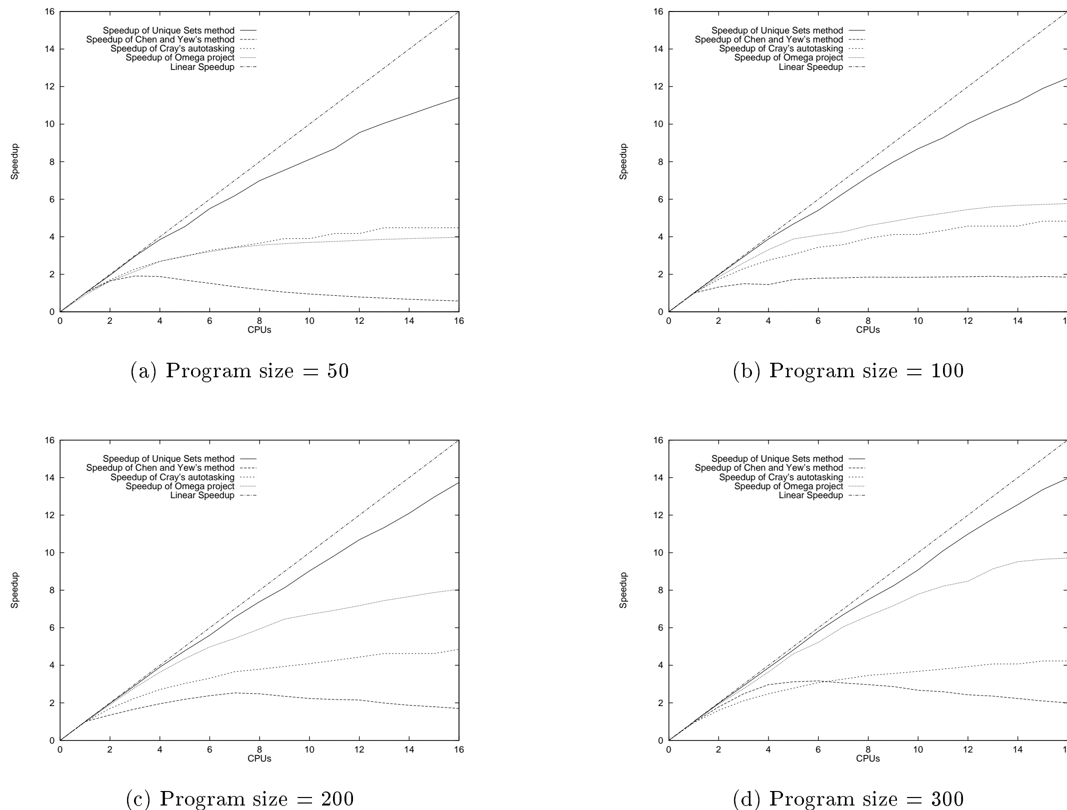
```
#pragma _CRI parallel defaults
#pragma _CRI taskloop
    loop
#pragma _CRI endparallel

#pragma _CRI guard
    loop or variable
#pragma _CRI endguard
```

Our results are compared with those of Chen and Yew's method 9, Cray's native Autotasking, Omega project of University of Maryland 5, and Zaafrani and Ito's method 13. Zaafrani and Ito's method is not implemented for *Program 2*, because it is unable to handle non-perfect nestings of loops. To implement Chen and Yew's method, guarded regions were used to simulate the function of semaphore. For the method of Omega project, version 1.1 of the Omega Project software was used. We run the source codes through *Petit*, a research tool developed by University of Maryland. It calls both the *Omega library* and the *Uniform library* 5 and generates parallelized *c* source code. We rewrite the parallelized source codes with Cray's Autotasking directives to do the experiments.

Figure 18 shows the speedup comparison of our technique, Chen and Yew's technique, Cray's autotasking, Omega project, and Zaafrani and Ito's three-region technique. Cray's autotasking did not give any speedup at all, running the loops sequentially. Omega project did not parallelize this program either. It is not so clear in Figure 18, because the speedups of Omega project and those of Cray's autotasking are overlapped. Both are 1.

Our method shows near linear speedup with the loop size of 500 and 1000, which are the models closer to the real world programs. Our technique is consistently outperforms other techniques considerably for all sizes. Chen and Yew's gave some speedup, but not too much,



(a) Program size = 50

(b) Program size = 100

(c) Program size = 200

(d) Program size = 300

FIGURE 19. *Performance Results for Program 2 on Cray*

because of the synchronization overhead. Zaafrani and Ito's method showed very little speedup. The sequential region of their method is the bottle neck for good performance. The figure shows that the loop sizes have a tremendous impact on the performance even for the same loop using the same parallelization technique. In practice, we alway want to parallelize the loops where programs spend most of their time.

Figure 19 shows the performance for the Cholesky Decomposition subroutine. From the plots, it is clear that our technique outperforms all the other techniques. As program size increases, our technique shows better results. Cray's Autotasking got some speed up for this routine. It parallelized the inner most loop. This is more like vectorizing than parallelizing. The result of Omega project is worse than that of Cray's autotasking when the program size of 50, as shown in Figure 19(a). As the program size increases, it outperformed the Cray's autotasking. When the program size is 300, the performance of Omega project is nearly twice that of Cray's autotasking. The reason is that Cray's autotasking only parallelizes the innermost loops, while Omega project does not. Overall, Chen and Yew's technique performed worst. Again, increased synchronization is responsible for this.

8. CONCLUSION

In this paper, we systematically analyzed the characteristics of the dependences in the iteration space. We proposed the concept of Complete Dependence Convex Hull, which contains the entire dependence information of the program. We also proposed the concepts of Unique head sets and Unique tail sets which isolated the dependence information and showed the relationship among the dependences. The relationship of the unique head and tail sets forms the foundation for partitioning the iteration space. Depending on the relative placement of these unique sets, various cases were considered. Several partitioning schemes were also suggested for implementating our technique. The suggested scheme was implemented on a Cray J916 and compared with Chen and Yew's method 9, Cray's native Autotasking, Omega project of University of Maryland 5, and Zaafrani and Ito's method 13. The implementation results of real benchmark code shows that our technique consistently outperformed all the other techniques considerably.

ACKNOWLEDGMENTS

We would like to thank Sumit Roy for his help in the implementation of the techniques on the Cray J916 and his comments on a preliminary draft of the paper. We

would also like to thank Chengzhong Xu for his constructive comments on the contents of this paper.

REFERENCES

[1] Kuck, D., Sameh, A., Cytron, R., Polychronopoulos, A., Lee, G., McDaniel, T., Leasure, B., Beckman, C., Davies. J., and Kruskal, C. (1984) The effects of program restructuring, algroithm change and architecture choice on program performance. *Proceedings of the 1984 International Conference on Parallel Processing.*

[2] Tzen T. H. (1992) Advanced Loop Parallelization: Dependence Uniformization and Trapezoid Self-scheduling. *PhD thesis, Michigan State University.*

[3] http://suif.stanford.edu/suif.html.

[4] http://www.csrd.uiuc.edu/parafrase2/.

[5] http://www.cs.umd.edu/projects/omega/.

[6] Shen, Z., Li, Z., and Yew, P. C. (1989) An empirical study on array subscripts and data dependencies. *Proceedings of the International Conference on Parallel Processing*, **II**, 145–152.

[7] Yang, Y. Q., Ancourt, C., and Irigoin, F. (1995) Minimal data dependence abstractions for loop transformations: Extended version. *International Journal of Parallel Programming*, **23**, 359–388.

[8] Tzen. Y. H. and Ni, L. M. (1993) Dependence uniformization: A loop parallelization tehnique. *IEEE transactions on Parallel and Distributed Systems*, **4**, 547–558.

[9] Chen, D. and Yew, P. (1996) On effective execution of nonuniform doacross loops. *IEEE transactions on Parallel and Distributed Systems*, **7**, 463–476.

[10] Chen, Z. and Shang, W. (1992) On uniformization of affine dependence algorithms. *Proceedings of the Fourth IEEE Symposium on Parallel and Distributed Processing*, 128–137.

[11] Shang, W. and Fortes, J. (1991) Time optimal linear schedules for algorithms with uniform dependences. *IEEE transactions on Parallel and Distributed Systems*, **40**, 723–742.

[12] Punyamurtula, S., Chaudhary, V., Ju, J., and Roy, S. (1996) Compile time partitioning of nested loop iteration spaces with non-uniform dependences. *Journal of Parallel Algorithms and Applications (special issue on Optimising Compilers for Parallel Languages).*

[13] Zaafrani, A. and Ito, M. (1994) Parallel region execution of loops with irregular dependences. *Proceedings of the International Conference on Parallel Processing*, **II**, 11–19.

[14] Tseng, S., King, C., and Tang, C. (1992) Minimum dependence vector set: A new compiler technique for enhancing loop parallelism. *Proceedings of 1992 International Conference on Parallel and Distributed Systems*, 340–346.

[15] Pugh, W. and Wonnacott, D. (1994) Static analysis of upper and lower bounds on dependences and parallelism. *ACM Transactions on Programming Languages and Systems*, **16**, 1248–1278.

[16] Li, Z., Yew, P., and Zhu, C. (1990) An efficient data dependence analysis for parallelizing compilers. *IEEE transactions on Parallel and Distributed Systems*, 26–34.

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

[18] Towle, R. (1976) Control and Data Dependence for Program Transformations. *PhD thesis, Computer Science Dept., University of Illinois at Urbana-Champaign.*

[19] Banerjee, U. (1988) Dependence Analysis for Supercomputing. *Kluwer Academic Publishers.*

[20] Sublok, J. and Kennedy, K. (1995) Integer programming for array subscript analysis. *IEEE transactions on Parallel and Distributed Systems*, **6**, 662–668.

[21] Wolfe, M. (1986) Loop skewing: The wavefront method revisited. *International Journal of Parallel Programming*, 279–293.

[22] Banerjee, U. (1993) Loop Transformations for Restructuring compilers. *Kluwer Academic Publishers.*

[23] Wolf, M. E. and Lam, M. S. (1991) A loop transformation theory and an algorithm to maximize parallelism. *IEEE transactions on Parallel and Distributed Systems*, **2**, 452–471.

[24] Wolfe, M. J. (1989) Optimizing Supercompilers for supercomputers. *MIT Press.*

[25] Punyamurtula, S. and Chaudhary, V. (1994) Minimum dependence distance tiling of nested loops with non-uniform dependences. *Symp. on Parallel and Distributed Processing*, 74–81.