Compile-Time Techniques for Data Distribution in Distributed Memory Machines

J. Ramanujam
Department of Electrical and Computer Engineering
Louisiana State University, Baton Rouge, LA 70803-5901

P. Sadayappan
Department of Computer and Information Science
The Ohio State University, Columbus, OH 43210-1277

Abstract

This paper addresses the problem of partitioning data for distributed memory machines (multicomputers). In current day multicomputers, interprocessor communication is more time-consuming than instruction execution. If insufficient attention is paid to the data allocation problem, then the amount of time spent in interprocessor communication might be so high as to seriously undermine the benefits of parallelism. It is therefore worthwhile for a compiler to analyze patterns of data usage to determine allocation, in order to minimize interprocessor communication. We present a machine-independent analysis of communication-free partitions. We present a matrix notation to describe array accesses in fully parallel loops which lets us derive sufficient conditions for communication-free partitioning (decomposition) of arrays. In the case of a commonly occurring class of accesses, we present a problem formulation to minimize communication costs, when communication-free partitioning of arrays is not possible.

Keywords: Distributed memory machines, parallelizing compilers, data decomposition, access and dependence based array partitioning, communication-free partitioning.

Appears in IEEE Transactions on Parallel and Distributed Systems, Volume 2, Number 4, October 1991 (pages 472–482).

1 Introduction

In distributed memory machines such as the Intel iPSC/2 and NCUBE, each process has its own address space and processes must communicate explicitly by sending and receiving messages. Local memory accesses on these machines are much faster than those involving inter-processor communication. As a result, the programmer faces the enormously difficult task of **orchestrating** the entire parallel execution. The programmer is forced to manually distribute code and data in addition to managing communication among tasks explicitly. This task, in addition to being error-prone and time-consuming, generally leads to non-portable code. Hence, parallelizing compilers for these machines have been an active area of research recently [3, 4, 5, 9, 14, 19, 20, 22, 24, 25].

The enormity of the task is to some extent relieved by the *hypercube programmer's paradigm* [6] where attention is paid to the partitioning of tasks alone, while assuming a fixed data partition or a programmer-specified (in the form of annotations) data partition [9, 7, 14, 20]. A number of efforts are under way to develop parallelizing compilers for multicomputers where the programmer specifies the data decomposition and the compiler generates the tasks with appropriate message passing constructs [4, 7, 14, 15, 20, 22, 25]. Though these rely on the intuition (based on domain knowledge) of the programmer, it is not always possible to verify that the annotations indeed result in efficient execution.

In a recent paper on programming of multiprocessors, Alan Karp [12] observes:

> " ... we see that data organization is the key to parallel algorithms even on shared memory systems ... The importance of data management is also a problem for people writing automatic parallelization compilers. Todate, our compiler technology has been directed toward optimizing control flow ... Even today when hierarchical (distributed) memories make program performance a function of data organization, no compiler in existence changes the data addresses specified by the programmer to improve performance. If such compilers are to be successful, particularly on message-passing systems, a new kind of analysis will have to be developed. This analysis will have to match the data structures to the executable code in order to minimize memory traffic."

This paper is an attempt at providing this "new" kind of analysis – we present a matrix notation to describe array accesses in fully parallel loops which lets us present sufficient conditions for communication-free partitioning (decomposition) of arrays. In the case of a commonly occurring class of accesses, we present a formulation as a fractional integer programming problem to minimize communication costs, when communication-free partitioning of arrays is not possible.

The rest of the paper is organized as follows. In section 2, we present the background and the assumptions we make, and discuss related work. Section 3 illustrates through examples, the importance and the difficulty of finding good array decompositions. In section 4, we present a matrix-based formulation of the problem of determining the existence of communication-free partitions of arrays; we then present conditions for the case of constant offset array access. In section 5, a series of examples are presented to illustrate the effectiveness of the technique for linear references; in addition, we show the use of loop transformations in deriving the necessary data decompositions. Section 6 generalizes the formulation presented in section 4 for arbitrary linear references. In section 7, we present a formulation that aids in deriving heuristics for minimizing communication when communication-free partitions are not feasible. Section 8 concludes with a summary and discussion.

2 Assumptions and related work

Communication in message passing machines could arise from the need to synchronize and from the non-locality of data. The impact of the absence of a globally shared memory on the compiler writer is severe. In addition to managing parallelism, it is now essential for the compiler writer to appreciate the significance of data distribution and decide when data should be copied or generated in local memory. We focus on distribution of arrays which are commonly used in scientific computation. Our primary interest is in arrays accessed during the execution of nested loops. We consider the following model where a processor owns a data element and has to make all updates to it and there is exactly one copy. Even in the case of fully parallel loops, care must be taken to ensure appropriate distribution of data.

In the next sections, we explore techniques that a compiler can use to determine if the data can be distributed such that no communication is incurred. Operations involving two or more operands require that the operands be *aligned,* that is the corresponding operands are stored in the memory of the processor executing the operation. In the model we consider here, this means that the operands used in an operation must be communicated to the processor that holds the operand which appears on the left hand side of an assignment statement. Alignment of operands generally requires interprocessor communication.

In current day machines, interprocessor communication is more time-consuming than instruction execution. If insufficient attention is paid to the data allocation problem, then the amount of time spent in interprocessor communication might be so high as to seriously undermine the benefits of parallelism. It is therefore worthwhile for a compiler to analyze patterns of data usage to determine allocation, in order to minimize interprocessor communication. We present a machine-independent analysis of communication-free partitions.

We make the following assumptions:

1. There is exactly one copy of every array element and the processor in whose local memory

the element is stored is said to "own" the element.

2. The owner of an array element makes all updates to the element, *i.e.* all instructions that modify the value of the element are executed by the "owner" processor.

3. There is a fixed distribution of array elements. (Data re-organization costs are architecture-specific)

2.1 Related work

The research on problems related to memory optimizations goes back to studies of the organization of data for paged memory systems [1]. Balasundaram and others [3] are working on interactive parallelization tools for multicomputers that provide the user with feedback on the interplay between data decomposition and task partitioning on the performance of programs. Gallivan et al. [7] discuss problems associated with automatically restructuring data so that it can be moved to and from local memories in the case of shared memory machines with complex memory hierarchies. They present a series of theorems that enable one to describe the structure of disjoint sub-lattices accessed by different processors, use this information to make "correct" copies of data in local memories, and write the data back to the shared address space when the modifications are complete. Gannon et al. [8] discuss program transformations for effective complex-memory management for a CEDAR-like architecture with a three-level memory. Gupta and Banerjee [10] present a constraint-based system to automatically select data decompositions for loop nests in a program. Hudak and Abraham [11] discuss the generation of rectangular and hexagonal partitions of arrays accessed in sequentially iterated parallel loops. Knobe *et al.* [13] discuss techniques for automatic layout of arrays in a compiler targeted to SIMD architectures such as the Connection Machine computer system. Li and Chen [17] (and [5]) have addressed the problem of *index domain alignment* which is that of finding a set of suitable alignment functions that map the index domains of the arrays into a common index domain in order to minimize the communication cost incurred due to data movement. The class of alignment functions that they consider primarily are *permutations* and *embeddings.* The kind of alignment functions that we deal with are more general than these. Mace [18] proves that the problem of finding optimal data storage patterns for parallel processing (the "shapes" problem) is NP-complete, even when limited to one- and two-dimensional arrays; in addition, efficient algorithms are derived for the shapes problem for programs modeled by a directed acyclic graph (DAG) that is derived by series-parallel combinations of tree-like subgraphs. Wang and Gannon [23] present a heuristic state-space search method for optimizing programs for memory hierarchies.

In addition several researchers are developing compilers that take a sequential program augmented with annotations that specify data distribution, and generate the necessary communication.

Koelbel et al. [14, 15] address the problem of automatic process partitioning of programs written in a functional language called BLAZE given a user-specified data partition. A group led by Kennedy at Rice University [4] is studying similar techniques for compiling a version of FORTRAN for local memory machines, that includes annotations for specifying data decomposition. They show how some existing transformations could be used to improve performance. Rogers and Pingali [20] present a method which, given a sequential program and its data partition, performs task partitions to enhance locality of references. Zima et al. [25] have developed SUPERB, an interactive system for semi-automatic transformation of FORTRAN programs into parallel programs for the SUPRENUM machine, a loosely-coupled hierarchical multiprocessor.

3 Deriving good partitions: some examples

Consider the following loop:

Example 1:
for $i = 1$ to N
  for $j = 4$ to N
    $A[i,j] \leftarrow B[i,j-3] + B[i,j-2]$

If we allocate row i of array A and row i of array B to the local memory of the same processor, then no communication is incurred. If we allocate by columns or blocks, interprocessor communication is incurred. There are no data dependences in the above example; such loops are referred to as DOALL loops. It is easy to see that allocation by rows would result in zero communication since, there is no offset in the access of B along the first dimension. Figure 1 shows the partitions of arrays A and B.

In the next example, even though there is a non-zero offset along each dimension, communication-free partitioning is possible:

Example 2:
for $i = 2$ to N
  for $j = 3$ to N
    $A[i,j] \leftarrow B[i+1,j+2] + B[i+2,j+1]$

In this case, row, column or block allocation of arrays A and B would result in interprocessor communication. In this case, if A and B are partitioned into a family of parallel lines whose equation is $i + j =$ constant, *i.e.* anti-diagonals, no communication will result. Figure 2 shows the partitions of A and B. The kth line in array A *i.e.* the line $i + j = k$ in A and the line $i + j = k + 3$ in array B must be assigned to the same processor

Figure 1: Partitions of arrays A and B for Example 1

Figure 2: Partitions of arrays A and B for Example 2

In the above loop structure, array A is modified as a function of array B; a communication-free partitioning in this case is referred to as a *compatible* partition. Consider the following loop skeleton:

Example 3:
for $i = 1$ to N
L_1 : for $j = 1$ to N
    $A[i,j] \leftarrow f(B[i,j])$
  ...
  ...
L_2 : for $j = 1$ to N
    $B[i,j] \leftarrow f(A[i,j])$

Array A is modified in loop L_1 as a function of elements of array B while loop L_2 modifies array B using elements of array A. Loops L_1 and L_2 are adjacent loops at the same level of nesting. The effect of a poor partition is exacerbated here since every iteration of the outer loop suffers inter-processor communication; in such cases, the communication-free partitioning where possible, is extremely important. Communication-free partitions of arrays involved in adjacent loops is referred to as *mutually compatible* partitions.

In all the preceding examples, we had the following array access pattern: for computing some element $A[i,j]$, element $B[i',j']$ is required where

$$i' = i + c_i \qquad \text{and} \qquad j' = j + c_j$$

where c_i and c_j are constants. Consider the following example:

Example 4:
for $i = 1$ to N
  for $j = 1$ to N
    $A[i,j] \leftarrow B[j,i]$

In this example, allocation of **row** i of array A and **column** i of array B to the same processor would result in no communication. See Figure 3 for the partitions of A and B in this example. Note that i' is a function of j and j' is a function of i here.

In the presence of arbitrary array access patterns, the existence of communication-free partitions is determined by the connectivity of the data access graph, described below. To each array element that is accessed (either written or read), we associate a node in this graph. If there are k different arrays that are accessed, this graph has k groups of nodes; all nodes belonging to a given group are elements of the same array. Let the node associated with the left hand side of an assignment statement S be referred to as *write(S)* and the set of all nodes associated with the array elements on

Figure 3: Array partitions for Example 4

the right hand side of the assignment statement S be called *read-set(S)*. There is an edge between *write(S)* and every member of *read-set(S)* in the data access graph. If this graph is connected, then there is no communication-free partition [19].

4 Reference-based data decomposition

Consider a nested loop of the following form which accesses arrays A and B:

Example 5:
for $i = 1$ to N
  for $j = 1$ to N
    $A[i, j] \leftarrow B[i^{'}, j^{'}]$

---

where $i^{'}$ and $j^{'}$ are linear functions of i and j, *i.e.*

$$i^{'} = f(i,j) = a_{11}i + a_{12}j + a_{10} \tag{1}$$

$$j^{'} = g(i,j) = a_{21}i + a_{22}j + a_{20} \tag{2}$$

With disjoint partitions of array A, can we find corresponding or required disjoint partitions of array B in order to eliminate communication? A partition of B is required for a given partition of A if the elements in that partition (of B) appear in the right hand side of the assignment statements in the body of the loop which modify elements in the partition of A. For a given partition of A, the required partition(s) in B is(are) referred to as its image(s) or map(s). We discuss array partitioning in the context of fully parallel loops. Though the techniques are presented for 2-dimensional arrays, these generalize easily to higher dimensions.

In particular, we are interested in partitions of arrays defined by a family of parallel hyperplanes; such partitions are beneficial from the point of view of restructuring compilers in that the portion of loop iterations that are executed by a processor can be generated by a relatively simple transformation of the loop. Thus the question of partitioning can be stated as follows: Can we find partitions induced by parallel hyperplanes in A and B such that there is no communication? We focus our attention on 2-dimensional arrays. A hyperplane in 2 dimensions is a line; hence, we discuss techniques to find partitions of A and B into parallel lines that incur zero communication. In most loops that occur in scientific computation, the functions f and g are linear in i and j.

The equation

$$\alpha i + \beta j = c \tag{3}$$

defines a family of parallel lines for different values of c, given that α and β are constants and at most one of them is zero. For example,

$$\alpha = 0, \qquad \beta = 1$$

defines columns, while

$$\alpha = -1, \qquad \beta = 1$$

defines diagonals.
Given a family of parallel lines in array A defined by

$$\alpha i + \beta j = c,$$

can we find a corresponding family of lines in array B given by

$$\alpha^{'} i^{'} + \beta^{'} j^{'} = c^{'} \tag{4}$$

such that there is no communication among processors?

The conditions on the solutions are: at most one of α and β can be zero; similarly, at most one of $\alpha^{'}$ and $\beta^{'}$ can be zero. Otherwise, the equations do not define parallel lines. A solution that satisfies these conditions is referred to as a *non-trivial solution* and the corresponding partition is called a *non-trivial partition.* Since $i^{'}$ and $j^{'}$ are given by equations 1 and 2, we have

$$\alpha^{'} (a_{11} i + a_{12} j + a_{10}) + \beta^{'} (a_{21} i + a_{22} j + a_{20}) = c^{'}$$

which implies

$$\left(\alpha^{'} a_{11} + \beta^{'} a_{21}\right) i + \left(\alpha^{'} a_{12} + \beta^{'} a_{22}\right) j = c^{'} - \alpha^{'} a_{10} - \beta^{'} a_{20}$$

Since a family of lines in A is defined by $\alpha i + \beta j = c$, we have

$$\alpha = \alpha' a_{11} + \beta' a_{21} \tag{5}$$

$$\beta = \alpha' a_{12} + \beta' a_{22} \tag{6}$$

$$c = c' - \alpha' a_{10} - \beta' a_{20} \tag{7}$$

A solution to the above system of equations would imply zero communication. In matrix notation, we have

$$\begin{pmatrix} a_{11} & a_{21} & 0 \\ a_{12} & a_{22} & 0 \\ -a_{10} & -a_{20} & 1 \end{pmatrix} \begin{pmatrix} \alpha' \\ \beta' \\ c' \end{pmatrix} = \begin{pmatrix} \alpha \\ \beta \\ c \end{pmatrix}$$

The above set of equations decouples into:

$$\begin{pmatrix} a_{11} & a_{21} \\ a_{12} & a_{22} \end{pmatrix} \begin{pmatrix} \alpha' \\ \beta' \end{pmatrix} = \begin{pmatrix} \alpha \\ \beta \end{pmatrix}$$

and

$$-a_{10}\alpha' - a_{20}\beta' + c' = c.$$

We illustrate the use of the above sufficient condition with the following example.

Example 6:
for $i = 2$ to N
  for $j = 2$ to N
    $A[i,j] \leftarrow B[i-1,j] + B[i,j-1]$

---

for each element $A[i,j]$, we need two elements of B. Consider the element $B[i-1,j]$. For communication-free partitioning, the system

$$\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 1 & 0 & 1 \end{pmatrix} \begin{pmatrix} \alpha' \\ \beta' \\ c' \end{pmatrix} = \begin{pmatrix} \alpha \\ \beta \\ c \end{pmatrix}$$

must have a solution. Similarly, considering the element $B[i,j-1]$, a solution must exist for the following system as well:

$$\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 1 & 1 \end{pmatrix} \begin{pmatrix} \alpha' \\ \beta' \\ c' \end{pmatrix} = \begin{pmatrix} \alpha \\ \beta \\ c \end{pmatrix}$$

Given that there is a single allocation for A and B, the two systems of equations must admit a solution. This reduces to the following system:

$$\begin{aligned} \alpha &= \alpha' \\ \beta &= \beta' \\ c &= c' + \alpha' \\ c &= c' + \beta' \end{aligned}$$

Figure 4: Partitions of arrays A and B for Example 6

The set of equations reduce to $\alpha = \alpha' = \beta = \beta'$ which has a solution say $\alpha = 1$. This implies that both A and B are partitioned by anti-diagonals. Figure 4 shows the partitions of the arrays for zero communication. The relations between c and c' give the corresponding lines in A and B.

With a minor modification of example 6 (as shown below),

Example 7:
for $i = 2$ to N
  for $j = 2$ to N
    $A[i,j] \leftarrow B[i-2,j] + B[i,j-1]$

---

the reduced system of equations would be:

$$\begin{aligned} \alpha &= \alpha' \\ \beta &= \beta' \\ c &= c' + 2\alpha' \\ c &= c' + \beta' \end{aligned}$$

which has a solution $\alpha = \alpha' = 1$ and $\beta = \beta' = 2$. Figure 5 shows the lines in arrays A and B which incur zero communication.

The next example shows a nested loop in which arrays can not be partitioned such that there is no communication.

Example 8:
for $i = 2$ to N

Figure 5: Lines in arrays A and B for Example 7

for $j = 2$ to N
  $A[i,j] \leftarrow B[i-2,j-1] + B[i-1,j-1] + B[i-1,j-2]$

The system of equations in this case is:

$$\begin{aligned} \alpha &= \alpha' \\ \beta &= \beta' \\ c &= c' + \alpha' + \beta' \\ c &= c' + 2\alpha' + \beta' \\ c &= c' + \alpha' + 2\beta' \end{aligned}$$

which reduces to

$$\begin{aligned} \alpha' + \beta' &= 2\alpha' + \beta' \\ \alpha' + \beta' &= \alpha' + 2\beta' \end{aligned}$$

which has only one solution $\alpha' = \beta' = 0$ which is not a *non-trivial solution.* Thus there is no communication-free partitioning of arrays A and B.

The examples considered so far involve constant offsets for access of array elements and we had to find *compatible* partitions. The next case considered is one where we need to find *mutually* compatible partitions. Consider the nested loop:

Example 9:

for $i = 2$ to N
L_1 : for $j = 1$ to N
  $A[i, j] \leftarrow B[i-1, j]$
 ...
 ...
L_2 : for $j = 2$ to N
  $B[i, j] \leftarrow A[i, j-1]$

In this case, the accesses due to loop L_1 result in the system:

$$\begin{aligned} \alpha &= \alpha^{'} \\ \beta &= \beta^{'} \\ c &= c^{'} + \alpha^{'} \end{aligned}$$

and the accesses due to loop L_2 result in the system:

$$\begin{aligned} \alpha^{'} &= \alpha \\ \beta^{'} &= \beta \\ c^{'} &= c + \beta \end{aligned}$$

Therefore, for communication-free partitioning, the two systems of equations written above must admit a solution – thus, we get the reduced system:

$$\begin{aligned} \alpha^{'} &= \alpha \\ \beta^{'} &= \beta \\ \alpha^{'} &= -\beta \end{aligned}$$

which has a solution $\alpha = \beta^{'} = 1$ and $\alpha^{'} = \beta = -1$. Figure 6 shows partitions of A and B into diagonals.

4.1 Constant offsets in reference

We discuss the important special case of array accesses with constant offsets which occur in codes for the solution of partial differential equations. Consider the following loop:

---

for $i = 1$ to N
 for $j = 1$ to N
  $A[i, j] \quad \leftarrow \quad B[i + q_1^i, j + q_1^j] \quad + \quad B[i + q_2^i, j + q_2^j] \quad + \quad \cdots \quad + \quad B[i + q_m^i, j + q_m^j]$

---

where q_k^i and q_k^j (for $1 \leq k \leq m$) are integer constants. The vectors $\vec{q_k} = (q_k^i, q_k^j)$ are referred to as *offset vectors.* There are m such *offset vectors,* one for each access pair $A[i, j]$ and $B[i + q_k^i, j + q_k^j]$.

Figure 6: Mutually compatible partition of A and B for Example 9

In such cases, the system of equations is (for each access indexed by k, where $1 \leq k \leq m$):

$$\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ -q_k^i & -q_k^j & 1 \end{pmatrix} \begin{pmatrix} \alpha' \\ \beta' \\ c' \end{pmatrix} = \begin{pmatrix} \alpha \\ \beta \\ c \end{pmatrix}$$

which reduces to the following constraints:

$$\begin{aligned} \alpha &= \alpha' \\ \beta &= \beta' \\ c &= c' - q_k^i \alpha' - q_k^j \beta' \qquad 1 \leq k \leq m \end{aligned}$$

Therefore, for a given collection of offset vectors, communication-free partitioning is possible, if

$$c = c' - q_k^i \alpha' - q_k^j \beta' \qquad 1 \leq k \leq m$$

If we consider the offset vectors (q_k^i, q_k^j) as points in 2-dimensional space, then communication free partitioning is possible, if the points (q_k^i, q_k^j) for $1 \leq k \leq m$ are collinear. In addition, if all $q_k^i = 0$, then no communication is required between rowwise partitions; similarly, if all $q_k^j = 0$, then partitioning the arrays into columns results in zero communication. For zero communication in nested loops involving K-dimensional arrays, this means that offset vectors treated as points in K-dimensional space must lie on a $K-1$ dimensional hyperplane.

In all the above examples, there was one solution to the set of values for $\alpha, \alpha', \beta, \beta'$. In the next section, we show an example where there are an infinite number of solutions, and loop transformations play a role in the choice of a specific solution.

5 Partitioning for linear references and program transformations

In this section, we discuss communication-free partitioning of arrays when references are not constant offsets but linear functions. Consider the loop in example 10:

Example 10:
for $i = 1$ to N
  for $j = 1$ to N
    $A[i,j] \leftarrow B[j,i]$

Communication free partitioning is possible if the system of equations

$$\begin{pmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} \alpha' \\ \beta' \\ c' \end{pmatrix} = \begin{pmatrix} \alpha \\ \beta \\ c \end{pmatrix}$$

has a solution where no more than one of α and β is zero and no more than one of α' and β' is zero. The set reduces to:

$$\begin{aligned} \alpha &= \beta' \\ \beta &= \alpha' \\ c &= c' \end{aligned}$$

This set has an infinite number of solutions. We present four of them and discuss their relative merits.

The first two equations involve four variables, fixing two of which leads to values of the other two. For example, if we set $\alpha = 1$ and $\beta = 0$, then array A is partitioned into rows. From the set of equations, we get $\alpha' = \beta = 0$ and $\beta' = \alpha = 1$, which means array B is being partitioned into columns; since $c = c'$, if we assign row k of array A and column k of array to the same processor, then there is no communication. See Figure 1 for the partitions.

A second partition can be chosen by setting $\alpha = 0$ and $\beta = 1$. In this case, array A is partitioned into columns. Therefore, $\alpha' = \beta = 1$ and $\beta' = \alpha = 0$ which means B is partitioned into rows. See Figure 7(a) for this partition. If we set $\alpha = 1$ and $\beta = 1$, then array A is partitioned into anti-diagonals. From the set of equations, we get $\alpha' = \beta = 1$ and $\beta' = \alpha = 1$, which means array B is also being partitioned into anti-diagonals. Figure 7(b) shows the third partition. A fourth partition can be chosen by setting $\alpha = 1$ and $\beta = -1$. In this case, array A is partitioned into diagonals. Therefore, $\alpha' = \beta = -1$ and $\beta' = \alpha = 1$ which means B is also partitioned into anti-diagonals. In this case, the kth sub-diagonal (below the diagonal) in A corresponds to the kth super-diagonal (above the diagonal) in array B. Figure 7(c) illustrates this partition.

Figure 7: Decompositions of arrays A and B for Example 10

From the point of loop transformations [2, 24], we can rewrite the loop to indicate which processor executes which portion of the loop iterations, partitions 1 and 2 are easy. Let us assume that the number of processors is p and the number of rows and columns in N and N is a multiple of p. In such a case, partition 1 (A is partitioned into rows) can be rewritten as:

`Processor` k `executes` $(1 \leq k \leq p)$ `:`
for $i = k$ to N by p
  for $j = 1$ to N
    $A[i,j] \leftarrow B[j,i]$

---

and all rows r of A such that r mod $p = k$ $(1 \leq r \leq N)$ are assigned to processor k; all columns r of B such that r mod $p = k$ $(1 \leq r \leq N)$ are also assigned to processor k.
In the case of partition 2 (A is partitioned into columns), the loop can be rewritten as:

`Processor` k `executes` $(1 \leq k \leq p)$ `:`
for $i = 1$ to N
  for $j = k$ to N by p
    $A[i,j] \leftarrow B[j,i]$

---

and all columns r of A such that r mod $p = k$ $(1 \leq r \leq N)$ and all rows r of B such that r mod $p = k$ $(1 \leq r \leq N)$ are also assigned to processor k. Since there are no data dependences anyway, the loops can be interchanged and written as:

`Processor` k `executes` $(1 \leq k \leq p)$ `:`
for $j = k$ to N by p
  for $i = 1$ to N
    $A[i,j] \leftarrow B[j,i]$

Partitions 3 and 4 can result in complicated loop structures. In partition 3, $\alpha = 1$ and $\beta = 1$. The steps we perform to transform the loop use *loop skewing* and *loop interchange* transformations [24]. We perform the following steps:

1. If $\alpha = 1$ and $\beta = 0$, then distribute the iterations of the outer loop in a round-robin manner; in this case, processor k (in a system of p processors) executes all iterations (i, j) where $i = k, k+p, k+2p, \ldots, k+N-p$ and $j = 1, \ldots, N$. This is referred to as *wrap distribution.* If $\alpha = 0$ and $\beta = 1$, then apply loop interchange and wrap-distribute the iterations of the interchanged outer loop. If not, apply the following transformation to the index set:

$$\begin{pmatrix} 1 & 0 \\ \alpha & \beta \end{pmatrix}$$

2. Apply loop interchanging so that the outer loop now can be stripmined. Since these loops do not involve flow or anti-dependences, loop interchanging is always legal. After the first transformation, the loop need not be rectangular. Therefore, the rules for interchange of trapezoidal loops in [24] are used for performing the loop interchange.

The resulting loop after transformation and loop interchange is the following:

Example 11:
for $j = 2$ to $2N$
  for $i = max(1, j - N)$ to $min(N, j - 1)$
    $A[i, j - i] \leftarrow B[j - i, i]$

The load-balanced version of the loop is:

`Processor` k `executes` $(1 \leq k \leq p)$ `:`
for $j = k + 1$ to $2N$ by p
  for $i = max(1, j - N)$ to $min(N, j - 1)$
    $A[i, j - i] \leftarrow B[j - i, i]$

The reason we distribute the outer loop iterations in a wrap-around manner is that such a distribution will result in load balanced execution when $N >> p$.
In partition 4, $\alpha = 1$ and $\beta = -1$. The resulting loop after transformation and loop interchange is the following:

`Processor` k `executes` $(1 \leq k \leq p)$ `:`
for $j = k + 1 - N$ to $N - 1$ by p
  for $i = max(1, 1 - j)$ to $min(N, N - j)$
    $A[i, j + i] \leftarrow B[j + i, i]$

Next, we consider a more complicated example to illustrate partitioning of linear recurrences:

Example 12:
for $i = 2$ to N
  for $j = 1$ to N
    $A[i, j] \leftarrow B[i + j, i] + B[i - 1, j]$

The access $B[i + j, i]$ results in the following system:

$$\begin{pmatrix} 1 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} \alpha' \\ \beta' \\ c' \end{pmatrix} = \begin{pmatrix} \alpha \\ \beta \\ c \end{pmatrix}$$

and the second access $B[i-1,j]$ results in the system:

$$\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 1 & 0 & 1 \end{pmatrix} \begin{pmatrix} \alpha' \\ \beta' \\ c' \end{pmatrix} = \begin{pmatrix} \alpha \\ \beta \\ c \end{pmatrix}$$

which together give rise to the following set of equations:

$$\begin{aligned} \alpha &= \alpha' + \beta' \\ \beta &= \alpha' \\ \alpha &= \alpha' \\ \beta &= \beta' \\ c &= c' + \alpha' \end{aligned}$$

which has only one solution which is $\alpha = \alpha' = \beta = \beta' = 0$. Thus communication-free partitioning has been shown to be impossible.

But for the following loop, communication-free partitioning into columns is possible.

Example 13:
for $i = 2$ to N
  for $j = 1$ to N
    $A[i,j] \leftarrow B[i+j,j] + B[i-1,j]$

The accesses give the following set of equations:

$$\begin{aligned} \alpha &= \alpha' + \beta' \\ \alpha &= \alpha' \\ \beta &= \beta' \\ c &= c' + \alpha' \end{aligned}$$

In this case, we have a solution: $\alpha = 0$ and $\beta = 1$ giving $\alpha' = 0$ and $\beta' = 1$. Thus both A and B are partitioned into columns.

6 Generalized linear references

In this section, we discuss the generalization of the problem formulation discussed in section 4.

Example 14:
for $i = 1$ to N
  for $j = 1$ to N
    $B[i'', j''] \leftarrow A[i', j']$

where $i^{'}, i^{''}, j^{'}$ and $j^{''}$ are linear functions of i and j, *i.e.*

$$i^{''} = f_l(i,j) = b_{11}i + b_{12}j + b_{10} \quad (8)$$

$$j^{''} = g_l(i,j) = b_{21}i + b_{22}j + b_{20} \quad (9)$$

$$i^{'} = f(i,j) = a_{11}i + a_{12}j + a_{10} \quad (10)$$

$$j^{'} = g(i,j) = a_{21}i + a_{22}j + a_{20} \quad (11)$$

Thus the statement in the loop is:

$$B[b_{11}i + b_{12}j + b_{10}, b_{21}i + b_{22}j + b_{20}] \leftarrow A[a_{11}i + a_{12}j + a_{10}, a_{21}i + a_{22}j + a_{20}]$$

In this case, the family of lines in array B are given by

$$\alpha i^{''} + \beta j^{''} = c$$

and lines in array A are given by:

$$\alpha^{'} i^{'} + \beta^{'} j^{'} = c^{'}.$$

Thus the families of lines are:

$$\texttt{Array B :} \quad \alpha\,(b_{11}i + b_{12}j + b_{10}) + \beta\,(b_{21}i + b_{22}j + b_{20}) = c \quad (12)$$

$$\texttt{Array A :} \quad \alpha^{'}\,(a_{11}i + a_{12}j + a_{10}) + \beta^{'}\,(a_{21}i + a_{22}j + a_{20}) = c^{'} \quad (13)$$

which is rewritten as:

$$\texttt{Array B :} \quad i\,(b_{11}\alpha + b_{21}\beta) + j\,(b_{12}\alpha + b_{22}\beta) = c - \alpha b_{10} - \beta b_{20} \quad (14)$$

$$\texttt{Array A :} \quad i\left(a_{11}\alpha^{'} + a_{21}\beta^{'}\right) + j\left(a_{12}\alpha^{'} + a_{22}\beta^{'}\right) = c^{'} - \alpha^{'} a_{10} - \beta^{'} a_{20} \quad (15)$$

Therefore, for communication-free partitioning, we should find a solution for following system of equations (with the constraint that at most one of α, β is zero and at most one of $\alpha^{'}, \beta^{'}$ is zero):

$$\begin{pmatrix} a_{11} & a_{21} & 0 \\ a_{12} & a_{22} & 0 \\ -a_{10} & -a_{20} & 1 \end{pmatrix} \begin{pmatrix} \alpha^{'} \\ \beta^{'} \\ c^{'} \end{pmatrix} = \begin{pmatrix} b_{11} & b_{21} & 0 \\ b_{12} & b_{22} & 0 \\ -b_{10} & -b_{20} & 1 \end{pmatrix} \begin{pmatrix} \alpha \\ \beta \\ c \end{pmatrix}$$

Consider the following example:

Example 15:
for $i = 2$ to N
  for $j = 1$ to N
    $B[i+j, i] \leftarrow A[i-1, j]$

Figure 8: Partitions of arrays A and B for Example 15

The accesses result in the following system of equations:

$$\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 1 & 0 & 1 \end{pmatrix} \begin{pmatrix} \alpha' \\ \beta' \\ c' \end{pmatrix} = \begin{pmatrix} 1 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} \alpha \\ \beta \\ c \end{pmatrix}$$

which leads to the following set of equations:

$$\begin{aligned} \alpha' &= \alpha + \beta \\ \beta' &= \alpha \\ c &= c' + \alpha' \end{aligned}$$

which has a solution $\alpha = 1, \beta = -1, \alpha' = 0, \beta' = 1$. See Figure 8 for the partitions.

Now for a more complicated example:

Example 16:

for $i = 2$ to N

  for $j = 2$ to N

    $B[i+j-3, i+2] \leftarrow A[i-1, j+1]$

The accesses result in the following system of equations:

$$\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 1 & -1 & 1 \end{pmatrix} \begin{pmatrix} \alpha' \\ \beta' \\ c' \end{pmatrix} = \begin{pmatrix} 1 & 1 & 0 \\ 1 & 0 & 0 \\ 3 & -2 & 1 \end{pmatrix} \begin{pmatrix} \alpha \\ \beta \\ c \end{pmatrix}$$

which leads to the following set of equations:

$$\begin{aligned} \alpha' &= \alpha + \beta \\ \beta' &= \alpha \\ \alpha' - \beta' + c' &= 3\alpha - 2\beta + c \end{aligned}$$

which has solutions where:

$$2 \begin{pmatrix} \alpha' \\ \beta' \end{pmatrix} = \begin{pmatrix} 5 & -1 \\ -1 & 3 \end{pmatrix} \begin{pmatrix} \alpha \\ \beta \end{pmatrix}$$

The system has the following solution: $\alpha = 1, \beta = 1, \alpha' = 2, \beta' = 1$. The loop after transformation is:

`Processor` k `executes` $(1 \le k \le p)$:
for $j = k + 4$ to $2N$ by p
  for $i = max(2, j - N)$ to $min(N, j - 2)$
    $B[j-3, i+2] \leftarrow A[i-1, j-i+1]$

The following section deals with a formulation of the problem for communication minimization, when communication-free partitions are not feasible.

7 Minimizing communication for constant offsets

In this section, we present a formulation of the communication minimization problem, that can be used when communication-free partitioning is impossible. We focus on two-dimensional arrays with constant offsets in accesses. The results can be generalized to higher dimensional arrays.

We consider the following loop model:

---

for $i = 1$ to N
  for $j = 1$ to N
    $A[i,j] \leftarrow B[i + q_1^i, j + q_1^j] + B[i + q_2^i, j + q_2^j] + \cdots + B[i + q_m^i, j + q_m^j]$

The array accesses in the above loop give rise to the set of offset vectors, $\vec{q_1}, \vec{q_2}, \ldots, \vec{q_m}$. The $2 \times m$ matrix Q whose columns are the offset vectors q_i is referred to as the *offset matrix.* Since $A[i,j]$

is computed in iteration (i,j), a partition of the array A defines a partition of the iteration space and *vice-versa*. For constant offsets, the shape of the partitions for the two arrays A and B will be the same; the array boundaries depend on the offset vectors.

Given the offset vectors, the problem is to derive partitions such that the processors have equal amount of work and communication is minimized. We assume that there are N^2 iterations (N^2 elements of array A are being computed) and the number of processors is p. We also assume that N^2 is a multiple of p. Thus, the workload for each processors is $\frac{N^2}{p}$.

The shapes of partitions considered are parallelograms, of which rectangles are a special case. A parallelogram is defined by two vectors each of which is normal to one side of the parallelogram. Let the normal vectors be $\vec{S_1} = (S_{11}, S_{12})$ and $\vec{S_2} = (S_{21}, S_{22})$. The matrix S refers to:

$$S = \begin{pmatrix} S_{11} & S_{12} \\ S_{21} & S_{22} \end{pmatrix}$$

If i and j are the array indices, $\vec{S_1}$ defines a family of lines given by $S_{11}i + S_{12}j = c_1$ for different values of c_1 and the vector $\vec{S_2}$ defines a family of lines given by $S_{21}i + S_{22}j = c_2$ for different values of c_2. S must be non-singular in order to define parallelogram blocks that span the entire array. The matrix S defines a linear transformation applied to each point (i,j); the image of the point (i,j) is $(S_{11}i + S_{12}j, S_{21}i + S_{22}j)$. We consider parallelograms defined by solutions to the following set of linear inequalities:

$$\begin{aligned} S_{11}i + S_{12}j &\geq c_1 \\ S_{11}i + S_{12}j &< c_1 + r_1 l \\ S_{21}i + S_{22}j &\geq c_2 \\ S_{21}i + S_{22}j &< c_2 + r_2 l \end{aligned}$$

where $r_1 l$ and $r_2 l$ are the lengths of the sides of the parallelograms.

The number of points in the discrete Cartesian space enclosed by this region (which must be the same as the workload for each processor, $\frac{N^2}{p}$) is $l^2 \frac{r_1 r_2}{|det(S)|}$ when $det(S) \neq 0$. The non-zero entries in the matrix $Q' = SQ$ represent inter-processor communication. Let $Q'(i)$ be the sum of the absolute values of the entries in the ith row of Q', *i.e.*

$$Q'(i) = \sum_{j=1}^{m} \left| Q'_{i,j} \right|$$

The communication volume incurred is:

$$2l \left(Q'(1) \frac{r_1}{|det(S)|} + Q'(2) \frac{r_2}{|det(S)|} \right) \tag{16}$$

Thus the problem of finding blocks which minimize inter-processor communication is that of finding the matrix S, the value l and the aspect ratios r_1 and r_2 such that the communication volume is minimized subject to the constraint that the processors have about the same amount of workload *i.e.*

$$l^2 \frac{r_1 r_2}{|det(S)|} = \frac{N^2}{p}$$

The elements of matrix S determine the shape of the partitions and the values of r_1, r_2, l determine the size of the partitions.

8 Summary

In current day distributed memory machines, interprocessor communication is more time-consuming than instruction execution. If insufficient attention is paid to the data allocation problem, then so much time may be spent in interprocessor communication that much of the benefit of parallelism is lost. It is therefore worthwhile for a compiler to analyze patterns of data usage to determine allocation, in order to minimize interprocessor communication. In this paper, we formulated the problem of determining if communication-free array partitions (decompositions) exist and presented machine-independent sufficient conditions for the same for a class of parallel loops without flow or anti dependences, where array references are affine functions of loop index variables. In addition, where communication-free decomposition is not possible, we presented a mathematical formulation that aids in minimizing communication.

Acknowlegdment

We gratefully acknowledge the helpful comments of the referees in improving the earlier draft of this paper.

References

[1] W. Abu-Sufah, D. Kuck and D. Lawrie, "On the Performance Enhancement of Paging Systems through Program Analysis and Transformations," *IEEE Trans. Computers,* Vol. C-30, No. 5, pages 341–356, May 1981.

[2] J. R. Allen and K. Kennedy, "Automatic Translation of FORTRAN Programs to Vector Form," *ACM Trans. Programming Languages and Systems,* Vol. 9, No. 4, pages 491–542, October 1987.

[3] V. Balasundaram, G. Fox, K. Kennedy and U. Kremer, "An interactive environment for data partitioning and distribution," *Proc. 5th Distributed Memory Computing Conference (DMCC5),* Charleston, S. Carolina, pages 1160–1170, April 1990.

[4] D. Callahan and K. Kennedy, "Compiling Programs for Distributed-Memory Multiprocessors," *The Journal of Supercomputing,* Vol. 2, pages 151–169, October 1988.

[5] M. Chen, Y. Choo and J. Li, "Compiling Parallel Programs by Optimizing Performance," *The Journal of Supercomputing,* Vol. 2, pages 171–207, October 1988.

[6] G. Fox, M. Johnson, G. Lyzenga, S. Otto, J. Salmon and D. Walker, *Solving Problems on Concurrent Processors – Volume 1: General Techniques and Regular Problems,* Prentice-Hall, Englewood Cliffs, New Jersey, 1988.

[7] K. Gallivan, W. Jalby and D. Gannon, "On the Problem of Optimizing Data Transfers for Complex Memory Systems," *Proc. 1988 ACM International Conference on Supercomputing,* St. Malo, France, pages 238–253, June 1988.

[8] D. Gannon, W. Jalby and K. Gallivan, "Strategies for Cache and Local Memory Management by Global Program Transformations," *Journal of Parallel and Distributed Computing,* Vol. 5, No. 5, pages 587–616, October 1988.

[9] M. Gerndt, "Array Distribution in SUPERB," *Proc. 1989 ACM International Conference on Supercomputing,* Athens, Greece, pages 164–174, June 1989.

[10] M. Gupta and P. Banerjee, "Automatic Data Partitioning on Distributed Memory Multiprocessors," Tech. Report CRHC-90-14, Center for Reliable and High-Performance Computing, University of Illinois, October 1990.

[11] D. Hudak and S. Abraham, "Compiler Techniques for Data Partitioning of Sequentially Iterated Parallel Loops," *Proc. ACM International Conference on Supercomputing,* pages 187–200, June 1990.

[12] A. Karp, "Programming for Parallelism," *IEEE Computer,* Volume 20, No. 5, pages 43–57, May 1987.

[13] K. Knobe, J. Lukas and G. Steele, "Data Optimization: Allocation of Arrays to Reduce Communication on SIMD Machines," *Journal of Parallel and Distributed Computing,* Vol. 8, No. 2, pages 102–118, February 1990.

[14] C. Koelbel, P. Mehrotra and J. van Rosendale, "Semi-automatic Process Partitioning for Parallel Computation," *International Journal of Parallel Programming,* Vol. 16, No. 5, pages 365–382, 1987.

[15] C. Koelbel, P. Mehrotra and J. van Rosendale, "Supporting Shared Data Structures on Distributed Memory Machines," *Proc. Principles and Practice of Parallel Programming,* Seattle, WA, pages 177–186, March 1990.

[16] C. Koelbel, *Compiling programs for non-shared memory machines,* Ph.D. thesis, CSD-TR-1037, Purdue University, November 1990.

[17] J. Li and M. Chen, "Index Domain Alignment: Minimizing Cost of Cross-Referencing between Distributed Arrays," *Technical Report YALEU/DCS/TR-275,* Department of Computer Science, Yale University, November 1989.

[18] M. Mace, *Memory Storage Patterns in Parallel Processing,* Kluwer Academic Publishers, Boston, Massachusetts, 1987.

[19] J. Ramanujam, *Compile-time Techniques for Parallel Execution of Loops on Distributed Memory Multiprocessors,* Ph.D. Thesis, Dept. of Computer and Information Science, The Ohio State University, Columbus, Ohio, September 1990.

[20] A. Rogers and K. Pingali, "Process Decomposition Through Locality of Reference," *Proc. ACM SIGPLAN 89 Conference on Programming Language Design and Implementation,* Portland, Oregon, pages 69–80, June 1989.

[21] A. Rogers, *Compiling for locality of reference,* Ph.D. thesis, Cornell University, August 1990.

[22] M. Rosing and R. Weaver, "Mapping Data to Processors in Distributed Memory Computations," *Proc. 5th Distributed Memory Computing Conference (DMCC5),* Charleston, S. Carolina, pages 884–893, April 1990.

[23] K. Wang and D. Gannon, "Applying AI Techniques to Program Optimization for Parallel Computers," in *Parallel Processing for Supercomputers and Artificial Intelligence,* K. Hwang and D. DeGroot (Eds), McGraw-Hill Publishing Company, New York, pages 441–485, 1989.

[24] M. Wolfe, *Optimizing Supercompilers for Supercomputers,* Pitman Publishing, London and the MIT Press, Cambridge, Massachusetts, 1989.

[25] H. Zima, H. Bast and H. Gerndt, "SUPERB: A Tool for Semi-automatic MIMD-SIMD Parallelization," *Parallel Computing,* Vol. 6, pages 1–18, 1988.

List of Figures

1 Partitions of arrays A and B for Example 1 5
2 Partitions of arrays A and B for Example 2 5
3 Array partitions for Example 4 7
4 Partitions of arrays A and B for Example 6 10
5 Lines in arrays A and B for Example 7 11
6 Mutually compatible partition of A and B for Example 9 13
7 Decompositions of arrays A and B for Example 10 15
8 Partitions of arrays A and B for Example 15 20