# Data Flow Analysis Driven Dynamic Data Partitioning

Jodi Tims Rajiv Gupta Mary Lou Soffa
Department of Computer Science

University of Pittsburgh Pittsburgh, PA 15260
{jlt, gupta, soffa@cs.pitt.edu}

**Abstract.** The use of distributed memory architectures as an effective approach to parallel computing brings with it a more complex program development process. Finding a partitioning of program code and data that supports sufficient parallelism without incurring prohibitive communication costs is a challenging and critical step in the development of programs for distributed memory systems. Automatic data distribution techniques have the goal of placing the responsibility of determining a suitable data partitioning into the domain of the compiler. Static program analysis techniques that expose data interrelationships and derive performance estimates are central to the development of automatic data distribution heuristics. In this paper we present a data partitioning heuristic that makes use of array data flow analysis information in the modeling of data interrelationships and the estimation of costs associated with resolving interrelationships via communication. The global view provided by data flow analysis permits consideration of potential communication optimizations before data partitioning decisions are made. Our heuristic uses tiling techniques to determine data partitionings. The resulting data distributions, while still regular, are not limited to the standard BLOCK, CYCLIC and BLOCK-CYCLIC varieties. Preliminary results indicate an overall reduction in communication cost with our technique.

## 1 Introduction

The distributed memory approach to parallel computing offers benefits of scalability and cost effectiveness. As in shared memory systems, program development for distributed memory architectures requires the parallelization of program code in order to exploit the power of multiple processors. Distributed memory systems encounter additional complexity, however, in the need to partition data among the independent memory units of each processor. If data is distributed to the local memories in a haphazard fashion, any gains due to parallel execution of the code may well be equaled or exceeded by communication costs.

The problems of process decomposition and data distribution share a high degree of coupling. The Single-Program—Multiple-Datastream (SPMD) model of execution [5] assumed in this work, addresses this interaction by requiring that computations occur on the processor where the result operand has been mapped. This "owner-computes" rule simplifies program development for distributed memory architectures somewhat, in that process decomposition is implicit in the determination of a data decomposition. Even assuming the SPMD execution model, however, data distribution is a complex task. Automatic data distribution algorithms place the burden of determining a data partitioning into the domain of the compiler. This approach not only simplifies program development for distributed memory systems, but also encourages the use of nonstandard distribution patterns.

Automatic data distribution techniques fall into two primary categories. *Static data distribution* techniques [10, 12] determine a single distribution for each array variable for the duration of the program's execution. The limitation of this approach is that if data interrelationship patterns vary significantly during the course of a program, some phases of the execution will incur significant communication costs. *Dynamic data distribution* techniques [2, 3, 4, 6, 7, 13, 15, 17] address this limitation by allowing arrays to be realigned and redistributed during execution so that more of the program's interrelationships may be resolved by the data distribution. Care must be taken, however, that the communication saved by changing the data distribution is not offset or exceeded by the inherent cost of relocating data values during realignment and redistribution operations. Finding the optimal dynamic data distribution for a given program has been determined to be an NP-complete problem [14].

Existing dynamic distribution techniques share a common view that a single distribution encompasses the partitioning of multiple array variables that are used within a particular code segment, most commonly a loop nest. Using an estimate of communication cost as a basis, all interrelationships within the code segment are analyzed to determine the "best" distribution for the segment. Distribution conflicts between adjacent code segments are resolved either by sacrificing parallelism, merging distributions and inserting communication into one of the nests, or redistributing all arrays involved in both segments. After determining the data distribution, a communication optimization phase tries to reduce communication costs using techniques such as message aggregation, overlapping communication and computation, and elimination of redundant messages.

In this paper we present an automatic data partitioning heuristic that makes significant use of array data flow analysis information in the modeling of data interrelationships and the estimation of costs associated with resolving interrelationships via communication. To support the analysis required by the distribution heuristic, we develop the *Distribution Interrelationship Flowgraph (DIF)* representation of a program. The DIF supports data flow analysis and makes explicit those interrelationships that impact data distribution decisions. The global view of data interrelationships provided by data flow analysis permits consideration of potential communication optimizations before data partitioning decisions are made. This allows the partitioning heuristic to choose a data distribution that resolves data interrelationships whose communication costs would be highest if communication were required. Our heuristic uses tiling techniques [16] to determine data partitionings. The resulting data distributions, while still regular, are not limited to the standard BLOCK, CYCLIC and BLOCK_CYCLIC varieties. Experimental results indicate an overall reduction in communication cost with our technique.

The technique presented in this paper differs from existing dynamic distribution techniques in two significant ways. First, the interrelationships that affect the partitioning of an array at a program point are extracted from global data flow analysis information. Thus the interrelationships discovered are value-centric rather than location-centric and span the lifetime of the values rather than their use in a localized code segment. The precision of array analysis information realizable using interval data flow analysis techniques [9] in comparison to other methods of dependence detection (e.g., dependence vectors) enables some false interrelationships to be excluded from consideration in partitioning decisions. The nature of the interrelationship information impacts the philosophy underlying our heuristic. A data distribution refers to the partitioning of a group of values generated at a single program point and exists until those values are no longer live. This group of values may span an entire array or some regular section of the array. Distribution of array locations is an implicit side effect of the computation of new values into the locations. The computation of the new values is carried out on their owner processor(s) rather than on the owner(s) of previous values. This approach avoids the need to communicate the newly generated values if there is a change in ownership. Our technique assigns a distribution to each section of array values generated during the program's execution. Disjoint sections of the same array may have different distributions at the same program point. Decisions to redistribute array locations are implicit in assigning distributions to the sections.

The second difference in our technique concerns the integration of communication optimization information into the communication cost estimates that drive data partitioning decisions. The applicability of many communication optimizations is determined from reaching definitions data flow analysis information. This is the same data flow information used to determine data interrelationships. Thus, it is known before partitioning decisions are made whether or not certain communication optimizations can be exploited if communication of values is necessary. The savings realizable by those optimizations are incorporated into the communication cost estimates. Consequently, our technique aligns and orients the programs arrays so that those data interrelationships that will incur the highest communication costs are satisfied by the data distribution.

The remainder of this paper is organized as follows. Section 2 discusses the intermediate representation and communication cost estimates used by the partitioning heuristic. Section 3 presents our data partitioning heuristic. Section 4 summarizes experimental results. Conclusions are presented in Section 5.

## 2 Program Representation and Communication Costs

A program representation that supports automatic data distribution driven by array data flow analysis information must have two primary characteristics.

First, the representation must include control flow information to support the performance of data flow analysis. Secondly, the representation must make explicit those interrelationships that affect data partitioning decisions. Such interrelationships result both from the use of multiple variables within a single expression and from multiple uses of the same variable in different expressions. We have developed the *Distribution Interrelationship Flow Graph (DIF)* to meet both of these requirements.

The DIF is based upon the Control Flow Graph (CFG) representation of a program. This CFG basis enables data flow analysis to be performed and also allows the DIF to be used for phases of compilation subsequent to automatic data distribution (e.g., code generation). The CFG constructed by our heuristic is an extension of the traditional CFG in that *merge nodes* are inserted at the join points of alternate control paths. While these merge nodes in no way affect the semantics represented, they allow for the establishment of a single last distribution point for each array value in much the same manner as the *phi* nodes of Static Single Assignment form [8] provide for single last definition points. This approach serves to constrain set sizes during data flow analysis and to simplify automatic generation of communication statments at the expense of some realignment/redistribution communication at the merge nodes associated with conditional constructs. The CFG with merge nodes is referred to as the *Extended Control Flow Graph (ECFG)* representation to distinguish it from the traditional CFG representation.

Efficient generation of data flow analysis information is supported in the DIF by the addition of a set of *distribution flow edges* associated with each of a program's array variables. Distribution flow edges bypass single-entry-single-exit regions of the ECFG where an array variable is not accessed. Placement of distribution flow edges is similar to the placement of dependence flow edges in the Dependence Flow Graph (DFG) [11]. The subgraph induced by the distribution flow edges for an array variable $A$ is termed the *Distribution Control Flow Graph of A* ($DCFG_A$).

Explicit representation of data interrelationships in the DIF comes through the extension of the ECFG with a set of *relationship edges*. These undirected edges connect nodes in the ECFG which represent distribution points of array sections. An edge between node $n$ and node $m$ indicates that the distribution of the array section that originates at node $n$ interacts with the distribution of the array section that originates at node $m$ at some program point. Since each array definition point in the program is considered to be a distribution point, reaching definitions data flow information is used to determine the placement of relationship edges. The undirected subgraph induced by the set of relationship edges is termed the *Interrelationship Graph (IG)* of the program.

Relationship edges are labeled with ordered pairs consisting of nodes and costs. In each ordered pair, the first element is the ECFG node number at which an interrelationship represented by the edge occurs. This information is needed so that the partitioning heuristic can retrieve information about the specific reference functions involved in the interrelationship. The second element of each ordered pair is an estimate of the cost incurred if the interrelationship needs to be resolved via communication. It is assumed that the send statement of the communication is moved as far away from the actual access as possible. If the communication can be initiated outside any loops in which the access is nested, its cost is adjusted to reflect a savings realizable through message aggregation. The cost also incorporates an estimated savings attributable to the overlap of communication with computation between the send and receive statements.

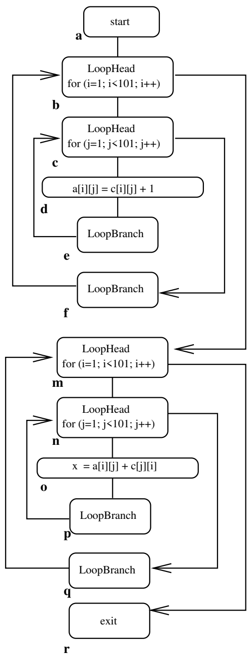
Figure 1b shows the ECFG for the program example in Fig. 1a. Nodes of the ECFG have been labeled for ease of reference. The boldfaced, undirected edges are the relationship edges of the DIF and are discussed below. Figure 1c shows the subgraph induced by the distribution flow edges for array variable $C$. Since the middle loop nest of the program does not reference array $C$, $DCFG_C$ does not include any of the ECFG associated with that loop.

The relationship edges in Fig. 1b are labeled only with the ECFG node names where the relationships originate. Communication costs are discussed below. Since arrays C and D are not assigned values before they are used, it is assumed that their distributions occur at the *start* node prior to the first loop. The use of elements of array $C$ to compute elements of array $A$ at node **d** results in the generation of a relationship edge $(a, d)$ labeled **d**. In an analogous manner, edges $(d, i)$ and $(a, j)$ with labels **i** and **j** are inserted. The final interrelationship in the example results from the use of elements from arrays $A$ and $C$ at node **o**. Since relationship edges are placed between the distribution points of the arrays involved in an interrelationship, the appropriate edge in this instance is $(a, j)$. Since edge $(a, j)$ already exists, it is given a second label, **o**, to indicate another interrelationship involving the same distribution points. When the IG is further processed by the partitioning heuristic, the single physical node, **a**, is treated as two nodes, $a_C$ and $a_D$, and the single physical edge, $(a, j)$, is treated as two logical edges, $(a_C, j)$ and $(a_D, j)$, since two different sets of variables are involved in the interrelationships labeling the edge.

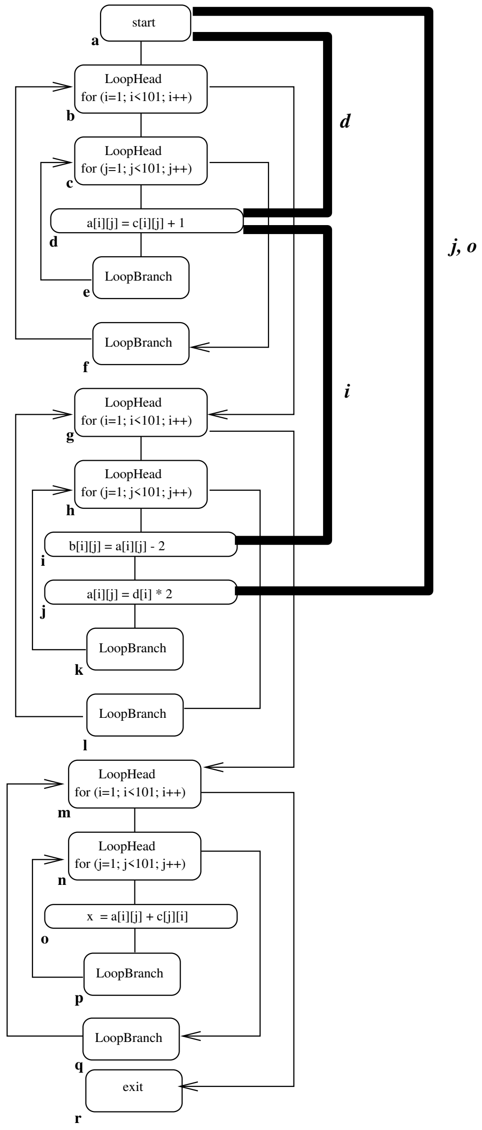
Relationship edges are labeled with symbolic communication cost estimates derived for each relationship. As an example of these costs, consider relationship edge $(d, i)$ labeled **i**. This interrelationship arises from the use of $a[i][j]$ to define $b[i][j]$ at node **i**. Since process decomposition is accomplished via the owner computes rule [5], the values of $a[i][j]$ will be communicated if the distribution does not resolve this interrelationship. Given that the values of $a[i][j]$ are defined prior to the loop nest in which node **i** occurs, the values involved in this communication can be aggregated and sent prior to the start of the loop nest. If there are $p$ processors available, this results in $p(p-1)$ messages, each of length $\frac{10000}{p(p-1)}$. The expression $Transfer(\frac{10000}{p(p-1)})$ represents the time required for one of these messages. The send statement of the communication is scheduled immediately after the loop in which the values of $a[i][j]$ are computed. Thus, the communication cost is adjusted to reflect the savings due to overlapping the communication with the computation that occurs between the send statement and node **i**. This savings is represented as the expression $time(f, i)$. Thus, the symbolic communication cost derived for edge $(d, i)$ is $p(p-1) * Transfer(\frac{10000}{p(p-1)}) - time(d, i)$.

```
for (i=1; i<1011 u++)
  for (j=1; j<101; j++)
    a[i][j] = c[i][j] + 1;
for (i=1; i<101; i++)
  for (j=1; j<101; j++)
  {
    b[i][j] = a[i][j] - 2;
    a[i][j] = d[i] * 2;
  }
for (i=1; i<101; i++)
  for (j=1; j<101; j++)
    x = a[i][j] + c[j][i]
```

**(a)**




**(c)**




**(b)**

**Figure 1.** The Distribution Interrelationship Flowgraph

## 3 The Partitioning Heuristic

**Determining relationship clusters.** The data distribution heuristic begins by partitioning the IG into groups of related nodes termed *relationship clusters*. Each cluster is built by starting with the ECFG node closest to the ECFG end node that is not already in a cluster. An adjacent node, *d*, is added to the cluster if two conditions are met. First, $d$ must not belong to another cluster. This condition ensures that each value is owned by a single processor at each program point. Secondly, if there is another node, $e$, already in the cluster that represents values also represented by $d$, then $d$ will only replace $e$ in the cluster if the highest cost interrelationship involving node $d$ is greater than the highest cost interrelationship involving node $e$. This condition enables each definition point of a value to be considered as a potential redistribution point for the array locations defined there. This second condition also establishes that lower cost interrelationship edges will fall between clusters while higher cost edges will fall within clusters. Those edges between relationship clusters are not considered as data distributions are derived. Therefore it is likely that those relationships between clusters will have to be resolved via communication and thus should carry as small a cost as possible. If node $d$ is added to the cluster, nodes adjacent to $d$ in the IG are also considered for inclusion. The cluster is complete when there are no more nodes that satisfy the conditions.

All nodes in a cluster are mapped to a single virtual data space. Selecting a distribution for the virtual data space implicitly selects a distribution for all array sections mapped to it. Specification of a section's orientation is represented by a function $\pi$ such that dimension $i$ of the section is mapped to dimension $\pi(i)$ of the virtual data space. The alignment within a dimension is specified by a linear function, $f(x) = ax + b$, where $a$ specifies the stride within the virtual data space between adjacent points of the section and $b$ represents the offset of each point of the section.

**Determining alignment and orientation.** If the alignment and orientation of an array section represented by a node in a relationship cluster are known, the interrelationship labeling an edge to an adjacent node in the cluster can be used to determine a communication-free alignment and orientation of the array section of the adjacent node. Assume section $s_1$ represented by node $n_1$ has permutation function $\pi_1$ and alignment $f_{1,i}$ in dimension $i$. Let $s_2$ be the section represented by adjacent node $n_2$ in the relationship cluster. Assume that in the relationship labeling edge $(n_1, n_2)$ that $s_1$ is accessed by the function $R_1$ in dimension $i$ and that $R_1$ references index variable $I$. If $s_2$ is accessed in dimension $j$ by function $R_2$ involving the same index variable $I$, then to avoid communication the orientation function $\pi_2$ of $s_2$ should specify that $\pi_2(j) = \pi_1(i)$. With respect to alignment, each coordinate in dimension $i$ of $s_1$ can be mapped to a corresponding point in dimension $j$ of $s_2$ via the function composition $R_2R_1^{-1}$. In order to avoid communication, these corresponding points of $s_1$ and $s_2$ must also coincide in the virtual data space. Thus, the alignment function $f_{2,j}$ is defined by the composite function $R_2R_1^{-1}f_{1,i}$.

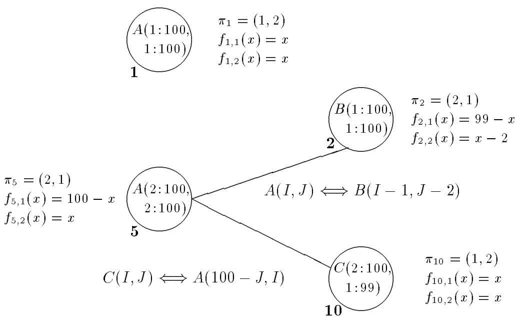
If a relationship cluster is acyclic, the highest cost interrelationship on each edge can be satisfied through proper alignment and orientation of the array sections in the cluster to the virtual data space. If a relationship cluster contains a cycle, then it is possible that two conflicting alignments and orientations are required for a single array section in order to avoid communication. Such conflicts are detected by traversing the biconnected components of each relationship cluster. The starting node of each traversal is arbitrarily aligned and oriented in direct relationship to its virtual data space. As each new node is encountered during the traversal, its alignment and orientation are computed. When a cycle is detected the two partitionings computed for the node are compared. If there is a conflict, the edge in the cycle with the minimum communication cost estimate is removed from the cluster. It is assumed this relationship will be resolved via communication and it is not further considered during the partitioning process.

Figure 2 presents an example of determining alignment and orientation. The partitioned IG for the program segment of Fig. 2a is shown in Fig. 2b. The nodes of the IG are labeled to correspond to the programs statements that define each array section. Only the highest cost interrelationship that affects partitioning decisions is indicated on each edge of the IG. The cluster containing nodes 2, 5 and 10 contains no cycles and therefore no conflicts. Assume that the section at node 10 is arbitrarily given orientation $\pi_{10} = (1, 2)$ and alignment $f_{10,1}(x) = x$ and $f_{10,2}(x) = x$. Given the interrelationship $C(I, J) \Longleftrightarrow A(100 - J, I)$ on edge (10,5), the orientation of the section at node 5 is determined to be $\pi_5 = (2, 1)$. In computing the alignment of dimension two of $A$ with dimension 1 of $C$, $R^{-1}_{10,1} = I$, $R_{5,2} = I$ and $f_{10,1}(x) = x$. Therefore, $f_{5,2}(x) = x$. Similarly, $R^{-1}_{10,2} = J$, $R_{5,1} = 100 - J$ and $f_{10,2}(x) = x$ yielding $f_{5,1}(x) = 100 - x$. The alignment and orientation of the section represented by node 2 are computed from the section at node 5. The section represented by node 1 is arbitrarily given an alignment and orientation that maps each point directly to the virtual data space.

**Selecting tile shape and size.** Once the sections of a relationship cluster have been aligned and oriented to the virtual data space, those interrelationships that have not been satisfied by the alignments and orientations chosen are examined to determine what constraints they place on the distribution of the virtual data space. Whenever two values from an array section (e.g., $B[I, J]$ and

```
(1)   read A
(2)   read B
(3)   for I=2,100
(4)     for J = 2,100
(5)       A(I,J) = A(I,J) + B(I-1, J-2) + B(I,J)
(6)     end for
(7)   end for
(8)   for I=2,100
(9)     for J = 2,99
(10)      C(I,J) = C(I-1,J) + A(100-J, I)
(11)    end for
(12)  end for
```

(a) Example program




(b) Partitioned IG with Alignment/Orientation

**Figure 2.** Example of Determining Alignment and Orientation

$B[I-1, J-1]$) are accessed in the same statement or with a single value from another array section (e.g., $A[I, J]$), then there is a preference for the two values (i.e., $B[I, J]$ and $B[I-1, J-1]$) to reside in the same processor memory. When possible, this preference is represented in the form of a distance vector (e.g., (1,1)) and is termed a *communication dependence*. Since constraints on the distribution must be relative to the virtual data space, communication dependences derived from the reference functions of an interrelationship are transformed based upon the alignments and orientations of the array sections to that space.

Consider the relationship cluster containing nodes 2, 5 and 10 from the example in figure 2. Figure 3 shows this cluster labeled with all interrelationships. The self loop on node 10 results in the creation of a communication dependence, (1,0), specifying that $C(I, J)$ should reside with $C(I-1, J)$. Since array $C$ is directly aligned to the virtual data space, no transformation of this dependence is required. There are 9801 non-invariant data values associated with this dependence. The interrelationships labeling edge (10,5) indicate a preference for the points defined by $A(100-J, I)$ to reside in the same partition as those defined by $A(100-J, I+1)$. This gives rise to the dependence (0, -1). The orientation and alignment functions of array $A$ indicate a transposition of dimensions and inversion in dimension 1. Thus, the transformed communication dependence is the vector (1,0). There are 10000 loop-invariant data values associated with this dependence. The two interrelationships on the edge (5,2) give rise to the dependence (1,2), indicating a preference for the points $B(I-1, J-2)$ and $B(I, J)$ to reside in the same processor memory. The reorientation indicated by $\pi_2$ and the inversion indicated by $f_{2,1}$ transform this communication dependence from (1,2) to (-2,1). There are 10000 loop-invariant values associated with this dependence. The self loop on node 2 gives rise to the communication dependence (1,2). Again, the orientation/alignment of array $B$ results in a transformation of this vector to (-2, 1). There are 9801 loop-invariant values associated with this dependence.

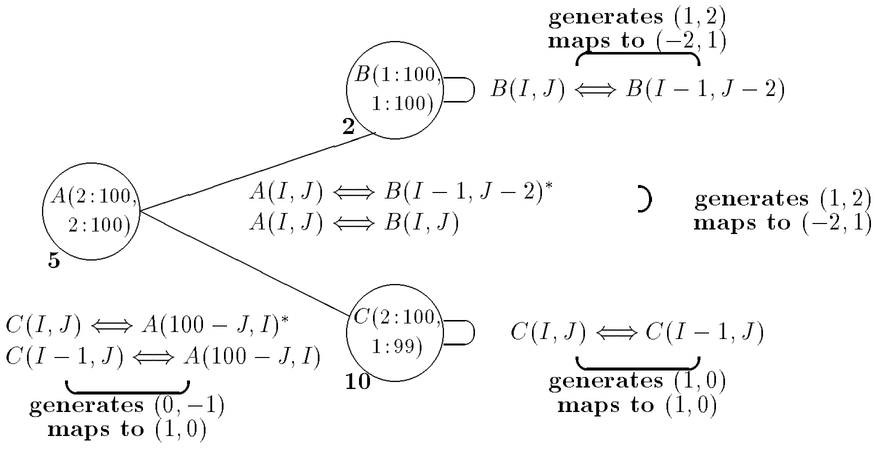



**Figure 3.** Example of Generating Communication Dependences

In order to minimize communication costs, tile boundaries are aligned with communication dependences. The extreme vectors [16] in the set of all dependences for a cluster are used for this purpose so that the tiles formed will encapsulate the most communication. Each tile is constrained to have length equal to some constant multiple of the length of its associated dependence vector. Determination of those constants is based upon estimates of the communication cost and parallelism derived as functions of the unknown constant multipliers. This approach enables the tradeoff between parallelism and communication inherent in some programs to be considered while determining the tile size. The communication cost function is also adjusted to reflect savings due to message aggregation when applicable.

Consider the representative tile shown in Fig. 4. The tile boundaries are formed by two dependences, $a = (a_1, a_2)$ and $b = (b_1, b_2)$, and have angles of inclination denoted by $\theta_a = \tan^{-1}(a_2/a_1)$ and $\theta_b = \tan^{-1}(b_2/b_1)$. The lengths of the sides are given as $c_1\sqrt{a_1^2 + a_2^2}$ and $c_2\sqrt{b_1^2 + b_2^2}$.

The distance vector, $d$, represents a communication dependence in the virtual data space. Each point in the tile is the source of an equivalent dependence. Those points that lie within the inner parallelogram will not incur any communication costs since their respective vectors lie completely within the tile. Those dependences whose source is in the boundary area between the inner and outer parallelograms will require communication to be satisfied.

The total number of communications generated within the tile is equal to the area of the boundary between the parallelograms. This boundary area is divided into three sections such that all of the communication within an area has a common destination. The areas of these sections, $A_1, A_2$ and $A_3$, are expressed as functions of the constants $c_1$ and $c_2$.

If the values being communicated for a given instance $i$ of dependence vector $d$ are loop invariant then those values can be aggregated into three messages of

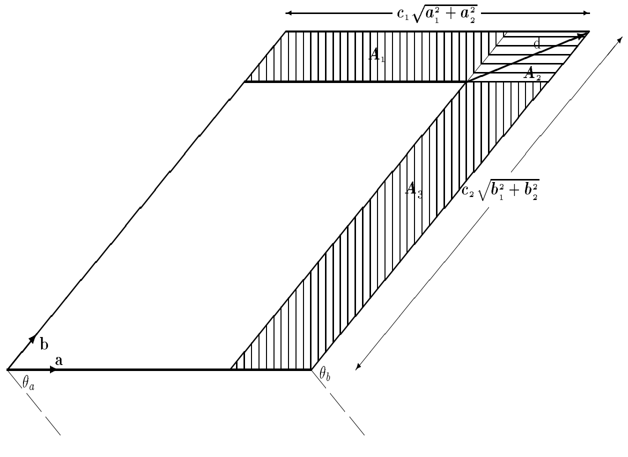



**Figure 4.** Representative Tile for Communication Analysis

sizes $A_1$, $A_2$ and $A_3$. If the values are not loop invariant then $A_1 + A_2 + A_3$ messages of size one data value ($m$ words) are needed. The communication for instance $i$ in a single tile follows.

$$C_t^i = \begin{cases} \sum_{i=1}^{3} Transfer(A_i * m) \text{ if invariant} \\ \sum_{i=1}^{3} A_i * Transfer(m) \text{ otherwise} \end{cases}$$

Assume the size of the data space associated with instance $i$ is $S_i$. Then the number of tiles, $t_i$, that will be in that space is given by

$$t_i = \frac{S_i}{c_1 c_2 \sqrt{a_1^2 + a_2^2}\sqrt{b_1^2 + b_2^2}\sin(\theta_b - \theta_a)}.$$

If $D$ is the set of all communication dependences in the cluster, then the total communication cost over all dependences is

$$C = \sum_{\text{all } d \epsilon D} \sum_{\text{all } i} C_t^i * t_i.$$

The parallelism estimate used is an approximation based upon the size of a tile relative to the size of the virtual data space. Tile boundaries are oriented with the extreme vectors, $a = (a_1, a_2)$ and $b = (b_1, b_2)$, such that they form angles of $\theta_a = \arctan(\frac{a_2}{a_1})$ and $\theta_b = \arctan(\frac{b_2}{b_1})$ with the horizontal boundary of the virtual data space. A diagonal line segment drawn through the virtual data space and coinciding with the diagonal of the tile forms the angle $\theta_d$ with the horizontal boundary of the virtual data space such that

$$\theta_d = \arccos\left(\frac{a_1 + b_1}{\sqrt{(a_1 + b_1)^2 + (a_2 + b_2)^2}}\right)$$

The length of the line segment, $L_d$, passing through the virtual data space at angle $\theta_d$ is given by

$$L_d = \begin{cases} \frac{N_h}{cos(\theta_d)} \text{ if } \mid \theta_d \mid \leq \arctan(\frac{N_v}{N_h}) \\ \frac{N_v}{sin(\theta_d)} \text{ otherwise} \end{cases}$$

where $N_h$ and $N_v$ are the horizontal and vertical lengths of the virtual data space respectively.

The length of the tile diagonal, $l_d$, is given by the equation

$$l_d = \sqrt{(c_1 \bar{a} \cos(\theta_a) + c_2 \bar{b} \cos(\theta_b))^2 + (c_1 \bar{a} \sin(\theta_a) + c_2 \bar{b} \sin(\theta_b))^2}$$

where $\bar{a}$ and $\bar{b}$ are the lengths of vectors $a$ and $b$. The parallelism estimate used is the quotient $P = \frac{L_d}{l_d}$.

Values for the constants $c_1$ and $c_2$ are determined using the method of LaGrange multipliers. Parallelism is constrained so that $P$ is the number of available processors. The communication cost function is viewed as an objective function in multiple variables to be minimized subject to the parallelism constraint.

**Representing the data distribution.** The partitioning of the virtual data space is expressed as families of parallel lines that form the tiles with the chosen orientation and size. Assume that the bottom left corner of the virtual data space is the point $(i_s, j_s)$. The two families of lines that are defined as functions of the constants $k_1, k_2 \;\epsilon Z$ follow.

1. lines with slope $b_2/b_1$ will pass through points
$$(i_s + k_1 \cos(\theta_a) c_1 \sqrt{a_1^2 + a_2^2}, j_s + k_1 \sin(\theta_a) c_1 \sqrt{a_1^2 + a_2^2})$$
2. lines with slope $a_2/a_1$ will pass through points
$$(i_s + k_2 \cos(\theta_b) c_2 \sqrt{b_1^2 + b_2^2}, j_s + k_2 \sin(\theta_b) c_2 \sqrt{b_1^2 + b_2^2})$$

Transformation of these families of lines into a partitioning of a related array section requires two steps. The orientation functions for the array sections must be examined to determine the relative roles of the variables $i$ and $j$ in the two sections. If the sections are transposed with respect to one another in the virtual data space, the roles of $i$ and $j$ must interchange. Secondly, any inversion (i.e., negation of $i$ or $j$) or shifting for alignment purposes requires translation of the lines. This information is available as the inverse of the alignment function that mapped each array section to the virtual data space.

## 4 Experimental Results

To assess the effectiveness of our heuristic, we have performed an experimental evaluation using a communication emulation system [18] that gathers statistics about the actual communication resulting from a series of send statements issued by a program. The emulator is implemented as a C library routine that simulates communication in a set of physical processors. A client program of the emulator invokes the library's primary function at the point of each send statement, passing parameters of the communication in the form of a *Section Communication Descriptor (SCD)* [18]. The SCD includes the name of the array, a bounded regular section descriptor defining which elements are to be communicated and a description of the mappings of the elements to both the source and destination processors of the communication. A description of the virtual-to-real processor mapping is provided. The emulator enumerates each point that is to be communicated and determines if the source and destination processors for the point differ. The total number of points that actually result in communication and the number of messages required to send those points after aggregation opportunities are exploited is recorded. The quotient of the number of data points divided by the number of messages is also reported as a measure of the average message length.

Five test programs were chosen for evaluation of the partitioning heuristic. The program kernel ARTDIF is taken from the Hydro2D program of the SPEC95 benchmark suite. It is an astrophysical program for the computation of galactical jets using hydrodynamical Navier Stokes equations. The tomcatv grid generation program was also taken from the SPEC95 benchmark suite. The third program contains a Conjugate Gradient solver. The alternating-direction implicit (adi) program solves a tridiagonal system using cyclic reduction. Swim is a weather prediction program from the SPEC95 benchmark suite. Its procedures have been inlined to produce a single procedure for this experimentation.

The experimental evaluation compares the results of the communication emulator for each of the five test programs when compiled with the dynamic distribution technique and with a static distribution technique followed by a communication optimization phase. The compiler used to generate and optimize communications given a static data distribution is an extension to the Stanford SUIF compiler [1]. A data flow analysis based communication optimizer then performs message aggregation, redundant communication emulation and global message scheduling. For each of the five test programs, multiple static distributions were tested. The chosen distribution for each program was one that had the lowest communication cost while enabling an acceptable level of parallelism.

Figure 5 summarizes the output of the communication emulator. The results indicate a reduction in the number of messages generated and the total number of data points communicated for three of the five test programs. In the program ARTDIF, the distributions chosen by the dynamic technique are virtually identical to those chosen for the static distribution. Consequently, communication savings in this program cannot be contributed to choosing better distributions. Instead, the savings can be traced to more effective redundant communication elimination in the dynamic technique. As the dynamic heuristic creates a list of potential communications, each new entry is compared to the other entries in the list. If two different sections of the same array are both defined and used at the same program point, a union operation is performed and the intersection of the sections appears only once in the list. The data flow based communication optimizer did not uncover such redundant communications. As can be seen from the results of ARTDIF, such redundancy may not significantly impact the total number of messages required but dramatically increases the total number of data points communicated. Thus the average message length of the dynamic technique is much smaller, indicating a savings in communication cost.

The program tomcatv has two significant sources of communication savings in the dynamic distribution approach. First, some parallelism is sacrificed by the heuristic when loop-invariant communications arise in order to reduce communication costs. Secondly, the largest cluster of definition points created by the heuristic exhibits several non-vertical, non-horizontal communication dependences. The generalized tiling approach allows the set of extreme vectors to better encompass those dependences than a standard BLOCK tiling would.

The conjugate gradient program alternately accesses several one-dimensional arrays with each other and with a single two-dimensional array. When the program's one-dimensional arrays are distributed for access with the two-dimensional array, the block sizes of the one-dimensional arrays are chosen to match that of the two-dimensional array. Whenever the one-dimensional arrays are reinitialized

| Program | total messages | total values communicated | average message size |
|---|---|---|---|
| ARTDIF (static) | 18 | 1470576 | 81698.664 |
| ARTDIF (dynamic) | 16 | 1124 | 70.25 |
| tomcatv (static) | 10804 | 13095388 | 1212.087 |
| tomcatv (dynamic) | 3896 | 9666462 | 2480.611 |
| adi (static) | 23544 | 124692 | 5.296 |
| adi (dynamic) | 27700 | 1667354 | 60.123 |
| conjugate gradient (static) | 669 | 330735 | 495.372 |
| conjugate gradient (dynamic) | 243 | 3384 | 13.296 |
| swim (static) | 2469 | 3994204 | 1617.742 |
| swim (dynamic) | 4379 | 20072364 | 4583.778 |

**Figure 5.** Results of the communication emulator for programs compiled using static and dynamic data distribution techniques

for use with each other exclusively, the block sizes are reduced so that more parallelism can be exploited. In the static case, matching one-dimensional block sizes to those of the two dimensional arrays restricts available parallelism at program points where the two-dimensional array is not accessed. Consequently, a static distribution must incur higher communication costs on the elements of the one-dimensional arrays when accessed in conjunction with the two-dimensional array.

The final two test programs in the suite performed more poorly when compiled using the dynamic distribution technique than when a static distribution was used. In the case of adi, this degradation can be traced to a common type of access pattern for array sections involving two adjacent loop nests. The first loop nest initializes an array section and the second loop nest reuses those initial values to redefine the same array elements. The use of the elements in the second loop is the only occurrence of a use for the values of the section in the first loop. However, since the heuristic demands that two definition points for the same array elements cannot be in the same partitioning cluster, the possibility exists that the two references to $x[i][j]$ in the second loop refer to values on different processors. This happened several times in this program. While any inter-cluster relationship edge has a high probability of requiring communication to resolve the interrelationship it represents, a case like the above is unnecessarily detrimental to the performance of the program.

In the case of the swim program, there are two reasons the dynamic heuristic performed poorly. First, swim has a significant number of program statements not contained in loops that assign to single array elements. The present implementation is ineffective at aligning and orienting clusters containing significant numbers of array sections with constant reference functions. A more significant source of increased communication in swim comes from a high number of messages that are generated at the merge point of a conditional control construct. If the last distribution point of an array along a control path of a conditional is given a partitioning different than the one assigned to the section at the merge node, the values must be redistributed at the merge node. Unlike redistributions upon assignment, explicit communication must be inserted to move those values. Furthermore, any values distributed by the merge node may require separate communication to resolve other interrelationships. The end result is that many points may be redistributed, only to be immediately communicated to yet another processor. Clearly this is inefficient. Extensions to the heuristic must consider the possibility of developing communication optimizations aimed at reducing communication costs at the merge points of alternate control paths.

## 5 Conclusion and Future Work

This paper details the development of an automatic data distribution heuristic for distributed memory parallel architectures. Through the use of global data flow analysis information and a program representation that makes explicit those data interrelationships that impact the level of communication required by a program, the heuristic selects data distributions and performs communication optimizations. Experimental results indicate a reduction in communication requirements for programs compiled using our heuristic when compared with programs compiled using a static distribution technique followed a data flow analysis based communication optimization phase.

Comparison of the effectiveness of our approach relative to other dynamic data distribution heuristics is a topic of future work. A code generator must be developed to allow execution statistics to be generated to support this work. Additional future work will center on the development of communication optimizations techniques applicable at the merge points of conditional constructs.

## References


1. S.P. Amarashinge, J.M. Anderson, M.S. Lam, and C.W. Tseng. "The SUIF Compiler for Scalable Parallel Machines". In *Proceedings of the Seventh SIAM Conference on Parallel Processing for Scientific Computing*, Feb. 1995.
2. Jennifer M. Anderson and Monica S. Lam. "Global Optimizations for Parallelism and Locality on Scalable Parallel Machines". In *ACM SIGPLAN '93 Conference on Programming Language Design and Implementation*, pages 112–125, Albuquerque, NM, Jun. 1993.
3. Eduard Ayguade, Jordi Garcia, Merce Girones, M. Luz Grande, and Jesus Labarta. "Data Redistribution in an Automatic Data Distribution Tool. In *Proceedings of the 8th Workshop on Languages and Compilers for Parallel Computing*, pages 407–421, Columbus, Ohio, Aug. 1995.
4. R. Bixby, K. Kennedy, and U. Kremer. "Automatic Data Layout Using 0-1 Integer Programming". In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT94)*, pages 111–122, Montreal, Canada, Aug. 1994.
5. D. Callahan and K. Kennedy. "Compiling Programs for Distributed-Memory Multiprocessors". *The Journal of SuperComputing*, 2, 1988.
6. Barbara M. Chapman, Thomas Fahringer, and Hans P. Zima. "Automatic Support for Data Distribution on Distributed Memory Multiprocessor Systems". In *Proceedings of the 6th Workshop on Languages and Compilers for Parallel Computing*, pages 184–199, Portland, Oregon, Aug. 1993.

7. S. Chatterjee, J. R. Gilbert, R. Schreiber, and T.J. Sheffler. "Array Distribution in Data-Parallel Programs". In *Proceedings of the 7th Workshop on Languages and Compilers for Parallel Computing*, pages 76–91, 1994.
8. Ron Cytron, Jeanne Ferrante, Barry Rosen, Mark N. Wegman, and F. Kenneth Zadeck. "Efficiently Computing Static Single Assignment Form and the Control Dependence Graph". In *ACM Transactions on Programming Languages and Systems*, volume 13, pages 451–490, Oct. 1991.
9. Thomas Gross and Peter Steenkiste. "Structured Dataflow Analysis for Arrays and its Use in an Optimizing Compilers". *Software Practice and Experience*, 20(2):133–155, Feb. 1990.
10. Manish Gupta and Prithviraj Banerjee. "Demonstration of Automatic Data Partitioning Techniques for Parallelizing Compilers on Multicomputers". *IEEE Transactions on Parallel and Distributed Systems*, 3(2):179–193, Mar. 1992.
11. Richard Johnson and Keshav Pingali. "Dependence-Based Program Analysis". In *ACM SIGPLAN '93 Conference on Programming Language Design and Implementation*, pages 78–89, Jun. 1993.
12. M. Kandemir, J. Ramanujam, and A. Choudhary. "Compiler Algorithms for Optimizing Locality and Parallelism on Shared and Distributed Memory Machines". In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT97)*, San Francisco, CA, Nov. 1997.
13. Kathleen Knobe, Joan D. Lukas, and Guy L. Steele Jr. "Data Optimization: Allocation of Arrays to Reduce Communication on SIMD Machines". *Journal of Parallel and Distributed Computing*, 8:102–118, 1990.
14. Uli Kremer. "NP-completeness of Dynamic Remapping". In *Proceedings of the Fourth Workshop on Compilers for Parallel Computers*, The Netherlands, Dec. 1993.
15. Daniel J. Palermo and Prithviraj Banerjee. "Automatic Selection of Dynamic Data Partitioning Schemes for Distributed-Memory Multipcomputers". In *Proceedings of the 8th Workshop on Languages and Compilers for Parallel Computing*, pages 392–406, Columbus, OH, Aug. 1996.
16. J. Ramanujam and P. Sadayappan. "Nested Loop Tiling for Distributed Memory Machines". In *5th Distributed Memory Computing Conference*, pages 1088–1096, Charleston, SC, Apr. 1990.
17. Thomas J. Sheffler, Robert Schreiber, William Pugh, John R. Gilbert, and Siddhartha Chatterjee. "Efficient Distribution Analysis via Graph Contraction". In *Frontiers '95: The 5th Symposium on the Frontiers of Massively Parallel Computation*, pages 377–391, McLean, VA, Feb. 1995.
18. X. Yuan, R. Gupta, and R. Melhem. "An Array Data Flow Analysis based Communication Optimizer". In *Proceedings of the 10th Workshop on Languages and Compilers for Parallel Computing*, Minneapolis, MN, Aug. 1997.

This article was processed using the LaTeX macro package with LLNCS style