Computing Systems in Engineering,vol. 6, nos 4/5, Oct. 1995, pp. 415-422

Memory Optimization for Parallel Functional Programs

Balaram Sinharoy
Large Scale Computing Division
IBM Corporation, P.O. Box 950
Poughkeepsie, NY 12602, USA

Boleslaw Szymanski
Department of Computer Science
Rensselaer Polytechnic Institute
Troy, NY 12180-3590, USA

Abstract

Parallel functional languages use single valued variables to avoid semantically irrelevant data dependence constraints. Programs containing iterations that redefine variables in a procedural language have the corresponding variables declared with additional dimensions in a single assignment language. This extra temporal dimension, unless optimized, requires an exorbitant amount of memory and in parallel programs imposes a large delay between the data producer and consumers. For certain loop arrangements, a window containing a few elements of the dimension can be created. Usually, there are many ways for defining a loop arrangement in an implementation of a functional program and a trade-off between the memory saving and the needed level of parallelism has to be taken into account when selecting the implementation.

In this paper we prove that the problem of determining the best loop arrangement by partitioning the dependence graph is NP-hard. In addition, we describe a heuristic for solving this problem. Finally, we present examples of parallel functional programs in which the memory optimization results in reducing the local and shared memory requirements and communication delays.

1 Introduction

Parallel functional languages use single valued variables to avoid semantically irrelevant data dependence constraints [1]. Programs containing iterations that redefine variables in a procedural language have the corresponding variables declared with additional dimensions in a single assignment language. Often a *window* containing a few elements of the dimension can be created. Only window elements need to be kept in the local memory. For a window to be feasible, all references to a variable must be enclosed in a loop that accesses a limited number of elements of the reduced dimension at each step.

In this paper we consider functional programs written as a set of statements defining array variables over their index domains (cf. EPL or Crystal functional languages [1]). Therefore each statement can be thought of as nested in the loops iterating over these index domains. Loops over the same statement can be nested in a different order. Loops with the same index domain iterating over different statements often can be merged together. Usually, there are many ways for defining a loop arrangement in an implementation of a functional program, as shown in Example 1.

Example 1 A recurrence defined as

$$A[i_1, i_2, i_3] = f(A[i_1 - 3, i_2, i_3 - 1], A[i_1 - 2, i_2, i_3 - 2])$$

can be implemented with various loop arrangements listed below with the corresponding windows (first subscript refers to the outermost loop):

1. For loop arrangement (i_3, i_2, i_1): $[3, --, --]$
2. For loop arrangement (i_2, i_3, i_1): $[1, 3, --]$
3. For loop arrangement (i_1, i_2, i_3): $[4, --, --]$

The algorithm presented in Subsection 2.2 will select loop arrangement (i_2, i_3, i_1) as requiring the minimum memory. The windows in case 2 and 3 are shown in Figure 1. □

Example 1 shows three choices of loop arrangements with different dimensions windowed in each case. Our experience indicates that the compiler analyzing global data dependencies in a functional program can make

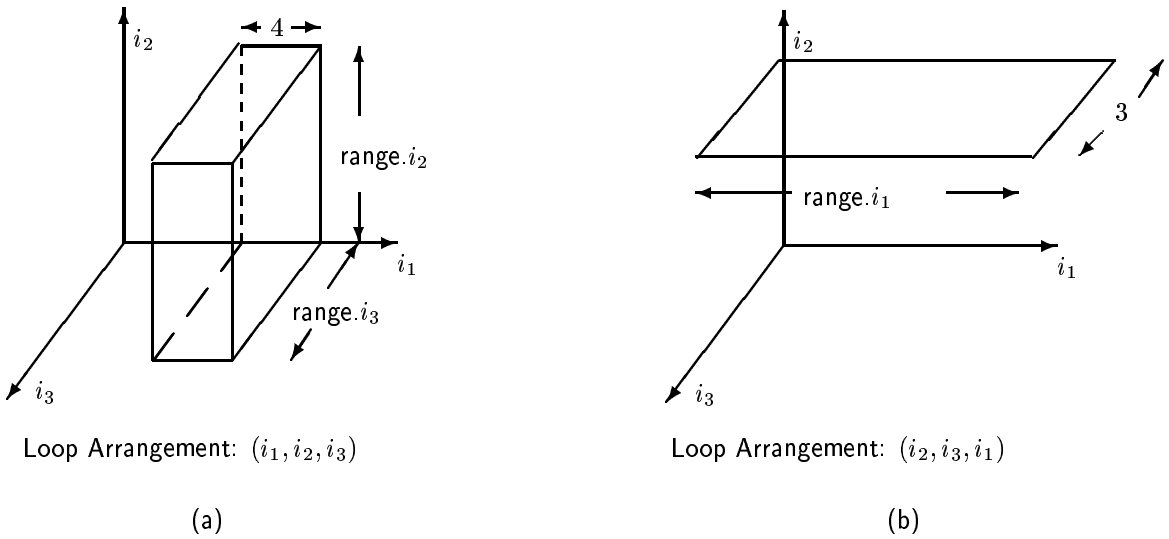

Figure 1: Some of the windows that can be formed for example 1.

equal or better choice of the loop arrangement than the programmer using a traditional language and relying more on intuition and the meaning of the variable than on the implementation efficiency.

The data dependencies in a functional program are represented by its *dependence graph*, which associates each dependence with its attributes such as the distance between dependent elements, conditions under which dependence holds, subscripts assigned to it, etc. All data dependencies can be represented as various kinds of edges in the dependence graph [2]. Each node in the graph contains information about the represented variable such as the number and ranges of its dimensions, its type and class, conditions guarding its definitions, etc.

A program represented by a dependence graph can be partitioned into many cooperating tasks, each of which can be implemented using many different semantically equivalent loop arrangements. Determination of suitable partitions of a program and mapping of the partitions onto the processors has been studied extensively in the literature (for example, [2, 3, 4, 5, 6, 7]). The communication cost in a distributed memory system (which is critical in determining the total execution cost) depends heavily on the partitioning of the dependence graph (*tasks*), the mapping of the tasks and data onto the processors as well as the hardware characteristics of the machine. It also depends on the way communication is interwoven with computation. For example, sending a window in each loop execution is preferable to sending the entire dimension at once, because the latter solution imposes a large synchronization delay, even though the total volume of data communicated may be lower.

To have window on a dimension of a variable, the statements in which the variable appears must all be nested in the loop that corresponds to that dimension. A dimension is not be windowable, if data dependence prohibits such nesting of the loops. Evidently, the dimension(s) on which window(s) can be formed depends on the loop arrangements in the implementation of the functional program. A particular loop arrangement may allow window(s) on some variables while prohibiting them on others. In Subsection 2.2 we prove that the problem of determining the loop arrangement of a functional program that results in the minimum memory requirements is NP-hard. Loop merging often increases the opportunity of forming windows on variables, but it may also limit extraction of parallelism. Thus, a trade-off between memory saving and the number of independent tasks must be considered when selecting the loop arrangement.

The following heuristic was proposed in [2, 8] to minimize memory for Model programs. The data dependence graph is decomposed into *strongly connected components* (abbreviated as SCC)[1]. The quotient graph, Q, of the array graph with respect to the SCCs, is formed. Each vertex in Q represents an SCC and limits the feasible order of loop nesting of the object program. In quotient graph an SCC is selected at random as a seed component

[1]The vertex set V_D of a digraph D can be partitioned into equivalence classes $V_i, 1 \leq i \leq r$, such that vertices v and w are equivalent if and only if there is a directed path from v to w and a path from w to v. Let $E_i, 1 \leq i \leq r$ be the set of edges connecting the pairs of vertices in V_i. The digraphs $D_i = (V_i, E_i)$ are called the SCC's D.

and loop merging starts with components connected to the seed by edges. Because the SCCs typically have several dimensions, there may be many possible bases for mergers and loop merging is performed by progressively finding the lowest memory cost of mergers. The heuristic merges the outer loops, without analyzing the influence of the inner loops on the memory cost. It also considers the memory savings of the single merger disregarding possible aggregates of mergers. The heuristic described in this paper selects the aggregate of mergers with the largest memory savings.

In Section 2, the problem of finding the loop arrangement that results in the highest memory saving is discussed. Subsection 2.1 discusses the complexity of finding the best loop arrangement, and in Subsection 2.2 and 2.3 exact and heuristic algorithms for doing so are provided.

2 Determination of Loop Arrangement

Consider the program decomposition defined by partitioning of its data dependence graph into SCCs. Methods for determining the loop arrangement for each of the SCCs and the ordering of the SCCs that minimize the memory requirement are investigated in this section.

Definition 1 The *loop arrangement* in a functional program is the nesting and scopes of loops in its implementation in a traditional language program. A *loop nest* in a loop arrangement is the ordered list of loops[2] $(I_1 > I_2 > \ldots > I_k)$ such that the loop on I_l is nested within the loop on I_{l-1} for $2 \leq l \leq k$. The *size* of the loop nest is k. A loop nest is *maximal* if it is not contained in any other loop nest. □

The execution ordering of the statements in a functional program is determined by a recursive procedure called the *scheduler* [2]. At each level of recursion, the scheduler linearly orders all SCCs in the dependence graph of the program to satisfy the existing data dependencies. It then attempts to schedule each SCC by enclosing it in such loops that dependencies represented by some of the edges inside the SCC are enforced by the loop execution. If successful, the scheduler removes the loop-enforced edges from the SCC thereby creating a cascade of new SCCs (components of the parent SCC) that can be ordered linearly. Removal of the redundant edges at each level defines the loops that should surround all statements represented by nodes within the parent SCC.

Each level of recursion produces one or more of such loops. Loops obtained for an SCC M are always nested within those obtained in a previous recursion for the SCC that contains M. By analyzing all SCCs at each level of recursion, it is possible to obtain the loops that should surround each statement and the partial order of the loop nests. In the general case, the loops that surround a statement α_1 and all their possible nestings can be represented by

$$\{(I_{11}, \ldots, I_{1n_1}),\ (I_{21}, \ldots, I_{2n_2}),\ \ldots,\ (I_{k1}, \ldots, I_{kn_k})\} \tag{1}$$

Loop ordering (1), indicates that k levels of recursion were needed to obtain an SCC that contains only statement α_1 and the loops obtained at recursion level i $(1 \leq i \leq k)$ are $I_{i1}, \ldots, I_{in_i}$. Loops obtained in the same level of recursion can be permuted in an arbitrary fashion; however, loops obtained at an earlier recursion always precede those obtained at a later recursion. In (1), $(I_{i1}, \ldots, I_{in_i})$ indicates any permutation of $I_{i1}, \ldots, I_{in_i}$. Expression (1) defines the set of all possible valid loop nest arrangements for a statement and constitutes the statement's set of permissible loop arrangements.

Let us assume that all the references to array A are nested within the set of permissible loop arrangements described by (1) and the array references to the corresponding n dimensions of the array are[3]:

$$A[i_1 - c_{11} \ldots, i_n - c_{1n} \ldots], \ldots, A[i_1 - c_{m1} \ldots, i_n - c_{mn} \ldots]$$

Then the following algorithm determines the window sizes for these n dimensions of the array A.

[2] The loop nest is defined by the loop control variables $I_1, \ldots, I_k$.

[3] Without loss of generality, it is assumed that $i_1, i_2, \ldots, i_n$ are the subscripts used to iterate over the corresponding dimensions of A, where $i_1, i_2, \ldots, i_n$ is a permutation of $I_{11}, \ldots, I_{1n_1}, I_{21}, \ldots, I_{2n_2}, \ldots, I_{k1}, \ldots, I_{kn_k}$.

Algorithm to determine the window size of a variable

Step 1: Compute

$$w_l = \begin{cases} \max_{1 \le k \le m} |c_{kl}| & if\ c_{kl}\ are\ of\ the\ same\ sign \\ range.i_l & otherwise. \end{cases}$$

Set $t = 1$.
Set $I = \{I_{t1}, \ldots, I_{tn_t}\}$

Step 2: Select $j \in I$ such that $w(j)$ is minimum for all $j \in I$.
Set $S(j) = 1 + w(j)$ and remove j from I.
If $S(j)$ is 1 and I is not empty then go to step 2.

Step 3: If $S(j) = 1$ then set $t = t + 1$ and go to step 2.

Step 4: For all j for which $S(j)$ is not set, do $S(j) = range.j$.

Definition 2 The *set of permissible loop arrangements* of a statement is the set of all possible valid loop nest arrangements that can be formed around the statement and is represented by a list

$$\{(I_{11}, \ldots, I_{1n_1}),\ (I_{21}, \ldots, I_{2n_2}),\ \ldots,\ (I_{k1}, \ldots, I_{kn_k})\}$$

Each I_k in the above expression represents a distinct loop, with I_k as the loop index. Loops appearing in the same parenthesis can be permuted. If loop I_k appears in a parenthesis preceding that which contains loop I_l, then I_l is always nested within the loop of I_k. The *set of permissible loop arrangements of a variable* is the intersection of the sets of permissible loop arrangements of all statements in which the variable appears. □

Only loop arrangements in the set of permissible loop arrangements of a variable need to be considered when determining the loop arrangement that provides the largest memory saving.

Definition 3 If the set of permissible loop arrangements of a variable is null (that is, there is no loop arrangement common to all statements in which the variable appears), then no dimension of the variable is windowable and the variable is termed *physical*. Also, if the array reference is not regular[4] on dimension k, then that dimension is not windowable. □

Definition 4 Two or more sets of loop nest arrangements are *strongly compatible* if there is at least one complete loop nest arrangement common to all. They are *compatible* if there is at least one outermost loop common to all of them, otherwise they are incompatible (e.g., $\{(i,j),(k)\}$ is strongly compatible with $\{(i),(j,k)\}$ and compatible with $\{(j,k),(i)\}$ but it is incompatible with $\{(k),(i,j)\}$). □

In practice, only a few such loop arrangements for a variable are possible. The problem now is to select the loop arrangements for each variable so that the total memory requirement of a program is minimized. To accomplish this, a directed graph, termed a data graph, is constructed. A variable may be represented by more than one node in the data graph and all such nodes will have distinct but permissible loop arrangements of that variable.

Definition 5 A *data graph* is a directed graph where the edges are colored with blue, red or green and each vertex is labeled with a variable along with one of its permissible loop arrangements (so each variable contributes as many vertices to this graph as there are permissible loop arrangements for it). There is an edge (of arbitrary orientation) between two vertices representing the same variable. If variable A is defined in terms of variable B, then there are edges from all vertices representing variable B to all those representing variable A. If a particular element of array B is referred, the edge is colored red. Edges connecting two vertices representing the same variable but with incompatible loop arrangements are colored blue. Edges connecting two vertices representing different variables with incompatible loop arrangements are colored red. All other edges are colored green. □

[4] An array reference is considered *regular* on dimension k, if and only if the k-th dimension of array A is always referred as $i_k - c$, where i_k is a fixed subscript and c is any integer.

It immediately follows from the above definition that if a node is selected (that is, the permissible loop arrangement that labels the node is chosen for the variable represented by that node), then any node that is reachable by a semipath consisting of only blue edges cannot be selected. Two nodes connected by a green edge have compatible loop arrangements and can be scheduled within the same nest of loops. Blue, green and red edges are represented by dotted, solid and dashed lines in this paper.

A computation that frequently occurs in scientific applications is to apply a binary operation over an entire vector and store the result in the last element of the vector[5]. Such operations result in references of the form $A[i, range.j, k]$, where $range.j$ indicates the range of the second dimension of array A. If a particular element (see example 2) of an array appears in the right side of a statement, then the edges from vertices representing this variable to those representing the variable on the left side of the statement are colored red (i.e., to indicate that the variables represented by the two nodes should not be scheduled within the same loop).

To ensure that single variables are being represented by a small number of nodes in the data graph, a single node is allowed to represent more than one loop arrangement of the variable. Suppose three five-dimensional variables A, B and C allow loops to be opened as described by

$$\{(i,\ j,\ k),\ (l,\ m)\},\ \{(i,\ j),\ (k,\ l,\ m)\},\ \{i,\ (j,\ k,\ l),\ m\},$$

respectively. In this case, the use of separate nodes for each loop arrangement of a variable will require 12 nodes to represent variable A, 12 to represent B and 6 to represent C. However, there are 4 loop arrangements common to A and B, 2 to each of B and C, and A and C and 1 to all three variables. We can reduce the size of the data graph, if each node for a variable is made to represent a group of loop arrangements which consists of loop arrangements that the variable has in common with none, one, two, ... of the variables with which it appears in a statement. In this case, the above example can have at most 12 nodes instead of 30. In general, with N variables the number of nodes in this compact data graph is never more than

$$N * \left[\binom{N-1}{0} + \binom{N-1}{1} + \binom{N-1}{N-1} \right] = N * 2^{N-1}$$

which is still large. To keep the number of nodes polynomially bounded, we use the reduced data graph, defined below.

Definition 6 In a *reduced data graph*, $d_A + 1$ of vertices represent a variable A, where d_A is the number of distinct variables with which A appears in the same statement. One of the vertices representing variable A is labeled with the set of permissible loop arrangements of A and the rest are labeled with the intersection of the set of permissible loop arrangements of A and that of another distinct variable with which A appears in the same statement. Edges are formed and colored in the same way as in the data graph. □

There are at most $N + 2E$ nodes in the reduced data graph, where E is the total number of edges in the dependence graph and N is the number of variables. Since there are relatively few nodes in the reduced data graph, it can be analyzed efficiently, to produce a good loop arrangement.

2.1 Complexity of the Loop Arrangement Problem

Theorem 1 *Loop arrangement problem is NP-hard.*

Proof Suppose that we are to determine the maximum weight independent set[6] of a graph $G = (V, E)$ with vertices $V = \{v_1, v_2, \ldots, v_n\}$ and edges $E = \{e_1, e_2, \ldots, e_m\}$. Let vertex v_i has weight w_i for $i = 1, \ldots, n$. To polynomially transform this problem to that of finding the best loop arrangement, we construct the following program. For each vertex v_i, there are arrays A_i's of one dimension with ranges $w_i + 1$'s, and statements α_i's that define arrays A_i's as

$$\alpha_i :\ A_i[I_i]\ =\ \text{if } (I_i == 1) \text{ then } c_i \text{ else } f_i(A_i[I_i - 1])$$

[5]For example, in scientific computation there is often a need to apply an associative operator (such as, $+, *, -, max, min$, etc.) selectively on the elements of an array. *Scan* and *reduce* are language constructs in some parallel languages that allow such operations. Reduce applied to a vector of values produces a scalar result, whereas scan results in a vector of partial results [1]. For example, $Y = \sum A[I]$ can be written as $X[I] = X[I-1] + A[I]; Y = X[range.I]$, which is equivalent to $Y = reduce(+, A[I], I)$.

[6]The maximum weight independent set problem is to select an independent set $V_s \subseteq V$ (that is, no two vertices in V_s are adjacent) such that $\sum_{i:v_i \in V_s} w_i$ is maximum. It is a well known NP-complete problem [9].

where I_i's are subscripts, c_i's are given constants and f_i's are given functions.

For each edge $e_k \in E$ incident on vertices v_i and v_j, there is a two dimensional array B_{ij}, and a statement β_{ij} that defines B_{ij} as

$$\beta_{ij}: \quad B_{ij}[I_i, I_j] \quad = \quad g_{ij}(A_i[I_i], A_j[I_j]);$$

where I_i's and I_j's are subscripts (defined for ranges of arrays A_i and A_j, respectively), and g_{ij}'s are given functions. Array B_{ij} is of size $(w_i + 1) \times (w_j + 1)$. From the construction of the program it is evident that

1. Window can be formed on each dimension of any array B_{ij}.
2. If window is formed on array A_i, then the loop representing its dimension, I_i, should be the outermost loop enclosing the statements β_{ij} and β_{ji}, created for edges adjacent to node V_i. Moreover, the memory saving from forming a window on A_i is w_i.
3. Since any loop arrangement allows optimum window (of size 1) to be formed on B arrays, we need to determine the loop arrangement that results in the largest total memory saving for the A arrays.

If v_i and v_j are adjacent in graph G, a window can be formed on array A_i or A_j but not both because the different outermost loop has to enclose statement β_{ij} for each window. Therefore, any set of arrays A_i's that are windowed, corresponds to an independent set of vertices v_i's that represent these arrays. Conversely, if set S of vertices is an independent set, then windows can be formed on all the arrays that correspond to nodes in the set S. Thus the loop arrangement problem reduces to the maximum weight independent set problem. □

The following definitions are used in describing exact and heuristic algorithms to find the loop arrangement in a program that provides the largest memory saving.

Definition 7 The *G-connected components* of a reduced data graph are the connected components in the underlying graph of a directed graph obtained by deleting all non-green edges in the reduced data graph. □

Definition 8 A *loop cluster* of a reduced data graph is a subgraph, L, of a G-connected component of the reduced data graph which satisfies the following conditions:

1. the subgraph of the reduced data graph induced by the vertices of L has no blue or red edges,
2. there is no directed path in the reduced data graph between two vertices belonging to L which include edge(s) not in the subgraph, and
3. if an external vertex v has an incoming edge from a vertex in $V(L)$, then all incoming edges to v are also from vertices in $V(L)$.

A loop cluster is *maximal* if it is not properly contained in any other loop cluster. The *variables of a loop cluster* are the variables used to label its vertices. The *statements of a loop cluster* are those in which any variable of a loop cluster appears. □

Definition 9 The *set of permissible loop arrangements of a loop cluster* is the intersection of all permissible loop arrangements of all vertices in the loop cluster. □

Definition 10 The *weight* of a loop cluster is the largest collective memory saving on all variables represented by the vertices in L, under the set of permissible loop arrangements of the loop cluster. □

The weight of a loop cluster can be determined by adding the memory saving at each variable in the loop cluster when the best windows are formed, under the set of permissible loop arrangements of the loop cluster. Such windows can be obtained by using the algorithm described earlier in this section. Figure 2 shows the loop clusters for two different data graphs, where it is assumed that there is only one representative vertex for each data node. It also shows that the loop clusters can be broken into smaller sizes by introducing additional data dependence.

Lemma 1 *It is possible to form a loop nest representing a loop arrangement common to all vertices of a loop cluster such that all statements of the loop cluster and only those statements can be scheduled within the loop cluster.*

Proof Since no two vertices of a loop cluster are connected by a red edge, at least one loop arrangement is common to all. The absence of a blue edge between any two nodes indicate that no instances of incompatible loop arrangements of the same variable appear in a loop cluster. So there is a loop arrangement that is common to all the vertices.

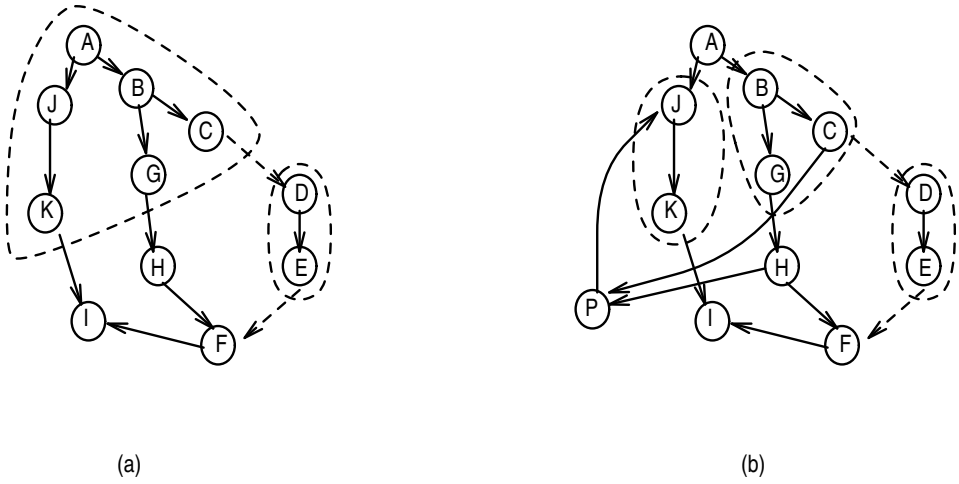

Figure 2: Maximal loop clusters for different data graphs. Figure (b) shows that additional data dependence can reduce the size of a maximal loop cluster.

A variable A, represented by node N_A in the data graph is defined in terms of the variables represented by nodes $N_{A_1}, \ldots, N_{A_k}$, if there is an incoming edge to N_A from nodes $N_{A_1}, \ldots, N_{A_k}$. Since all or none of the incoming edges to an external node are from the internal nodes of a loop cluster (condition 3), all statements of a loop cluster can appear within the loop nest.

Condition 2 in the definition of a loop cluster guarantees that no other statement needs to be scheduled within the loop nest. Since there is no directed path in the dependence graph between two nodes belonging to the loop cluster which include edge(s) not belonging to it, there is no need to schedule a statement (requiring a different loop arrangement) between those of the loop cluster. □

Lemma 2 *If all vertices of a loop cluster represent a single permissible loop arrangement, the memory saving obtained by scheduling its component loop clusters can be achieved by scheduling maximal loop clusters only. This is not necessarily true if the vertices represent a collection of permissible loop arrangements.*

Proof Let $l_i \ \ (1 \leq i \leq p)$ be p node-disjoint loop clusters, each saving storage of amount s_i (under the best loop arrangement for l_i). Assume that these loop clusters are subgraphs of the maximal loop cluster L which saves S amount of storage (under best loop arrangement for L). If the vertices of a loop cluster represent a single permissible loop arrangement, then there is only one way to form the best window on a variable represented by a vertex in the data graph (since all nodes in $l_i \ \ (1 \leq i \leq p)$ are also in L, $S \geq \sum_{i=1}^{p} s_i$).

However, if there is more than one permissible loop arrangement for any vertex in loop cluster L, the best loop arrangement for l_i can be different from that for L. Thus the formation of a window on a variable under loop cluster l_i to which this variable belongs can be different from the window for this variable. Hence different storage saving can be obtained under different loop clusters. Thus we cannot guarantee that by selecting only the maximal loop clusters the largest memory saving will be achieved. □

Example 2 Suppose we are to determine windows on the variables in the following two programs, which differ in the reference to X in the last statement.

```
B[i] = f(A[i])                      B[i] = f(A[i])
X[i] = g(X[i-1], A[i])              X[i] = g(X[i-1], A[i])
C[i] = h(A[i], B[i], X[range.i])    C[i] = h(A[i], B[i], X[i])
        (a)                                 (b)
```

Figure 3 shows the data graph obtained in each of the two cases. In Figure 3(a) there are two loop clusters $\{X\}$ and $\{B, C\}$. No loop cluster consists of A only as it would violate condition 3. In Figure 3(b) the entire graph forms a loop cluster. Thus, a window (of size 1) can be formed only on X, B and C in the first case and on all arrays in the second case and the codes generated for the two programs are shown in Example 3. □

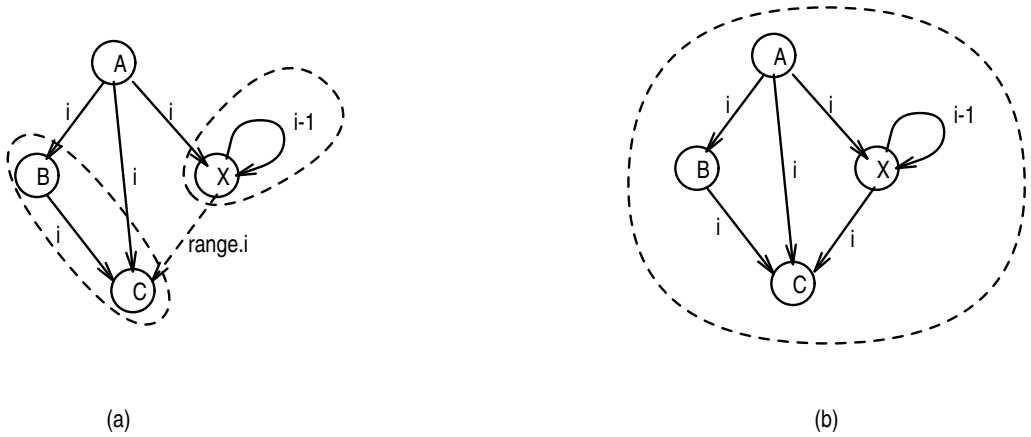

Figure 3: Data graphs when a particular element of an array, X[range.i], is (a) and is not (b) referred to.

Two algorithms for determining the optimal or suboptimal (respectively) selection of loop clusters are described in the remainder of this section. Once a loop cluster is selected, the algorithm presented earlier in this section can be used to find the best loop arrangements for each loop cluster and the best window that can be formed on all variables represented by nodes belonging to the chosen loop cluster.

If loop cluster L is chosen, array A represented by vertex v adjacent to a vertex in $V(L)$ cannot be windowed (often because at least one of the statements in which A appears is scheduled in a loop arrangement that is different than the loop arrangement that labels v). Before the next loop cluster can be chosen, all such vertices must be deleted from the data graph. The set of all vertices that should be deleted once a loop cluster L is chosen constitutes the set of removable vertices of loop cluster L as defined below.

Example 3 Programs generated for the two specifications in example 2. Figure 3 shows the respective data graphs and the loop clusters selected. Lower case letters represent the windowed variables corresponding to the original variables in upper case.

(a)

```
Read array A
Loop on i
    x = g(x, A[i])
Loop on i
    b = f(A[i])
    c = h(a[i], b, x)
```

(b)

```
Loop on i
    a = Read next element of array A
    b = f(a)
    x = g(x, a)
    c = h(a, b, x)
```

Definition 11 The *set of removable vertices of loop cluster* L, denoted by D_L, consists of :

1. the vertices in L
2. the vertices at a distance of 1 (through red or green edges) from a vertex in L
3. any external vertex that can be reached by a semipath consisting of blue edges and at most one green edge from a vertex in L.

□

Although the problem of determining the optimal loop arrangement is NP-hard, in certain applications the problem size is small enough to warrant investigating methods for obtaining an optimal solution.

2.2 Exact Algorithm

If the number of permissible loop arrangements for each array is small, each vertex in the data graph can be made to represent a variable for a particular permissible loop arrangement. For the exact algorithm, the color

of the edges connecting two vertices labeled with compatible, but not strongly compatible, loop arrangements is changed from green to red. According to lemma 2, only the maximal loop clusters of the data graph need to be considered in this case.

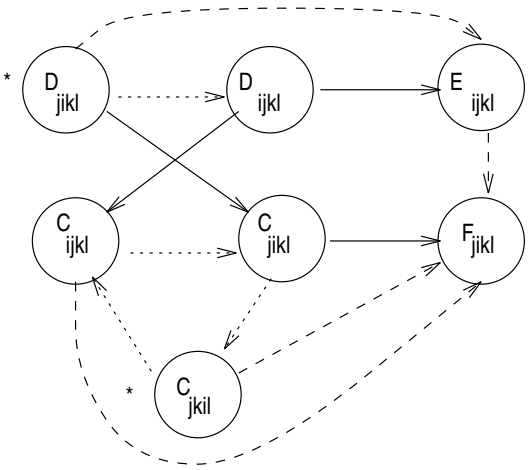

Figure 4: Example of a complete data graph.

Example 4 A complete data graph where permissible loop arrangements for arrays D, C, E and F are $\{(i,j),k,l\}$, $\{(i,j,k),l\}$, $\{i,j,k,l\}$ and $\{j,i,k,l\}$, respectively ("*" near a vertex indicates that the loop arrangement represented by the vertex produces the largest memory saving for the corresponding array), is shown in Figure 4. By an exhaustive search, one of the following five loop arrangements can be determined to result in the largest memory saving: $\{D_{jikl}, F_{jikl}\}$, $\{C_{jkil}, E_{ijkl}\}$, $\{D_{ijkl}, C_{ijkl}, E_{ijkl}\}$, $\{C_{jikl}, F_{jikl}\}$, and $\{D_{jikl}, C_{jikl}\}$ (here, D_{ijkl}, etc., indicates that all statements in which array D appears should be within the loop arrangement $\{i,j,k,l\}$). □

A backtracking algorithm can be designed to search all possible sets of loop clusters for the best set. Different pruning strategies based on the current knowledge of memory saving can be employed at different stages of the backtracking algorithm to reduce the search time.

2.3 Heuristic Algorithms

Heuristic algorithms can be utilized to provide a good solution to this problem when either the number of permissible loop arrangements for the arrays is large or the number of arrays to be optimized is considerable. One such algorithm is described below. It starts with a reduced data graph, as defined in Section 2, and forms a loop cluster dependence graph (defined later) from which object code can easily be generated.

Definition 12 The *loop cluster dependence graph* of a reduced data graph is a digraph $D = (V_D, E_D)$ where $V_D = \{L_1, L_2, \ldots, L_k\}$ is the set of loop clusters found in the reduced data graph and $(L_i, L_j) \in E_D$ for $L_i, L_j \in V_D$, if there is a path in the reduced data graph from a vertex in L_i to a vertex in L_j. □

Heuristic A:

1. Select an unscheduled G-connected component, C, of the data graph. If none is found, Go to step 5.
2. Select a vertex V in C and set L ={V} (for example, $L = \{A\}$, in Figure 2(a)).
3. Add an unselected vertex in C to L, such that L remains a loop cluster and its weight increases. Repeat this step until no more vertices can be added (using Figure 2(a) as an example, continue adding vertices $\{B, J, C, G, K\}$ to L, in this sequence, as long as the weight of L increases). L is a maximal loop cluster.
4. Select loop cluster L and delete its removable vertices from the data graph. If no more vertices remain in C, go to step 1, else go to step 4.

5. Create the loop cluster dependence graph where vertices represent the loop clusters selected at step 4 and edges represent the execution dependence (which can be obtained from the dependence graph or the data graph) among the loop clusters.

If the array sizes vary significantly in a program, better performance can be expected by selecting the maximal loop cluster at each iteration across all the G-connected components with the following criterion. The selected loop cluster should have the highest ratio of the sum of weights on each node in the loop cluster to that of the nodes that will be deleted if the loop cluster is chosen. The loop cluster dependence graph contains all the information needed to form the loop arrangement for the analyzed program.

3 Summary

This paper describes a program transformation method for optimizing the execution-time memory requirements for local and shared variables as well as the communication delays incurred by accessing distributed data structures. The method is based on reducing dimensions to windows of arrays in a given functional program. It is also suitable for programs written in languages observing single assignment rule (i.e., dataflow). Exact and heuristic algorithms are provided for analyzing the partitions of a dependence graph (which represents a task mapped onto a processor) to determine a loop arrangement that results in the minimum memory requirement.

The heuristic algorithms have been implemented on Sequent Balance Symmetry 81000, a shared memory multiprocessor system, and incorporated with the EPL compiler [1]. The results obtained when heuristics were applied to EPL programs for LU decomposition and for solving a system of linear statements by Jacobi iterations showed that automatic memory optimization matched the efficiency of handwritten codes.

Computer arithmetic is not associative and caution must be exercised when reordering loop arrangements for memory optimization. In the programming paradigm considered in this paper, computation is specified without explicit loop arrangements and the compiler is allowed to select any loop ordering that satisfies the optimization criteria. Some numerical computations are highly sensitive to the order of evaluation and for such computations the user is allowed to provide annotations that prohibit some error-prone loop orderings from being considered by the compiler.

Acknowledgment

This work was supported in part by ONR Grant N00014-93-1-0076 and by the IBM Development Award. The content of this entry does not necessarily reflect the position or policy of the US Government – no official endorsement should be inferred or implied.

References

[1] B. K. Szymanski (ed.), *Parallel Functional Programming Languages and Compilers,* ACM Press, New York, 1991.

[2] B. K. Szymanski and N. Prywes, "Efficient Handling of Data Structures in Definitional Languages," *Science of Computer Programming* **10** 221-45 (1988).

[3] J. Baxter and J. H. Patel, "The LAST Algorithm: A Heuristic-Based Static Task Allocation Algorithm," Proceedings of the 1989 International Conference on Parallel Processing, **II**, pp. 217 - 222, 1989.

[4] L. Bic, M. D. Nagel and J. M. A. Roy, "Automatic Data/Program Partitioning Using the Single Assignment Principle," Proceedings of Supercomputing 1989, Reno, Nevada, pp. 551 - 556, 1989.

[5] P.-Z. Lee and Z. M. Kedem, "Mapping Nested Loop Algorithms into Multidimensional Systolic Arrays," *IEEE Transactions on Parallel and Distributed Processing* **1** (1990).

[6] J.-K. Peir and R. Cytron, "Minimum Distance: A Method for Partitioning Recurrences for Multiprocessors," Proceedings of the 1987 International Conference on Parallel Processing, pp. 217-225, 1987.

[7] B. Sinharoy and B. K. Szymanski, "Finding Optimum Wavefront of Parallel Computation," *Journal of Parallel Algorithms and Applications,* **1** (1993).

[8] K. S. Lu, *Program Optimization based on a Non-Procedural Specification,* Ph. D. dissertation, University of Pennsylvania, Philadelphia, PA, 1981.

[9] M. R. Garey and D. S. Johnson, *Computers and Intractability A Guide to the Theory of NP-Completeness*, W. H. Freeman and Company, New York, 1979.