Removal of Redundant Dependences in DOACROSS Loops with Constant Dependences

V.P. Krothapalli
Computer Science Department
University of Wisconsin Oshkosh
Oshkosh, WI 54901

P. Sadayappan
Department of Computer Science
The Ohio State University
Columbus, OH 43210

Abstract

Dependences constrain the parallel execution of an imperative program and are typically enforced by synchronization instructions. These synchronization instructions can represent a significant part of the overhead in the parallel execution of a parallel program. Some of the dependences in the program may be redundant because they are *covered* by some other dependences. In this paper an efficient algorithm to remove the redundant dependences in simple loops with constant dependences is presented. In the case of simple loops the redundancy is uniform, and all the redundant dependences are determined by determining redundant dependences at a node. Redundancy is not necessarily uniform in a doubly nested loop. A sufficient condition for uniformity of redundancy in a doubly nested loop is developed.

1 Introduction

Dependences constrain the parallel execution of programs. The dependence structure of a computation can be characterized by a directed acyclic graph (DAG), where the nodes represent tasks and directed edges represent precedence constraints. These dependence edges correspond to explicit synchronization required under asynchronous model of parallel execution. If the dependence edge in the DAG is transitive then there is an alternate path from the source node of a dependence to the sink node of a dependence through a sequence of other edges. The synchronizing instruction corresponding to a transitive edge is superfluous because the occurrence of the synchronization event through the alternate path implies that the precedence constraints implied by this edge is implicitly guaranteed. Since synchronization events involve non-trivial and often significant execution overhead such as message passing or busy-waiting, it is of interest to minimize the number of synchronization events, without decreasing exploitable parallelism. This paper addresses the efficient removal of redundant dependence edges in the context of simple as well as doubly nested loops with constant dependence vectors.

The problem of recognizing and removing redundant dependences in simple loops with constant dependences is studied by Li and Abu-Sufah [9] and Midkiff and Padua [10]. Li and Abu-Sufah identify only a subset of redundant dependences because they only test for potential dependence that can cover a given dependence. On the contrary, Midkiff and Padua search for a sequence of dependences that can cover a given dependence. They recognized that the regular nature of the dependence distances at each point of the iteration space could be exploited. They apply the transitive closure algorithm [1] on a small subgraph of size $d_{max} + 1$, where d_{max} is the maximum dependence distance among the dependence edges. However, their algorithm is inefficient on two counts -

1. The transitive closure is generated k times, where k is the number of dependences in the loop; each time removing the edge to be tested from the initial adjacency matrix. As shown by Aho, Gary and Ullman [1] it is not really necessary to generate k transitive closures to obtain all the redundant information.

2. As will be shown in this paper, in the context of loops with constant dependence vectors, it is in fact not necessary even to generate a single complete transitive closure - the regularity of the reduced graph can be further exploited to more efficiently

deduce all the redundant edges.

This paper extends the previous work in two ways. First an efficient algorithm for recognition of redundant edges with regular loops is proposed. The key to the efficient recognition of redundant edges in regular loops is that the identification of transitive edges from a single node of a DAG can be done by a variant of a depth first search. This is possible because sequential execution order (SEO) of the nodes of the graph can be used to ensure that no node is visited before any node that precedes it in the SEO.

The second contribution of this paper is characterization of redundancy for doubly nested loops with constant dependences. Although theoretically all the multidimensional nested loops can be transformed mechanically into a simple, often in this process the regularity in the dependence structure is lost. When a doubly nested loop with constant dependences is so transformed into a single loop, the transformed loop will not have uniform constant dependences. It is necessary to retain the 2-dimensional nature of the iteration space in order to accurately characterize redundancy of dependence edges. Two basic issues are addressed here :

1. How can an appropriate subgraph of the iteration space be identified, that can serve in the determination of redundant edges?
2. In a bounded 2-dimensional iteration space, at which points a dependence edge is redundant?

Since dependence vectors can have negative components along the inner dimension, an edge is not necessarily uniformly redundant at all points of the iteration space. A characterization of a sufficient condition for uniformity of redundancy is developed.

The background material is presented in section 2. Section 3 presents the algorithm for finding redundant edges at a node. Application of this algorithm to simple loops with constant dependences is discussed in section 4. The central idea in section 4 is similar to that of Midkiff and Padua. They consider controlled path graph (CPG), whereas we consider the iteration space dependence graph. In this section we **formally** identify the subgraph to be analyzed. This section presents a nice introduction to our proof techniques. Doubly nested loops with constant dependences are discussed in section 5. Section 6 gives the conclusions of this research. The concepts in our paper are presented by considering ISDG, and they are equally applicable for any regular graphs.

2 Background

Iterations of a simple loop can be mapped onto discrete points on a line[15]. Henceforth, unless explicitly specified, all the loops are assumed to have a step of one and the lower bound on the loop control variable (*lcv*) is assumed to be one. Hence, a simple loop with n iterations is represented by points 1 to n on a line. Similarly, the iterations of a doubly nested loop can be mapped onto discrete points in a two dimensional space. Based on the shape of the iteration space, doubly nested loops are classified into three categories: *rectangular*, *trapezoidal*, and *irregular*. An example of each of the three classes is given in Fig. 1. In the case of a *rectangular* loop the number of iterations in the inner loop is independent of the outer loop iteration. If the number of iterations in the inner loop is a linear function of the iteration number of the outer loop, then such loops are classified as *trapezoidal*. In an *irregular* loop the number of iterations in the inner loop is a non-linear function of the iteration number of the outer loop.

```
DO I := 1,IMAX
   DO J := 1,N

   END
END
```

a) Rectangular Loop

```
DO I := 1,IMAX
   DO J := 1,a * I + b

   END
END
```

b) Trapezoidal Loop

```
DO I := 1,IMAX
   DO J := 1,ub[i]

   END
END
```

c) Irregular Loop

Figure 1: Classification of Iteration Space for Doubly Nested Loops

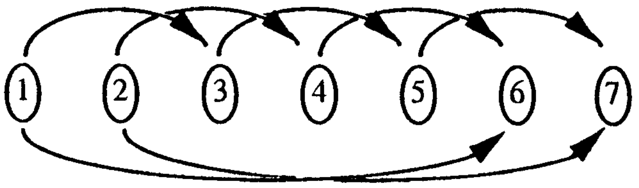
In general, iterations of a $D-$nested loop can be viewed as points in a $D-$dimensional iteration space.

The *index vector* of an iteration gives the coordinates of the corresponding point in the discrete Cartesian coordinate system. A point in an iteration space can be viewed as a vector defined by the line connecting the origin of the coordinate system to the point. We use the notation p_i to denote a point and P_i to denote the corresponding vector.

Parallel execution of the iterations of a loop is constrained by dependences. The inter-iteration dependences in a loop can be formally represented by an Iteration Space Dependence Graph (ISDG)[15]. An ISDG is a directed acyclic graph $G(V, E)$, where V and E are the set of nodes and the set of edges, respectively. Each point in the iteration space is associated with a unique node in the graph, hence, the cardinality of V is equal to the number of points in the iteration space. Henceforth, we use point and node interchangeably. An edge $(V_i, V_j) \in E$ signifies the existence of a dependence from the iteration denoted by V_i to the iteration denoted by V_j. The dependence distance of a inter-iteration dependence is given by subtracting the iteration vector of the source of a dependence from that of the sink. Hence, a dependence distance of an $N-$way nested loop is a vector in $N-$dimensional space. For any point p_1, and dependence distance d_i the *adjacent point* of the point p_1 due to the dependence is given by $P_1 + d_i$. An ISDG is classified as regular if the existence of a dependence of distance d_i from a node implies the existence of such a dependence from each node that has an *adjacent point* at distance d_i in the iteration space. Note that in a multidimensional loop d_i is a vector.

```
DO i := 1 to 7
   A[i] := A[i-3] + A[i+5];
END
```

a) Do Loop used to Illustrate ISDG



b)ISDG of the loop in (a)

Figure 2: Illustration of ISDG

Fig. 2 shows a regular ISDG. The loop given in Fig. 2(a) has seven iterations, hence, the iteration space has seven nodes. The flow dependence from iteration i to iteration $i + 3$ manifests itself as an edge from node i to node $i + 3$ in the ISDG. Similarly, due to the anti-dependence from iteration i to iteration $i + 5$ there is an edge from node i to node $i + 5$. ¿From the perspective of parallel execution, an edge (V_i, V_j) signifies the constraint that some instruction in the iteration corresponding to V_j has to be executed only after some instruction in the iteration corresponding to V_i. The denser the iteration space graph, the higher the number of synchronization to be executed. In general, synchronization instructions are expensive, and the sparser the ISDG the smaller the overheads in parallel execution. Some of the dependences in the iteration space may be redundant because they are *covered* by some other dependences. A dependence from iteration i to iteration k is redundant if and only if

- There exists a sequence of iterations $i_1, i_2, \ldots, i_j$ such that $j > 0$ and there is a dependence from i_m to i_{m+1} for $1 \leq m < j$.
- There is a dependence from iteration i to iteration i_1 and from iteration i_j to iteration k.

Elimination of redundant dependences reduces the synchronization overheads in the parallel execution of a loop. In the next section, we present an algorithm to reduce the number of synchronization instructions required by removing the redundant edges from a regular ISDG. This algorithm uses a variant of depth first search [13] in identifying all the redundant edges from a node. We present the algorithm in the next section.

3 Removal of Redundant Edges from a Node

We define the Control-flow Numbered Iteration Space Dependence Graph (CNISDG) of an ISDG by labeling the vertices of the ISDG to correspond to the sequential execution order (SEO). Since nodes are numbered by SEO, for any edge (i, j) of a CNISDG, the relation $i < j$ is true. An edge $(i, k) \in E$ is a redundant edge in a CNISDG, if and only if there exists a node j such that there is a path from node i to node k through node j and $i < j < k$. A redundant dependence in a loop manifests itself as a redundant edge in the corresponding CNISDG. Henceforth, we will use the terms redundant edge and redundant dependence interchangeably. A variant of depth first search [13] is used to identify all the redundant edges from a node.

Henceforth, we focus on removal of redundant edges from node 1 of a CNISDG. Let V' be the set of all vertices reachable from node 1 of a CNISDG and $G'(V', E')$ be the induced subgraph of the CNISDG on V'. In order to find the redundant edges from node 1, it is sufficient to analyze $G'(V', E')$. The definition of V' implies that for any node i, there exists a path from node 1 to node i in G'. Without loss of generality, we assume that any node i of a CNISDG is reachable from node 1.

Depth First Search (DFS) starting at node 1 on a CNISDG $G(V, E)$ results in a spanning tree. Given a DFS spanning tree $T(V, E')$, an edge of the CNISDG can be classified as one of the following: tree edge, forward edge, or cross edge. If an edge (i, j) belongs to E', then it is a tree edge. An edge (i, j) is a forward edge, if and only if (i, j) does not belong to E' and there exists a path from node i to node j in T. An edge (i, j) is a cross edge, if and only if (i, j) does not belong to E' and there is no path from node i to node j in T. Fig. 3 illustrates these notions. A CNISDG has one or more DFS spanning trees. For instance, three different DFS spanning trees for the graph in Fig. 3(a) are shown in Fig. 3(b), Fig. 3(c), and Fig. 3(d). The SEO number of a node is shown within the circle; the order in which a node is visited is shown outside the node. The notion of forward and cross edges are defined with respect to a DFS spanning tree. We claim that a forward edge from a node is a transitive edge.

Lemma 1 *A forward edge of a CNISDG with respect to a DFS spanning tree is redundant.*

Proof: Let $G(V, E)$ be a CNISDG, and $T(V, E')$ be a DFS spanning tree of the G. Let $(i, j) \in E$ be a forward edge with respect to the spanning tree T. The definition of a forward edge implies that there exists a path $i \rightarrow k_1 \rightarrow k_2 \rightarrow \ldots k_{n-1} \rightarrow j$ of length $n > 1$, in T from i to j. A node in a CNISDG corresponds to an iteration and an edge between two nodes signifies a dependence between the two iterations. This implies that

- There is a dependence from iteration i to iteration j.
- There is a sequence of iterations $i, k_1, k_2, \ldots, k_{n-1}, j$ such that there is a dependence between any two consecutive iterations in the sequence.

Since, $n > 1$, by definition, edge (i, j) is redundant. □

Lemma 1 states that a forward edge at a node is a redundant edge, but the converse is not true. For instance, in Fig. 3(c) edge $(1, 5)$ is a tree edge, but it is redundant. A forward or cross edge with respect to one DFS tree can be a tree edge in a different DFS tree. We eliminate the redundant edges at a node by building a DFS tree that is rooted at that node. We are interested in a DFS tree where all the redundant edges of node 1 are forward edges of the tree.

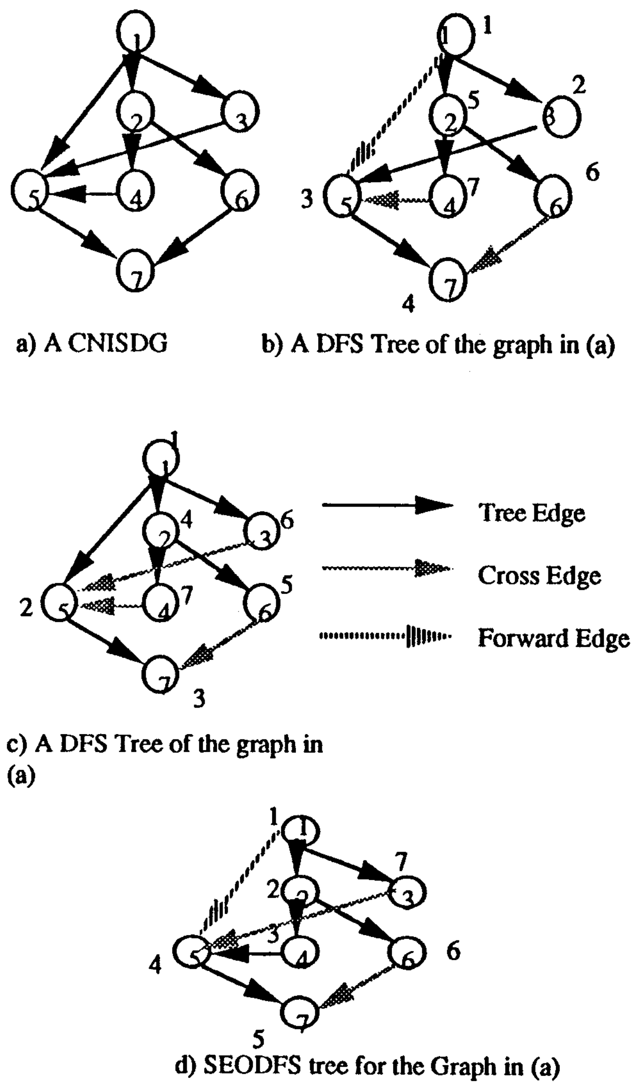

a) A CNISDG
b) A DFS Tree of the graph in (a)
Tree Edge
Cross Edge
Forward Edge
c) A DFS Tree of the graph in (a)
d) SEODFS tree for the Graph in (a)

Figure 3: An Example for Depth First Search

We use a variant of the DFS algorithm as given in Fig. 4 to find a DFS spanning tree rooted at node 1. This algorithm is very similar to the DFS algorithm, except that a node's sequential execution order is used as a tie-breaker in choosing an unvisited node from the adjacency list. Hence, we call this search Sequential Execution Order Depth First Search (SEODFS). By enforcing the SEO as a tie-breaker, we guarantee that all the paths from node 1 to node j through any node in the range 2 to $j-1$ are searched before selecting the direct path. After selecting an unvisited node from the adjacency list, the edges to the rest of the unvisited nodes are pushed onto the stack in inverse sequential execution order. This is illustrated by the example in Fig. 3. The graph in Fig. 3(a) has nine nodes. Application of SEODFS to this graph with node 1 as a start node, results in the tree shown in Fig. 3(d). After visiting node 1, the tie-breaker selects node 2 and pushes the edges $(1,5)$ followed by $(1,3)$ onto the stack. Similarly, at node 2, node 4 is selected while edge $(2,6)$ is pushed onto the stack. Node 5 is visited from node 4, and from there node 7 is visited. At this point, since node 7 has no unvisited adjacent nodes, the algorithm pops an edge (u,v) from the stack until it finds an unvisited node v, *i.e.*, edge $(2,6)$. At node 6, the algorithm finds that node 7 is already visited, and detects $(6,7)$ is a cross edge. It is not a forward edge because none of the descendants of node 6 has been visited yet. After visiting node 6, edge $(1,3)$ is popped from the stack and node 3 is visited. At node 3, edge $(3,5)$ is recognized as a cross edge. In the next step, edge $(1,5)$ is popped from the stack and detected as a forward edge. In other words, whenever an edge (u,v) is popped from the stack, if node v has already been visited, then (u,v) is a forward edge. The following lemma and theorem characterize the SEODFS algorithm.

Lemma 2 *If edge $(1,i)$ is redundant at node 1 then it is not a tree edge in the SEODFS tree rooted at node 1.*

Proof: Let $G(V,E)$ be a CNISDG. Since edge $(1,i)$ is redundant, there exists at least one path of length greater than 1 from node 1 to node i. Let V' be the set of all nodes in any path from node 1 to node i and k be the smallest of the nodes in V'. The definition of k implies that edge $(1,k) \in E$. The control-flow numbering of the nodes in a CNISDG implies that $1 < k < i$. Therefore, the SEODFS algorithm pushes edge $(1,i)$ before edge $(1,k)$ on to the stack. This implies that when edge $(1,i)$ is popped out of the stack node k is already visited. Hence, node i is visited before edge $(1,i)$ is popped out of the stack. Therefore, edge $(1,i)$ is not a tree edge. □

Theorem 1 *An edge $(1,i)$ of a CNISDG is redundant if and only if it is a forward edge with respect to the SEODFS tree rooted at node 1.*

Proof: From lemma 1 any forward edge is redundant. Lemma 2 implies that edge $(1,i)$ is not a tree edge in the SEODFS tree rooted at node 1. It can not be a cross edge because the source of the edge is the root node. Therefore, it has to be a tree edge. □

```
DO i := 1 to N
   visited[i] := false;
END
count := 0;
v := 1;
WHILE (count $\neq$ N) DO
BEGIN
   If visited[v] = false THEN
   BEGIN
      count := count + 1;
      visited[v] := true;
      Let S be the set of unvisited nodes in the
      adjacency list of v
      IF |S| $\neq$ 0 THEN
      BEGIN
         for all but the smallest u $\in$ S push $(v,u)$
         onto the stack in decreasing SEO of u.
         Set v to the smallest of S.
      END;
   END;
   ELSE IF stack is not empty Then
   REPEAT
      pop(stack,(u,v));
   UNTIL visited[v] is false;
END;
```

Figure 4: Sequential Execution Order Depth First Search Algorithm

In this section an approach to detect redundant edges at a node of a CNISDG has been presented. The complexity of the algorithm is O(E). This algorithm is useful in identifying redundant edges in a regular ISDG. In a simple and regular loops all the redundant dependences can be identified by applying this algorithm at node 1 of a subgraph of the corresponding ISDG. We present the construction of the subgraph in the next section.

4 Removal of Redundant Dependences in a Regular Simple Loop

When an ISDG is regular, it may be possible to analyze a subgraph of the ISDG and identify its redundant edges. In this section, we show that analysis of a subgraph of a regular ISDG called the R_subgraph can identify the redundant edges. The size of the R_subgraph is independent of the size of the iteration space, but dependent on the values of dependence distances.

Without loss of generality, we assume that the lower bound of the loop control variable is 1 and that there are m dependences. Let $\Delta = d_1, d_2, \ldots, d_m$ be the distances of these dependences. Since a dependence distance is given by the subtraction of the iteration number of a source of a dependence from that of a sink of the dependence, any d_i is positive. Let $d_{max} = MAX(\Delta)$ be the largest value in the set Δ. We now show that if a dependence edge from node 1 is redundant, then it is redundant at every node and vice versa.

Theorem 2 *In an ISDG, edge $(1, d_i + 1)$ is redundant if and only if the edge due to the dependence distance d_i is redundant at every node.*

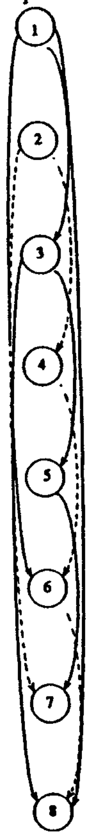
Proof:(Only if) Let the edge due to a dependence d_i be redundant at node 1. From the hypothesis, there exists a sequence of $d_{i_j} \in \Delta$, not necessarily distinct, such that $d_i = \sum_{j=1}^{l} d_{i_j}, l > 1$ and $1 + d_i \leq n$, the number of nodes. The partial sum s_k for $1 \leq k \leq l$ is defined as $s_k = \sum_{j=1}^{k} d_{i_j}$. Since the edge due to d_i is redundant from node 1 there is an alternate path at node 1 to node $d_i + 1$ given by $1 \rightarrow 1 + s_1 \rightarrow 1 + s_2 \rightarrow \ldots \rightarrow 1 + s_{l-1} \rightarrow 1 + d_i$.

Consider an arbitrary node $m \leq n$. If $m + d_i > n$, then the edge due to dependence d_i does not exist and is therefore redundant. If $m + d_i \leq n$, since the graph is regular and s_k is monotonically increasing, there is an alternate path from node m to $m + d_i$ given by $m \rightarrow m + s_1 \rightarrow m + s_2 \rightarrow \ldots \rightarrow m + s_{l-1} \rightarrow m + d_i$. That is edge $(m, m + d_i)$ is redundant. The proof of the if part is very similar to that of the only if part. □

From theorem 2, it is clear that in order to determine all the redundant edges in an ISDG, it is sufficient to determine the redundant edges at node 1. For this purpose we construct the R_subgraph $R(V', E')$. The set of vertices V' is $\{1, 2, \ldots, d_{max} + 1\}$ and for every $1 \leq i < j \leq (d_{max} + 1)$, $(i, j) \in E'$ if and only if there exists a d_k such that $(i + d_k)$ is equal to j. In other words, the R_subgraph of an ISDG is the *induced subgraph*[4] on vertices $\{1, 2, \ldots, d_{max} + 1\}$ of the ISDG. This is illustrated through an example in Fig. 5. The loop in Fig. 5(a) has three inter-iteration dependences with distances of 2, 5, and 7. The R_subgraph corresponding to this loop has eight nodes. The connectivity of this subgraph is shown in Fig. 5(b). We show that an edge is redundant at node 1 of the R_subgraph of an ISDG if and only if it is redundant in the ISDG.

```
DO i := 1 to 100
   A[i] := A[i-2] + A[i+5] + A[i-7];
END
```

a) A One-D Do Loop with Constant Dependences



b)R_subgraph of the loop in (a)

Figure 5: Illustration of ISDG

Theorem 3 *An edge from node 1 of an ISDG $G(V, E)$ is redundant if and only if it is also redundant in the corresponding R_subgraph $G'(V', E')$.*

Proof:(Only if) Let edge $(1, i)$ be redundant in G and not redundant in G'. Since $i \leq d_m + 1$, and V' includes all the nodes in the range $1 \ldots d_m + 1$, i belongs to V'. Edge $(1, i)$ is not redundant in G' implies that there exists a path $1 \rightarrow v_1 \rightarrow v_2 \rightarrow \ldots v_j \rightarrow i$, where $j > 0$ and $1 < v_k < i$ for all $1 \leq k \leq j$, in G but not in G'. This is not possible because G' is an induced subgraph of G and all the nodes in the range $1 \ldots i$ belong to V'. Hence, any edge of the form $(1, i)$ redundant in G is also redundant in G'. The if part is true because $V' \subseteq V$ and

$E' \subseteq E$. □

In order to find redundant edges of a regular one dimensional loop, we construct an R_subgraph using the above procedure and find the redundant edges at node 1 of it using the algorithm described in section 3. The complexity of the algorithm is $O(m * d_{max})$. Midkiff and Padua use an algorithm [10] based on the transitive closure of an adjacency matrix. The complexity of their algorithm is of the order $m * d_{max}^3$. Moreover, Midkiff and Padua consider only one dimensional loops. In the next section we show the applicability of our scheme to two dimensional loops

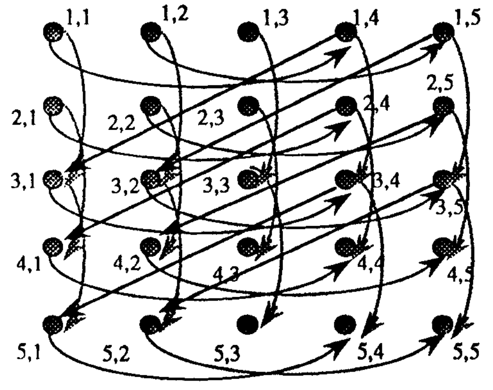
5 Removal of Redundant Dependences in a Regular 2-D Loop

Without loss of generality we assume that the lower bound of the loop control variables of both loops is one. First, we will consider loops that have rectangular iteration space, that is the upper bound of the inner loop is independent of the iteration number of the outer loop. Unlike simple loops, redundancy of a dependence is not ***uniform*** in the iteration space of doubly nested loops. For instance, consider dependence vectors $(0,15)^T, (1,-13)^T$, and $(1,2)^T$. At point $(1,1)$ the dependence $(1,2)$ is redundant, but if the width of the rectangular iteration space is less than twenty five units then the dependence $(1,2)^T$ is not redundant at point $(1,10)$. The presence of negative components in dependence vectors constrains the uniformity of redundancy in an ISDG. In this section we present an R_subgraph for rectangular loops, and prove that we can identify all the redundant dependences. The applicability of this scheme to triangular loops is also discussed.

The dependence distances of two dimensional loops are vectors. The dependence vectors of the loop shown in Fig. 6 are $(2,0)^T, (0,3)^T$, and $(2,-3)^T$. The dependence vector $(2,0)^T$ is due to a flow dependence from iteration (i,j) to iteration $(i+2,j)$. Similarly, the dependence vector $(0,3)^T$ signifies a flow dependence from iteration (i,j) to iteration $(i,j+3)$. The dependence vector $(2,-3)^T$ is due to an anti-dependence from iteration (i,j) to iteration $(i+2,j-3)$. The first component of a dependence vector corresponds to the outer loop and the second to the inner loop. In the case of nested loops, the component of an inner loop in a dependence vector can be negative, because with respect to a particular inner loop, the source of a dependence can potentially be numbered higher than the sink of the dependence. In any case, the first non-zero component of a dependence vector is positive.

```
DO i := 1 to 5
   DO j := 1 to 5
      A[i,j] := A[i-2,j] + A[i+2,j-3] + A[i,j-3];
   END
END
```

a) A Two-D Loop with Constant Dependences



b) ISDG for the Loop in (a)

Figure 6: Illustration of Dependence Vectors Iteration Space for a Two Dimensional Loop

The nodes of an ISDG in a doubly nested loop can be viewed as points in a two dimensional space. The node corresponding to iteration vector (i,j) is represented by the point (i,j) in two dimensional space. From a different perspective, point (i,j) can be viewed as a vector $(i,j)^T$ at the origin of the coordinate system. This provides a uniform perception for points in the iteration space and the dependence vectors, and is useful in proving some results. Henceforth, we use point and vector interchangeably. In an ISDG, if there is an edge from point p_1 to point p_2 then $P_2 = P_1 + d_i$, where P_1 and P_2 are vectors representing points p_1 and p_2, respectively, and d_i is the dependence vector corresponding to the edge. We capture the spatial relation between any two points in a rectangular iteration space with the following definitions.

- The relation $p_1 \trianglelefteq p_2$ between two points p_1 and p_2 is true, if and only if $P_{11} \leq P_{21}$ and $P_{12} \leq P_{22}$, where $(P_{11}, P_{12})^T$ and $(P_{21}, P_{22})^T$ are vectors representing points p_1 and p_2 respectively.
- The relation $p_1 \trianglerighteq p_2$ between two points p_1 and p_2 is true if and only if $P_{11} \geq P_{21}$ and $P_{12} \geq P_{22}$, where $(P_{11}, P_{12})^T$ and $(P_{21}, P_{22})^T$ are vectors representing

the points p_1 and p_2 respectively.

In Fig. 6(b), any point in the grid is $\trianglerighteq$ than $(1,1)^T$ and $\trianglelefteq$ than $(5,5)$. In other words, a point p in the grid is characterized by $p \trianglerighteq first$ and $p \trianglelefteq last$, where *first* and *last*, respectively, be the points representing the first and last iterations of a rectangular loop.

We redefine a redundant dependence of a two dimensional rectangular loop as follows: Let $\Delta = \{d_1, d_2, \ldots, d_m\}$ be the set of dependence vectors. An outgoing edge due to dependence $d_i, 1 \le i \le m$ is redundant at point p_1, if and only if

1. $d_i = d_{i_1} + d_{i_2} + \ldots + d_{i_n}$, where $d_{i_j} \in \Delta$ and $d_i \neq d_{i_j}$ for all $1 \le j \le n$.
2. $P_1 + \sum_{k=1}^{j} d_{i_k} \trianglerighteq first$ for all $1 \le j \le n$.
3. $P_1 + \sum_{k=1}^{j} d_{i_k} \trianglelefteq last$ for all $1 \le j \le n$.

The first condition states that there is an alternate path in a regular graph of unbounded size. Conditions two and three guarantee that the path is in the rectangular grid. Let us consider the following dependence vectors $(0,7)^T, (1,-11)^T$, and $(3,2)^T$. The dependence vector $(3,2)^T$ is redundant in an unbounded ISDG. However, it is not redundant if the number iterations in the inner loop is equal to twelve because conditions two and three are not satisfied.

We have proved that if an edge is redundant at a point of the ISDG of a one dimensional loop then it is redundant at every other node. However, this is not true for an ISDG of a two dimensional loop. Now we claim th at if all the components of any dependence vector are non-negative and if a dependence vector is redundant at any point then it is redundant at every other point.

Theorem 4 *If none of the dependence vectors of a 2-D rectangular loop has negative components then the redundancy of a dependence vector at any point of the ISDG implies its redundancy at all points of the ISDG.*

Proof: Let dependence $d_i = d_{i_1} + d_{i_2} + \ldots + d_{i_n}$ at point p_1 be redundant. This implies that all the three conditions of redundancy are true at point p_1. Let $s_j = \sum_{k=1}^{j} d_{i_k}, 1 \le j \le n$. Since all the components of any d_{i_j} are positive, $s_p \trianglelefteq s_q, 1 \le p < q \le n$. Condition two is true at every point, p_2, because $p_2 \trianglelefteq P_2 + s_j, 1 \le j \le n$ and $p_2 \trianglerighteq first$. Condition three is true because $P_2 + s_n \trianglelefteq last$ and $P_2 + s_j \trianglelefteq P_2 + s_n$. Hence, redundancy is uniform at every point in the grid. □

When some of the components of dependence vectors are negative redundancy is uniform in an unbounded space. We have shown an example of bounded space, where redundancy is not uniform. Next we present a sufficient condition for the uniformity of a redundancy in a bounded space. The following notation is used in presenting it. A dependence vector d_i is denoted by its components $(c_{i1}, c_{i2})^T$. Let, $\Delta = \{d_1, d_2, \ldots, d_m\}$ be the set of dependence vectors. We define p_{max} as $Max(Max(\{c_{i2}| \ 1 \le i \le m\}), 0)$. That is, p_{max} is equal to zero if the second component of all the dependence vectors is negative, otherwise, it is the maximum of the second component of all the dependences. Similarly, n_{min} is defined as $Min(Min(\{c_{i2}| \ 1 \le i \le m\}), 0)$. That is, n_{min} is equal to zero if the second component of all the dependence vectors is positive, otherwise, it is the minimum of the second component of all the dependences. We define an_{min} as the absolute value of a n_{min}. In a doubly nested loop if the number of iterations in the innermost loop is greater than or equal to the sum of p_{max} and an_{min} then redundancy is uniform throughout the iteration space. Before proving this we prove the following lemma.

Lemma 3 *Let p_1 be a point in a rectangular iteration space of width $p_{max} + an_{min}$ and $s_i = \sum_{k=1}^{n} e_{i_k}$, where $n \ge 1$ and $e_{i_k} \in \Delta$ not necessarily distinct. From any point within the rectangular grid, if the adjacent point due to s_i is within the grid then there exists a path of length n given by a sequence of adjacent points in the grid that corresponds to some permutation of $e'_{i_k}s$.*

Proof: We prove this using induction on the length of the path, that is the number of $e'_{i_k}s$. Let p_2 be the point given by $P_1 + s_i$.
Basis: When k is equal to one, s_i is equal to e_{i_1}. Since k is equal to one there is only one permutation and the path is given by p_1, p_2.
Hypothesis: Let us assume that for any $s_i = \sum_{k=1}^{l} e_{i_k}, l \ge 1$ there exists a path from p_1 to p_2 given by a sequence of adjacent points in the grid and that this sequence is determined by a permutation of $e_{i_k}s$.
Inductive step: Let us consider the case $s_i = \sum_{k=1}^{l+1} e_{i_k}, l \ge 1$
Case1: All the c_{i_k2}s are positive. Since p_2 is in the grid, $\sum_{k=1}^{l+1} c_{i_k2} \le width$ This implies that there exists a point p_3 in the grid and a e_{i_j} such that $P_3 + e_{i_j} = P_2$. Since vector addition is commutative and the length of the path from p_1 to p_3 is l, by induction hypothesis there is a path of adjacent points from p_1 to p_2 in the grid.
Case2: All the c_{i_k2}s are negative. Proof is very similar to case one.
Case3: There is at least one positive c_{i_k2} and one nega-

tive $c_{i_k 2}$. Let $P_2 = (cP_{21}, cP_{22})^T$ is the vector representing point p_2. Suppose $cP_{22} \leq p_{max}$, we can find a e_{i_k} and a point p_3 such that $P_3 + e_{i_k}$ is equal to P_2 and $e_{i_k 2} < 0$. The $cP_{32} \leq p_{max} + an_{min}$ because $cP_{22} \leq p_{max}$ and the an_{min} is less than the width of the grid. The point p_3 is in the grid because $cP_{11} > 0$ and $cP_{21} \geq cP_{31}$. From the induction hypothesis there is a path of adjacent points within the grid from p_1 to p_2 and it corresponds to a permutation of $e_{i_k}s$.

When $cP_{22} > p_{max}$ we can find a e_{i_k} and a point p_3 such that $P_3 + e_{i_k}$ is equal to P_2 and $e_{i_k 2} > 0$. The $cP_{32} > 1$ because $cP_{22} > p_{max}$. The point p_3 is in the grid because $cP_{11} > 0$ and $cP_{21} \geq cP_{31}$. From the induction hypothesis there is a path of adjacent points within the grid from p_1 to p_2 and it corresponds to a permutation of $e'_{i_k}s$. □

We show that when the width of the rectangular iteration space is greater than or equal to the sum of p_{max} and an_{min}, if an edge is redundant at a point in an ISDG then it is redundant at every other point.

Theorem 5 *If a dependence edge is redundant at a point of the ISDG of a rectangular two dimensional loop and the width of the iteration space is greater than $p_{max} + an_{min}$, then it is redundant at all points.*

Proof: Let us assume that the dependence d_i is redundant at point p_1. Let p_2 be an arbitrary point. Let P_1 and P_2 be the vectors representing the points p_1 and p_2 respectively.
Case1: $P_2 + d_i \trianglerighteq$ *last*
This implies that the edge corresponding to the dependence d_i does not exist at point p_2. Hence, the theorem is vacuously true.
Case2: $P_2 + d_i \trianglelefteq$ *first*
This implies that the edge corresponding to the dependence d_i does not exist at point p_2. Hence, the theorem is vacuously true.
Case3: $P_2 + d_i \trianglerighteq$ *first* and $P_2 + d_i \trianglelefteq$ *last*
The condition one for redundancy is vacuously true. Let p_3 be the adjacent point of p_2 by distance d_i. Since the width of the iteration space is greater than or equal to sum of p_{max} and an_{min}, we can always find a rectangular grid, region of width equal to sum of p_{max} and an_{min} enclosing p_2 and p_3. From Lemma 3 there exists an alternate path in the selected region from point p_2 to point p_3. Hence, conditions two and three of redundancy are satisfied. Therefore the edge due to dependence d_i is redundant at any point p_2. □

Let $w = p_{max} + an_{min}$ and $l = Max(\{d_{i1} | 1 \leq i \leq m\})$. The R_subgraph of a two dimensional rectangular loop of width greater than or equal to w is given by the induced subgraph over the nodes in the iteration space given by 1 to $1 + w$ along the inner loop and 1 to $l + 1$ along outer loop. We apply SEODFS algorithm at the node with index vector $(1, (1 + an_{min}))$. Since all the dependence vectors are defined at this point, any redundant dependence vector is detected by SEODFS algorithm. From the above theorem any edge that is redundant in the R_subgraph is also redundant in the original graph. We define a precedence relation to be used as a tie-breaker by the SEODFS algorithm. Node $p_1 = (i, j)^T$ precedes node $p_2 = (k, l)^T$ if and only if either $i < k$ or $i = k \wedge j < l$. This precedence relation provides a total order among nodes and this order is same as SEO.

We have presented a scheme to determine redundant edges for rectangular loops. It is possible to extend this result to trapezoidal loops. In trapezoidal loops, redundancy is uniform only over subspaces. In general, in two dimensional loops a dependence is redundant at a point if - a) there is a rectangular region of width $(w + 1)$ enclosing the source and the sink of the dependence in which all the discrete points in the space belong to iteration space, b) the dependence vector is redundant in the R_subgraph.

6 Discussion and Conclusions

We have presented an efficient algorithm for detecting redundant edges when there are constant dependences. Previous work considers only simple loops, whereas we have investigated and characterized doubly nested loops. These results can also be extended to higher dimensions. In the case of higher dimensions, if all the components of dependence vectors are positive, then redundancy is uniform. If any negative component is present, redundancy is not uniform. This algorithm can be implemented as a phase in parallelizing compilers such as Parafrase [7], PTRAN [2], and PFC [3]. The reduction in parallel execution time is dependent on the type of synchronization primitives available and the scheduling scheme.

We have presented our scheme using ISDG. One of the technique proposed for execution of loops on a distributed memory computers is iteration space tiling. The tile size is dependent on the dependences. Elimination of redundant dependences in the ISDG can potentially reduce the tile size and increase the concurrency. This scheme as presented here, is also applicable for pre-synchronized scheduling [6, 12] on shared memory multi-processors. If a shared memory multi-processor uses self-scheduling then we need to apply this algorithm to CPG.

References

[1] A.V. Aho, M.R. Garey, and J.D. Ullman. The Transitive Reduction of a Directed Graph. *SIAM Journal of Computing* , June 1972, pp.131-137.

[2] F. Allen, M. Burke, P. Charles, R. Cytron, and J. Ferrante An Overview of the PTRAN Analysis System for Multiprocessing. *Journal of Parallel and Distributed Computing.*, Vol. 5, No. 5, Oct 1988, pp.617-640.

[3] J.R. Allen and K. Kennedy. PFC: A Program to Convert FORTRAN to Parallel Form. *Proceedings of the IBM Conference on Parallel Computers and Scientific Computations*, March 1982.

[4] Shimon Even *Graph Algorithms.* Computer Science Press, 1979.

[5] Z.Fang, P. Yew, P. Tang, and C. Zhu Dynamic Processor Self-Scheduling for General Parallel Nested Loops. *Int. Conf. on Parallel Processing*, 1987, pp.1-10.

[6] V.P. Krothapalli and P. Sadayappan. Dynamic Scheduling of DOACROSS Loops, *Proc. Parbase-90 International Conference on Databases and Parallel Architectures*, 1990.

[7] D.J. Kuck. et. al. Dependence Graphs and Compiler Optimizations. In *Proc. ACM 8th Annual Symposium on Programming Languages*, 1981, pp.207-218.

[8] D.J. Kuck. et. al. Parallel Supercomputing Today and the CEDAR Approach. *Science*, Vol. 231, No. 2, 1986, pp.967-974.

[9] Zhiyuan Li and Abu-Sufah On Reducing Data Synchronization in Multiprocessed Loops *IEEE Transactions on Computers*, Vol. C-36, No. 1, 1987, pp. 105-109.

[10] S.P. Midkiff and D.A. Padua. Compiler Algorithm for Synchronization. *IEEE Trans. Comput.*, Vol. C-36, No. 12, 1987, pp.1485-1495,

[11] C.D.Polychronopoulos. and D.Kuck. Guided Self-Scheduling: A Practical Scheduling Scheme for Parallel Supercomputers. *IEEE Transactions on Computers*, Vol. C-36, No. 12, 1987, pp.1425-1439.

[12] P. Tang, P. Yew, Z. Fang, and C. Zhu Deadlock Prevention in Processor Self-Scheduling for Parallel Nested Loops. *Int. Conf. on Parallel Processing*, 1987, pp.11-18.

[13] R. Tarjan. Depth First Search and Linear Graph Algorithms. *SIAM Journal of Computing*, 1(2), 1972, pp.146-160.

[14] M.J. Wolfe. *Techniques for Improving the Inherent Parallelism in Programs.* Master's thesis, University of Illinois, Urbana-Champaign, July 1978.

[15] M.J. Wolfe. *Optimizing Supercompilers for Supercomputers.* The MIT Press, Cambridge, 1988

[16] C.Q. Zhu and P.C. Yew. A Scheme to Enforce Data Dependencies on Large Multiprocessor Systems. *IEEE Transactions on Software Engineering*, Vol. SE-13, No.6, 1987, pp.726- 739.

[17] C.Q. Zhu and P.C. Yew. A Synchronization Scheme and its Applications for Large Multiprocessor Systems. *Proc. 4th Int. Conf. on Distributed Computing Systems*, 1984, pp.486-493.