Statement-Level Independent Partitioning of Uniform Recurrences

J. Ramanujam* S. Vasanthakumar

Abstract

This papers presents an approach to statement-level independent partitioning of uniform recurrences, *i.e.,* loops with constant dependence distance vectors. Uniform recurrences may be partitioned into independent subsets—of the set of all statement instances—that require no communication or synchronization on a multiprocessor. Therefore, independent partitioning is highly efficient and desirable. This paper presents a method to partition uniform recurrences using statement-level affine schedules, and an algorithm for code generation. We consider a statement instance as a basic unit that can be allocated to a processor, in contrast to existing methods that use an iteration instance. Using this approach, we not only find maximal independent partitions for problems with a single statement inside a perfectly nested loop, but also solve the partitioning problem for multiple statements in a single or multiple nested loops. The method uses schedules derived using linear programming, which is very fast in practice.

1 Introduction

In recent years, much research has been done in the area of recognizing and transforming sequential code into parallel code suitable for a wide variety of parallel architectures [8, 9]. Sophisticated compiler techniques recognize parallelism through data flow analysis and program restructuring. Several factors such as communication, synchronization, and load balancing affect the performance of the loop in addition to available parallelism.

The speedup that can be achieved by parallel computers depends on two conflicting assumptions, namely, that many computations take place concurrently and that the time spent in data communication between these computational blocks is negligible. In order to reduce communication and synchronization overheads, it is thus desirable to identify *independent* computational blocks—if there are any—of a program. By independent computational blocks, we mean blocks that do not require any communication between them. With this, not only is there no time spent on data exchanges, but also there are no synchronization constraints between the computational blocks.

But practically, it not possible to find such partitions for all programs. Several partitioning algorithms have been developed in recent years to achieve the above mentioned goal [1, 3, 7]. All the existing partitioning algorithms consider an iteration as a basic schedulable unit. When there are multiple statements in the body of a loop, the number of independent blocks found by using methods in [1, 3, 7] is limited. Therefore, we consider a statement as a basic schedulable unit. The statement-level approach is shown to result in a larger number of independent partitions than extant methods. We present a partitioning algorithm which first models the problem as a *statement-level rational affine scheduling* problem [4]. Then, the schedules obtained are utilized to identify the number of partitions and to generate parallel code. We model the scheduling problem as a *Linear Programming (LP)* problem. There are several advantages of this method over the existing ones with respect to number of partitions attainable, simplicity of implementation, and greater applicability to several cases.

This paper is organized as follows. In Section 2, we present the background necessary and discuss related work. Section 3 presents an LP formulation of the scheduling problem, and shows how to identify independent partitions using the generated schedules. Section 4 describes an efficient partitioning algorithm for multi-statement nested loops with loop-carried dependences. Several examples are used Section 5 to illustrate the the use of this technique. Section 6 concludes with a summary and discussion.

2 Terminology and Related Work

A dependence vector is associated with each dependence, where the distance represents the number of iterations between the source and sink of the dependence. If each entry in the dependence vector is a fixed integer (independent of the source of partition), then these are called distance vectors. If all the entries of a distance vector are zeros, then the dependence is *loop-independent*; otherwise, it is *loop-carried.* Problems in which all dependence vectors are distance vectors are referred to as *uniform recurrences* [7]. We use the term "uniform recurrence" and the term "nested loops with constant distance vector" interchangeably in this paper. For details on dependences, see [8, 9]. In this paper, we focus on a class of problems that are fully parallelizable.

2.1 Dependence Graph (DG)

Dependence graphs are used to depict statement-level dependence relations. The dependence relations for a perfectly n-nested loop with r statements $(S_1, \ldots, S_r)$, are represented by a dependence graph G with r nodes, one for each statement S_k, $k \in [1 \ldots r]$. For each dependence relation from statement S_x to statement S_y, there exists a directed edge from node representing S_x to the node representing S_y, labeled with the corresponding distance vector. A dependence from a node to itself is called a *self dependence*. The dependence graph for Example 1 is as shown in Figure 1.

Example 1:

do $I_1 = 1, N$
  do $I_2 = 1, N$
$S_1:$ $A[I_1, I_2] = B[I_1 - 1, I_2 - 3] + D[I_1 - 1, I_2 + 3]$
$S_2:$ $B[I_1, I_2] = C[I_1 - 1, I_2 + 4] + X[I_1, I_2]$
$S_3:$ $C[I_1, I_2] = A[I_1, I_2 - 2] + Y[I_1, I_2]$
$S_4:$ $D[I_1, I_2] = A[I_1, I_2 - 1] + Z[I_1, I_2]$

An Iteration Space Graph (ISG) represents data dependence relations at a coarser level. The nodes of an ISG are iterations which, for an n-nested loop, are points of an n-dimensional Cartesian space. A directed edge from iteration $\vec{\imath}$ to iteration $\vec{\jmath}$ denotes a dependence from some statement in iteration $\vec{\imath}$ to some statement in iteration $\vec{\jmath}$.

2.2 Banerjee's Inequalities

Banerjee's inequalities [8] are extensively used in data dependence testing in programs. We use Banerjee's inequalities in modeling the scheduling problem as an LP problem [4] (discussed in Section 3.1) and in the partitioning algorithm (discussed in Section 4). The

*Department of Electrical and Computer Engineering, Louisiana State University, Baton Rouge, LA 70803. (jxr@gate.ee.lsu.edu). Supported in part by an NSF Young Investigator Award CCR–9457768, an NSF grant CCR–9210422, and by the Louisiana Board of Regents through contract LEQSF (1991-94)-RD-A-09.

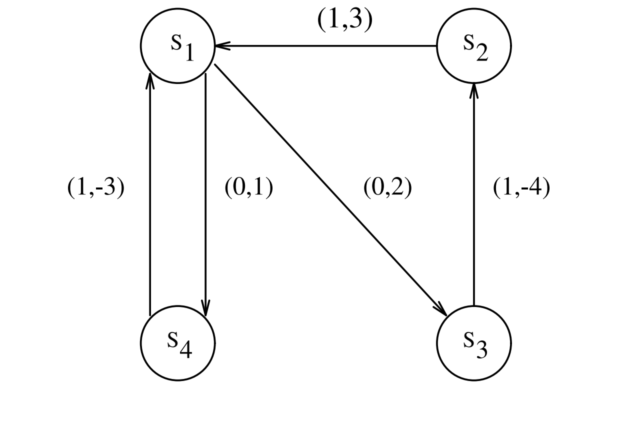

Figure 1: Dependence Graph for Example 1

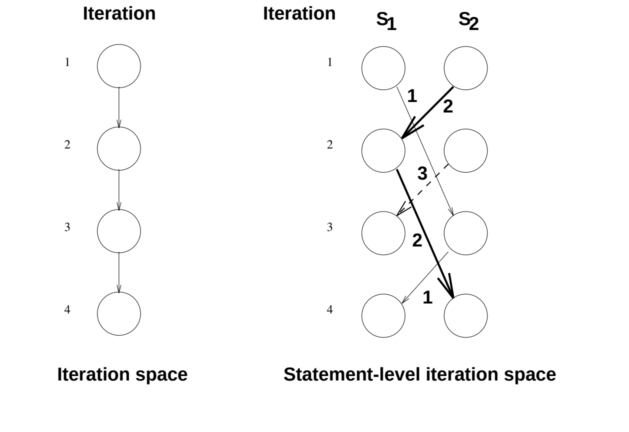

Figure 2: Difference Between Hollander's and Our Method

positive part a^+ and the *negative part* a^- of a real number a are defined as follows:

$$a^+ = \begin{cases} a & \text{if } a \geq 0 \\ 0 & \text{if } a < 0 \end{cases} \qquad a^- = \begin{cases} -a & \text{if } a \leq 0 \\ 0 & \text{if } a > 0 \end{cases}$$

The main properties of positive and negative parts of a number are listed below [8]:

$$a^+ \geq 0 \; a^- \geq 0 \qquad a = a^+ - a^-$$
$$|a| = a^+ + a^- \qquad a^- \leq a \leq a^+$$

2.3 Independent Partition

An *independent partition* is defined [7] as follows: Given an algorithm $(\mathcal{I}, D)$ where $\mathcal{I}$ is the index set and D is the iteration space graph $(\mathcal{I}, E)$, let $p = \{\mathcal{I}_1, \mathcal{I}_2, \ldots, \mathcal{I}_q\}$, (where $q \in N^+$), be a partition of $\mathcal{I}$. If for any arbitrary points, $\vec{i_1} \in \mathcal{I}_k$ and $\vec{i_2} \in \mathcal{I}_l$, $k \neq l$ and $0 < k, l \leq q$, the edges $(\vec{i_1}, \vec{i_2}) \notin E$, then p is an independent partition of the algorithm $(\mathcal{I}, D)$. The sets $\mathcal{I}_j, j = 1, \ldots, q$, are called the *blocks* of the partition p. For an independent partition p, if any two arbitrary points $\vec{i_i}, \vec{i_2} \in \mathcal{I}_k, k = 1, \ldots, q$, are connected in the iteration space graph, then p is a *maximal independent partition* of $(\mathcal{I}, D)$ and is denoted by $p_{\max}$. The number of *blocks* in the independent partition p is called the *cardinality* of the partition p. The cardinality of the maximal independent partition $p_{\max}$ is called the *partitionability* of the algorithm $(\mathcal{I}, D)$.

2.4 Related Work

Several partitioning algorithms [1, 3, 7] have been proposed in recent years gradually increasing the cardinality of the partition (number of synchronization-free computational blocks) and simplifying the algorithm to do so. All these methods work at the iteration level; they use linear algebraic operations on the dependence distance matrix to determine the maximum number of partitions that can be derived. Of these, the method of Hollander [1] is the most general and generates the highest cardinality of partitions at the iteration-level. Peir and Cytron [3] also introduce a method to handle multiple statements. They reduce an r-statement recurrence into a single statement recurrence by solving for just one of the variables, and the other assigned variables derived later from the recurrence solution for the single variable. A code conversion algorithm is given to generate the loop limits for the *doall* and *do* loops. The code conversion algorithm uses integer programming problem and hence it is computationally expensive. This method also requires reordering of statements in the body of the loop when handling multiple statement problems.

In contrast, the method proposed in Section 3 uses a single framework to solve problems having dependence matrix with or without linearly dependent dependence vectors. In addition, our method can handle multiple statements, and indeed, can detect higher partitionability in such cases. This method cannot find maximal independent partition for cases in which the dimensionality of the dependence matrix is less than the nesting level of the loop. In order to handle such cases, the method uses different values, by trial and error, to obtain a maximal partition.

3 Independent Partitioning of Loops

This section presents a precise algorithm to partition uniform recurrences. We use the affine scheduling method presented in [4] to first obtain an optimal schedule for the given program. Then the schedule obtained is utilized to identify the partitions and to generate the parallel code. The scheduling algorithm of [4] generates optimal schedules suitable for software pipelining. We make use of the above algorithm to assist in identifying and generating independent partitions. The problem of scheduling nested loops at the statement level is formulated as an LP problem, which admits polynomial time solutions and is extremely fast in practice. Also, since statements are used as computational units, instead of iterations, this method generates higher cardinality of partition than the methods in [1, 3, 7]. The advantage of using statements as units to generate more partitions is illustrated using Example 2.

Example 2:
do $i = 1$ to 4 do
S_1: $a[i] = b[i-1] + c[i]$
S_2: $b[i] = a[i-2]$

Figure 2 shows the effect of representing the above problem in iteration space and in statement-level iteration space. As is evident from the figure, our method has a clear advantage since it recognizes three partitions (the three partitions are labeled 1, 2 and 3 in Figure 2) in contrast to one generated by Hollander's method [1].

3.1 Statement-Level Affine Scheduling

The basis for this method is the hyperplane scheduling technique developed by Lamport [2]. We discuss the formulation in [4] next. Consider an n-level perfectly nested loop with r statements in the body of the loop. Let $t_k(\vec{i})$ denote the time at which statement S_k, $(k = 1, ..., r)$ in iteration $\vec{i} = \{i_1, i_2, ..., i_n\}$ is executed. We assume that the execution time for each statement is one. Now $t_k(\vec{\imath})$ can be expressed as a rational function,

$$t_k(\vec{\imath}) = h_1 i_1 + h_2 i_2 + ... + h_n i_n + \delta_k = \vec{h} \cdot \vec{\imath} + \delta_k$$

The schedule is a tuple $< \vec{h}, \vec{\delta} >$ where $\vec{h} = (h_1, h_2, ..., h_n)$ is a rational n-vector and $\vec{\delta} = (\delta_1, \delta_2, ..., \delta_r)$ is a rational r-vector. In order for the schedule $< \vec{h}, \vec{\delta} >$ to be legal, it should satisfy all the dependences, *i.e.,* if statement S_q in iteration $\vec{\imath}$ depends on statement S_p in iteration $(\vec{\imath} - \vec{d^{p,q}})$, then statement S_q in iteration $\vec{\imath}$ can be scheduled only after statement S_p in iteration $(\vec{\imath} - \vec{d^{p,q}})$. In other words,

$$\begin{aligned} & t_q(\vec{\imath}) \geq t_p(\vec{\imath} - \vec{d^{p,q}}) + 1 \\ \Longrightarrow \quad & \vec{h} \cdot \vec{\imath} + \delta_q \geq \vec{h} \cdot (\vec{\imath} - \vec{d^{p,q}}) + \delta_p + 1 \\ & \text{or,} \quad \vec{h} \cdot \vec{d^{p,q}} \geq \delta_p - \delta_q + 1 \end{aligned}$$

Thus the execution time E for the schedule is

$$E = \max(t_p(\vec{\imath}) - t_q(\vec{\jmath})) + 1 \qquad (\forall \vec{\imath}, \vec{\jmath} \in \mathcal{I}) \wedge (\forall p, q \in [1 \dots r])$$

where $\mathcal{I}$ is the set of all the possible iterations. The optimal execution time is the minimum value of the expression E.

$$E \approx \max(\vec{h} \cdot (\vec{\imath} - \vec{\jmath})) + \max(\delta_k) - \min(\delta_k)$$

In practice, the number of iterations is very large and thus E can be written as

$$E \approx \max(\vec{h} \cdot \vec{\imath}) - \min(\vec{h} \cdot \vec{\jmath}) \qquad \forall \vec{\imath}, \vec{\jmath} \in \mathcal{I}$$

We assume a rectangular iteration space, *i.e.,* the lower and upper limits, l_i and u_i, $i \in [1 \dots n]$, are integer constants. Now the optimal value for E can be derived using Banerjee's inequalities:

$$\begin{aligned} \max_{\vec{\imath} \in \mathcal{I}}(\vec{h}.\vec{\imath}) &= \sum_{i=1}^{n}(h_i^+ u_i - h_i^- l_i) \\ \min_{\vec{\imath} \in \mathcal{I}}(\vec{h}.\vec{\imath}) &= \sum_{i=1}^{n}(h_i^+ l_i - h_i^- u_i) \end{aligned}$$

$$E \approx \sum_{i=1}^{n}\{(h_i^+ u_i - h_i^- l_i) - (h_i^+ l_i - h_i^- u_i)\}$$

$$E \approx \sum_{i=1}^{n}\{(h_i^+ + h_i^-)(u_i - l_i)\} = \sum_{i=1}^{n}\{|h_i|(u_i - l_i)\}$$

Thus we can observe that the optimal value of the expression E is a function of $\sum_{i=1}^{n} |h_i|$. The problem reduces to minimizing the execution time E subject to dependence constraints:

$$\text{Minimize} \sum_{i=1}^{n} |h_i| \quad \text{subject to the constraints:}$$

$$\vec{h} \cdot \vec{d^{p,q}} \geq \delta_p - \delta_q + 1 \qquad p, q \in [1 \dots r]$$

for all the dependences *i.e.,* all the edges in graph G.

In order to solve the above problem, it is translated into an LP problem. For a problem to be solved using LP, the objective function and constraints are required to be linear functions of non-negative variables [6]. But our formulation has the following problems that need to be addressed before using LP: (a) The variables in the constraints, h_i, $i = 1, \dots, n$ and δ_i, $i = 1, \dots, r$ are not always non-negative; and (b) The objective function $\sum_{i=1}^{n} |h_i|$ has absolute value of variables. Each variable h_i, $i = 1, \dots, n$ and δ_i, $i = 1, \dots, r$ is replaced by the difference of two non-negative variables, *i.e.,* h_i is replaced with $h_{i,+} - h_{i,-}$, where $h_{i,+} \geq 0$ and $h_{i,-} \geq 0$ [6]. The second deficiency is handled by adding a set of variables θ_j, $j = 1, \dots, n$. By this, the new objective function is $\sum_{i=1}^{n} \theta_i$, with constraints, $\theta_i - h_i \geq 0$ and $\theta_i + h_i \geq 0$. Now the problem is in standard LP form as shown below:

$$\text{Minimize} \sum_{i=1}^{n} \theta_i \quad \text{subject to}$$

$$\begin{aligned} \theta_i - h_{i,+} + h_{i,-} \geq 0 \qquad & \forall i \in \{1, n\} \\ \theta_i + h_{i,+} - h_{i,-} \geq 0 \qquad & \forall i \in \{1, n\} \end{aligned}$$

$$\sum_{i=1}^{n}\{(h_{i,+} - h_{i,-})d_i^{p,q}\} - \delta_{p,+} + \delta_{p,-} + \delta_{q,+} - \delta_{q,-} \geq 1$$

where $p, q \in [1 \dots r] \wedge (p, q) \in \text{edges}(G)$. If we have m dependence vectors in a n-nested loop, then we have $2n + m$ number of constraints. By solving the above LP problem, solutions for the variables $h_i, i = 1, \dots, n$, and $\delta_i, i = 1, \dots, r$, are obtained.

3.2 Identifying Independent Partitions

Our goal now is to first calculate the number of partitions (cardinality of the partition) that can be generated. At this point, we have the optimum schedule for the given nested loop. The time at which each statement S_k, $k = [1 \dots r]$ of any iteration $\vec{\imath}$ where $\vec{\imath} = \{i_1, i_2, ..., i_n\}$ is scheduled is given by

$$t_k(\vec{\imath}) = \vec{h} \cdot \vec{\imath} + \delta_k \qquad k = [1...r].$$

Let each element in $\vec{h}$ be of the form $h_i = \frac{a_i}{b_i}$ where a_i and b_i are integers and $\gcd(a_i, b_i) = 1$. Let each element in $\vec{\delta}$ be of the form $\delta_i = \frac{c_i}{d_i}$ where c_i and d_i are also integers and $\gcd(c_i, d_i) = 1$.

The schedule ensures that all the statement instances that are scheduled for a particular $\lfloor t_k \rfloor$ do not depend on each other [4] and thus may belong to different partitions. We generate different labels for these statement instances. We observe that the labels of statement instances that depend on each other will always differ by b_i (if all the denominators in $\vec{h}$ and $\vec{\delta}$ are equal). The number of partitions, nop is the least common multiple of b_j $(j = 1, \dots, n)$ and d_k $(k = 1, \dots, r)$. Once the number of partitions nop is known, the goal is to generate code for the partitions. Let $p_{\max} = nop = \text{lcm}(b_1, ..., b_n, d_1, ..., d_r)$. In order to facilitate code generation, we multiply all the elements of $\vec{h}$ and $\vec{\delta}$ by $p_{\max}$. This converts all the values of execution times, $t_k(\vec{\imath})$, $(\forall \vec{\imath} \in \mathcal{I} \wedge \forall k \in [1 \dots r])$ into integers; this is extremely useful in code generation as illustrated next.

4 Code Generation Algorithm

The parallel *doall* code is generated using the schedule derived for each statement instance, $t_k(\vec{\imath})$, $(\forall \vec{\imath} \in \mathcal{I} \wedge \forall k \in [1 \dots r])$. The loop limits and the array indices in the transformed code are derived using the time at which the statement instances are scheduled. Since the loop limits and the array indices should be integers, we derive integer labels for each statement instance:

$$T(\vec{\imath}) = p_{\max} * t_k(\vec{\imath}) \qquad \forall \vec{\imath} \in \mathcal{I} \wedge \forall k \in [1 \dots r]$$

where $p_{\max} = \operatorname{lcm}(b_1, b_2, \ldots, b_n, d_1, d_2, \ldots, b_r)$, and $h_i, (i = 1 \ldots n)$, and $\delta_k, k = (1 \ldots r)$, as defined in Section 3.2. We observe that the statement instances that have the same label and those which have labels such that the difference between them is always an integral multiple of nop belong to the same independent set. This property is used in the code generation algorithm to step through all the statement instances that belong to the same independent set. The code conversion algorithm is given below.

Step 1: The loop limits for the *doall* loop are calculated using the labels derived for each statement instance. We find the minimum and maximum of the labels of all statement instances. Let $MIN = \min(\delta_1, \delta_2, ..., \delta_r)$ and $MAX = \max(\delta_1, \delta_2, ..., \delta_r)$.

$$\begin{aligned} T_{\min} &= p_{\max} * \min_{\forall \vec{\imath} \in \mathcal{I} \wedge \forall k \in [1 \ldots r]} (t_k(\vec{\imath})) \\ &= p_{\max} * \sum_{\forall j \in [1 \ldots n]} (h_{j,+} l_j - h_{j,-} u_j) + MIN \\ T_{\max} &= p_{\max} * \max_{\forall \vec{\imath} \in \mathcal{I} \wedge \forall k \in [1 \ldots r]} (t_k(\vec{\imath})) \\ &= p_{\max} * \sum_{\forall j \in [1 \ldots n]} (h_{j,+} u_j - h_{j,-} l_j) + MAX \end{aligned}$$

The initial points of all the independent sets are those statement instances whose labels are sequentially from $T_{\min}$ to $T_{\min} + nop - 1$. Now the *doall* code will be as shown below:

$$doall\ \ p = T_{\min}, T_{\min} + nop - 1$$

Step 2: Once we have found the initial points of each independent set, the next step is to find all the statement instances that belong to each independent set. As mentioned earlier, the labels of statement instances that belong to the same independent set differ from each other by a value nop, the partitionability. Thus we create the following *do* loop which generates the labels for all statement instances in each independent set:

$$do\ k = p, T_{\max}, nop$$

Step 3: Next we need to find the loop indices in each statement instance from its label. We consider each statement, in the same order as in the original loop. For each statement, we calculate the loop limits of all the indices that generate the same label as shown below: For any arbitrary label k, generated in Step 2, and for any arbitrary statement S_j, we have $h_1 i_1 + \cdots + h_n i_n + \delta_j = \frac{k}{p_{\max}} j \in [1 \ldots r]$ with constraints $l_t \le i_t \le u_t$ $(t = 1, \cdots, n)$. The above equation with the constraints can be easily solved using Banerjee's inequalities [8] as shown below. First calculate the temporary loop limits. This is because the loop limits calculated may not always lie within the original loop limits because of the use of floor and ceiling. For loop index i_1, for statement S_1,

$$i_{1_{\text{temp1}}} = \left\lceil \frac{\frac{k}{p_{\max}} - \left(\sum_{j=1 \wedge j \ne 1}^{n} (h_{j,+} u_j - h_{j,-} l_j)\right) - \delta_1}{h_1} \right\rceil$$

$$i_{1_{\text{temp2}}} = \left\lfloor \frac{\frac{k}{p_{\max}} - \left(\sum_{j=1 \wedge j \ne 1}^{n} (h_{j,+} l_j - h_{j,-} u_j)\right) - \delta_1}{h_1} \right\rfloor$$

The loop limits calculated above are checked to ensure that they lie within the original loop limits; later the actual limits for the loop index under consideration are derived. This procedure is repeated for all the loop indices. The procedure for a loop index is:

if $((i_{1_{\text{temp1}}} \le u_1) \wedge (i_{1_{\text{temp2}}} \ge l_1))$ {
    $i_{1_{\min}} = \max(i_{1_{\text{temp1}}}, l_1)$, $i_{1_{\max}} = \min(i_{1_{\text{temp2}}}, u_1)$, flag = TRUE }
else flag = FALSE
if (flag) {
    do $i = i_{1_{\min}}, i_{1_{\max}}$
        /* Similarly calculate the loop limits for all the loop indices using previous ones; check if they lie within original loop limits */
        $\ldots S_1(i_1, i_2, \ldots, i_n) \ldots$
    enddo }

The statement under consideration is executed only if *flag* is *TRUE* in loop index calculations of *all* the loop indices. This ensures correct ordering of statement instances even though the δ values for statements S_i, $(i = i, \ldots, r)$ are not in increasing order.
Step 4: Repeat Step 3 for all the statements in the original loop in the same order.

The algorithm can be simplified for cases with single statement and also for cases with single level loop. If the value of all $\delta_k, k \in [1 \ldots r]$, are same, then all the statements generate the same labels for a given iteration instance. If one of $h_i, i \in [1 \ldots n]$, is zero, then that particular loop can be parallelized. The *doall* code conversion algorithm for two-level loop nests is shown in Figure 3. This algorithm can easily be generalized to handle loops with nesting level greater than two. The method presented in this paper is illustrated next through several examples. For a detailed discussion, see [5].

5 Examples

5.1 Loop with Multiple Statements

Consider Example 2; the dependence graph for Example 2 with iteration as a basic unit and statement as a basic unit, is shown in Figure 2. By using any of the existing methods [1, 3, 7], we get partition of partitionability of only 1; but by using our method, we get a partitionability of 3. Since we use statement instance as basic schedulable unit, we get higher partitionability over existing methods in all cases of uniform recurrences with multiple statements. The LP problem for this example is:

$$\text{Minimize } \theta \qquad \text{subject to:}$$

$$\begin{gathered} \theta - h_+ + h_- \ge 0 \\ \theta + h_+ - h_- \ge 0 \\ h_+ - h_- - \delta_{1,+} + \delta_{1,-} + \delta_{2,+} - \delta_{2,-} \ge 1 \\ 2h_+ - 2h_- + \delta_{1,+} - \delta_{1,-} - \delta_{2,+} + \delta_{2,-} \ge 1 \end{gathered}$$

The solution to this problem is: $h = \frac{2}{3}$, $\delta_1 = 0$, $\delta_2 = -\frac{1}{3}$. Thus, the number of partitions, $nop = 3$.

5.2 Single Statement in a Multiple Nested Loop

Consider a 3-level nested loop with a single statement (Example 7.1 of [1]) shown below:

Example 3:
do $i_1 = 1, N$
    do $i_2 = 1, N$
        do $i_3 = 1, N$
            $a(i_1, i_2, i_3) = a(i_1, i_2, i_3) + a(i_1 - 1, i_2 + 2, i_3 - 4)$
                $- a(i_1 - 2, i_2 - 4, i_3 + 1) + a(i_1, i_2 - 4, i_3 - 2)$

Since this loop has a single statement, vector $\vec{\delta}$ is not needed. The LP problem for this example is:

$$\text{Minimize } \theta_1 + \theta_2 + \theta_3 \qquad \text{subject to:}$$

$$\begin{gathered} \theta_1 - h_{1,+} + h_{1,-} \ge 0 \\ \theta_1 + h_{1,+} - h_{1,-} \ge 0 \\ \theta_2 - h_{2,+} + h_{2,-} \ge 0 \\ \theta_2 + h_{2,+} - h_{2,-} \ge 0 \\ \theta_3 - h_{3,+} + h_{3,-} \ge 0 \\ \theta_3 + h_{3,+} - h_{3,-} \ge 0 \\ h_{1,+} - h_{1,-} - 2h_{2,+} + 2h_{2,-} + 4h_{3,+} - 4h_{3,-} \ge 1 \\ 2h_{1,+} - 2h_{1,-} + 4h_{2,+} - 4h_{2,-} - h_{3,+} + h_{3,-} \ge 1 \\ 4h_{2,+} - 4h_{2,-} + 2h_{3,+} - 2h_{3,-} \ge 1 \end{gathered}$$

```
/* get the minimum and maximum of all labels T_k(i) */
MIN = min(δ_1, δ_2, ..., δ_r)
MAX = max(δ_1, δ_2, ..., δ_r)
T_min = ((h_{1,+}l_1 − h_{1,−}u_1) + (h_{2,+}l_2 − h_{2,−}u_2) + MIN)p_max
T_max = ((h_{1,+}u_1 − h_{1,−}l_1) + (h_{2,+}u_2 − h_{2,−}l_2) + MAX)p_max
/* parallel code */
doall p = T_min, T_min + nop − 1
    /* get labels for the corresponding partition */
    do k = p, T_max, nop
        do q = 1, r /* for each statement q */
            i_{q_temp1} = ⌈ (k/p_max − (h_{2,+}u_2 − h_{2,−}l_2) − δ_q) / h_1 ⌉
            i_{q_temp2} = ⌊ (k/p_max − (h_{2,+}l_2 − h_{2,−}u_2) − δ_q) / h_1 ⌋
            if ((i_{q_temp1} ≤ u_1) ∧ (i_{q_temp2} ≥ l_1)) {
                i_{q_min} = max(i_{q_temp1}, l_1); i_{q_max} = min(i_{q_temp2}, u_1);
                flag = TRUE }
            else        flag = FALSE
            if (flag) {
                do i = i_{q_min}, i_{q_max}
                    j_q = (k/p_max − h_1 i − δ_q) / h_2
                    if (j_q is an integer) ∧ ((j_q ≥ l_2) ∧ (j_q ≤ u_2)) {
                        ··· S_q(i_q, j_q) ··· }
                enddo }
        enddo
    enddo
enddoall
```

Figure 3: Doall code conversion for 2-nested loops

The solution to this problem is: $h_1 = \frac{18}{52}$, $h_2 = \frac{7}{52}$, and $h_3 = \frac{12}{52}$. Thus, $nop = 52$ since $b_1 = b_2 = b_3 = 52$; this is the same as that obtained by Hollander [1].

5.3 Loop with Linearly Dependent Distance Vectors

Consider the following example where all the distance vectors are not linearly independent.

Example 4:

do $i_1 = 1, N$
  do $i_2 = 1, N$
    $a(i_1, i_2) = a(i_1 - 3, i_2 + 3) + a(i_1, i_2 - 2) + a(i_1, i_2 - 4)$

Since this loop has a single statement, vector $\vec{\delta}$ is not needed. The LP problem for this example is:

Minimize $\theta_1 + \theta_2$ subject to:

$$\begin{aligned}
\theta_1 - h_{1,+} + h_{1,-} &\geq 0 \\
\theta_1 + h_{1,+} - h_{1,-} &\geq 0 \\
\theta_2 - h_{2,+} + h_{2,-} &\geq 0 \\
\theta_2 + h_{2,+} - h_{2,-} &\geq 0 \\
3h_{1,+} - 3h_{1,-} - 3h_{2,+} + 3h_{2,-} &\geq 1 \\
2h_{2,+} - 2h_{2,-} &\geq 1 \\
4h_{2,+} - 4h_{2,-} &\geq 1
\end{aligned}$$

The solution to this problem is: $h_1 = \frac{5}{6}$, $h_2 = \frac{1}{2}$. Thus, $nop = 6$.

5.4 Multi-Statement Nested Loop

Consider Example 1 again. The LP problem for this example is:

Minimize $\theta_1 + \theta_2$ subject to:

$$\begin{aligned}
\theta_1 - h_{1,+} + h_{1,-} &\geq 0 \\
\theta_1 + h_{1,+} - h_{1,-} &\geq 0 \\
\theta_2 - h_{2,+} + h_{2,-} &\geq 0 \\
\theta_2 + h_{2,+} - h_{2,-} &\geq 0 \\
2h_{2,+} - 2h_{2,-} - \delta_{1,+} + \delta_{1,-} + \delta_{3,+} - \delta_{3,-} &\geq 1 \\
h_{1,+} - h_{1,-} - 4h_{2,+} + 4h_{2,-} - \delta_{3,+} + \delta_{3,-} + \delta_{2,+} - \delta_{2,-} &\geq 1 \\
h_{1,+} - h_{1,-} + 3h_{2,+} - 3h_{2,-} - \delta_{2,+} + \delta_{2,-} + \delta_{1,+} - \delta_{1,-} &\geq 1 \\
h_{2,+} - h_{2,-} - \delta_{1,+} + \delta_{1,-} + \delta_{4,+} - \delta_{4,-} &\geq 1 \\
h_{1,+} - h_{1,-} - 3h_{2,+} + 3h_{2,-} + \delta_{1,+} - \delta_{1,-} - \delta_{4,+} + \delta_{4,-} &\geq 1
\end{aligned}$$

The solution to this problem is: $h_1 = \frac{8}{5}$, $h_2 = \frac{-1}{5}$, $\delta_1 = 0$, $\delta_2 = 0$, $\delta_3 = \frac{7}{5}$, and $\delta_4 = \frac{6}{5}$. Thus, $nop = 5$, since $b = d_1 = d_2 = 5$. If the dependence matrix were used, all the existing schemes will generate just one partition here, where as there are five partitions.

6 Summary

This paper has dealt with the important issue of finding independent partitions of *do* loops with uniform dependences. An efficient algorithm was proposed to identify and partition the given uniform recurrence into a maximal independent partition. Since this method uses statement instances as basic schedulable units, it can find partitions with larger cardinality than the existing methods [1, 3, 7] which use iteration instance as a basic block. The method presented in this paper generates maximal independent partitions for an exhaustive set of problems, and can be used to partition multiple nested loops, single nested loop, and multiple statement and single statement recurrences, as illustrated by the examples in Section 5. It can also be used to handle problems with linearly dependent distance vectors without any modification to the algorithm, in contrast to the method in [1], where the linearly dependent distance vectors have to be first transformed into a linearly independent set of vectors. This method uses the schedules derived from the statement level rational affine scheduling algorithm. Since the scheduling algorithm is solved using linear programming—which is polynomial in time [4, 6]—our method is quite fast in practice. For multiple statement recurrences, this method produces the highest partitionability when compared to the existing partitioning algorithms. For problems where, one of $h_i, i \in [1 \ldots n]$, is zero or for cases with some of $\delta_k, k \in [1 \ldots r]$, is zero, statement instances belonging to different independent sets will have the same label. In such cases, our algorithm does not produce the maximal independent partition, but can indicate that such statement instances can be parallelized. Work is in progress in characterizing the set of conditions under which our method generates independent partitions of optimal cardinality.

References

[1] E. D'Hollander, Partitioning and labeling of loops by unimodular transformations, *IEEE Trans. Par. & Dist. Syst.,* 3(4):465-476, Jul. 1992.

[2] L. Lamport, The parallel execution of DO loops, *Comm. ACM,* 17(2):83-93, Feb. 1974.

[3] J. Peir and R. Cytron, Minimum distance: A method for partitioning recurrences, *IEEE Trans. Comp.,* 34(8):1203-1211, Aug. 1989.

[4] J. Ramanujam, Optimal software pipelining of nested loops, in *Proc. 8th Intl. Par. Proc. Symp.,* Apr. 1994, pp. 335-342.

[5] J. Ramanujam and S. Vasanthakumar, Independent partitioning of uniform recurrences, TR-93-12-03, Louisiana State University, Dec. 1993.

[6] A. Schrijver, *Theory of linear and integer programming,* John Wiley & Sons, New York, 1986.

[7] W. Shang and J. Fortes, Independent partitioning of algorithms with uniform dependences, *IEEE Trans. Comp.,* 41(2):190-206, Feb. 1992.

[8] M. Wolfe, *Optimizing supercompilers for supercomputers,* MIT Press, 1989.

[9] H. Zima and B. Chapman, *Supercompilers for parallel and vector supercomputers,* ACM Press, 1990.