Access based data decomposition for distributed memory machines

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

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

Abstract

This paper addresses the problem of partitioning data for distributed memory machines or multicomputers. 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 matrix notation to describe array accesses in fully parallel loops which lets us derive sufficient conditions for communication-free decomposition of arrays.

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. 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 the compiler writer is severe. In addition to managing parallelism, it is now essential for the the compiler writer to appreciate the significance data distribution and when data should be copied or generated in local memory. This paper is an attempt at providing a new kind of compile-time analysis to determine communication-efficient decomposition of data structures. 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. A matrix notation to describe array accesses in fully parallel loops is presented which lets us present sufficient conditions for communication-free partitioning of arrays.

The rest of the paper is organized as follows. In section 2, we present a discussion of related work. In section 3, 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. Section 4 generalizes the formulation presented in section 3 for arbitrary linear references. Section 5 concludes with a summary and discussion.

2 Background

Operations involving two or more operands require that the operands be *aligned, i.e.,* the operands must be 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 the following sections, 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.

2.1 Related work

Balasundaram and others [1] at CalTech 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. Gupta and Banerjee [2] present a constraint-based system for automatic data distribution. Knobe *et al.* [4] discuss techniques for automatic layout of arrays in a compiler targeted to SIMD architectures such as the Connection Machine computer system. They present an approach based on array occurrences rather than array declarations. Li and Chen [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 deal with primarily are *permutations* and *embeddings.* The kind of alignment functions that we deal with are more general than these. Mace [6] 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.

2.2 Communication-free partitions

Consider the following loop skeleton:

Example 1:
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 partitioning of arrays involved in adjacent loops where different arrays are modified is referred to as *mutually compatible* partitions.

To check if communication-free partitioning is not possible, one can build a k-partite graph (the graph can be partitioned into k sets such that each edge in the graph has its vertices from different sets) where each set corresponds to an array. This is possible since we do not consider flow or anti dependences. To each array element that is accessed, associate a vertex in this graph. For every occurrence of that array element on the left hand-side of an assignment statement, connect the node corresponding to that element and all the elements of other arrays that are used on the right hand side. If this graph is connected, then communication-free partitioning is not possible [7].

3 Problem formulation

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

Example 2:
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 which do not have flow or anti-dependences. Because of the assumption that all updates to an element are made by the processor that "owns" the element, loops with output dependences are treated in the same manner as *DOALL* 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 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 two dimensional arrays. A hyperplane in two dimensions is a line and discuss techniques to find parallel lines in A and B which incur zero communication. In most loops that occur in scientific computation, the functions f and g are linear in i and j.

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

defines a family of parallel lines for different values of k, given that α and β are constants. 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?

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

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

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

$$\alpha = \left(\alpha^{'} a_{11} + \beta^{'} a_{21}\right) \tag{5}$$
$$\beta = \left(\alpha^{'} a_{12} + \beta^{'} a_{22}\right) \tag{6}$$
$$c = \left(c^{'} - \alpha^{'} a_{10} - \beta^{'} a_{20}\right) \tag{7}$$

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

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

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.* The above set of equations decouple into:

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

and

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

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

Example 3:
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

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

must have a solution. Similarly, considering the element $B[i, j-1]$, the system

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

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:

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

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.

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

Example 4:
for $i = 2$ to N
  for $j = 2$ to N
    A[i,j] ← 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 has only one solution $\alpha' = \beta' = 0$ which is not a *non-trivial solution.* Thus one can not find 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 5:
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:

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

The system has a solution $\alpha = \beta' = 1$ and $\alpha' = \beta = -1$ – thus communication-free partitioning is possible.

3.1 Constant offsets in reference

In the case of constant offsets in array access, i.e. where $i' = i + c_i$ and $j' = j + c_j$, we can denote the offset vectors as $\vec{q_k} = (q_k^i, q_k^j)$. Let there be m such offset vectors, one for each access pair. In such cases, the system of equations is (for each access indexed by k, where $1 \leq k \leq m$):

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

which reduces to the following constraints:

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

For a given set of offset vectors, communication-free partitioning is possible, if

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

If we consider the offset vectors (q_k^i, q_k^j) as points in two dimensional space, communication free partitioning is possible, if the points (q_k^i, q_k^j) for $1 \leq k \leq m$ are collinear. 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.

4 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 6:
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

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

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:

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

The first two equations involve four variables, fixing two of which leads to values of the other two. Thus, this set has an infinite number of non-trivial solutions. 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.

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. 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. 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.

From the point of loop transformations [8], 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 $r \bmod p = k$ $(1 \leq r \leq N)$ are assigned to processor k; all columns r of B such that $r \bmod p = k$ $(1 \leq r \leq N)$ are also assigned to processor k. The loop transformation for partition 2 is similarly derived.

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 [8]. We perform the following steps:

1. If $\alpha = 1$ and $\beta = 0$, then wrap-distribute the iterations of the outer loop resulting in the partitions of the array. 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:
$$\begin{bmatrix} 1 & 0 \\ \alpha & \beta \end{bmatrix}$$
2. Apply loop interchanging so that the outer loop now can be strip mined. 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 [8] are used for performing the loop interchange.

For partition 3, the resulting loop after transformation and loop interchange is the following:

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

When $N >> p$, we can derive a load-balanced version of the loops by distributing the iterations of the outer loop index among the processors in a wrap-around fashion; a processor that is assigned an outer loop index performs all the iterations of the inner loop for that index. 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 $min(N, j - 1)$
    $A[i, j - i] \leftarrow B[j - i, i]$

---

5 Summary

In current day distributed memory machines, interprocessor communication is expensive. 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.

References

[1] V. Balasundaram, G. Fox, K. Kennedy and U. Kremer, "An interactive environment for data partitioning and distribution," *Proc. 5th Distributed Memory Computing Conference,* pp. 1160–1170, April 1990.

[2] 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.

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

[4] 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, pp. 102–118, February 1990.

[5] 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.

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

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

[8] M. Wolfe, *Optimizing Supercompilers for Supercomputers,* MIT Press, Cambridge, Massachusetts, 1989.