AUTOMATIC DATA AND COMPUTATION MAPPING FOR DISTRIBUTED MEMORY MACHINES

A Dissertation

Submitted to the Graduate Faculty of the
Louisiana State University and
Agricultural and Mechanical College
in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy

in

The Department of Electrical and Computer Engineering

by
Isidoro Couvertier-Reyes
B.S., University of Puerto Rico-Mayagüez, 1981
M.S., University of Wiscosin-Madison, 1983
May 1996

TABLE OF CONTENTS

List of Tables

LIST OF FIGURES

Abstract

Distributed memory parallel computers offer enormous computation power, scalability and flexibility. However, these machines are difficult to program and this limits their widespread use. An important characteristic of these machines is the difference in the access time for data in local versus non-local memory; non-local memory accesses are much slower than local memory accesses. This is also a characteristic of shared memory machines but to a less degree.

Therefore it is essential that as far as possible, the data that needs to be accessed by a processor during the execution of the computation assigned to it reside in its local memory rather than in some other processor's memory. Several research projects have concluded that proper mapping of data is key to realizing the performance potential of distributed memory machines. Current language design efforts such as Fortran D and High Performance Fortran (HPF) are based on this.

It is our thesis that for many practical codes, it is possible to derive good mappings through a combination of algorithms and systematic procedures. We view mapping as consisting of wo phases, *alignment* followed by *distribution.* For the alignment phase we present three constraint-based methods – one based on a linear programming formulation of the problem; the second formulates the alignment problem as a constrained optimization problem using Lagrange multipliers; the third method uses a heuristic to decide which constraints to leave unsatisfied (based on the penalty of increased communication incurred in doing so) in order to find a mapping.

In addressing the distribution phase, we have developed two methods that integrate the placement of computation—loop nests in our case—with the mapping of data. For one distributed dimension, our approach finds the best combination of data and computation mapping that results in low communication overhead; this is done by choosing a loop order that allows message vectorization. In the second method, we introduce the distribution preference graph and the operations on this graph allow us to integrate loop restructuring transformations and data mapping.

These techniques produce mappings that have been used in efficient hand-coded implementations of several benchmark codes.

CHAPTER 1

INTRODUCTION

Distributed memory computers (DMCs) offer great promise to scientists because of their scalability and potential for enormous computational power. Yet, their widespread use is hindered by the difficulty of parallel programming. Scientific programmers have had to write explicitly parallel code, and face many efficiency issues in deriving satisfactory performance. When a parallel program is run in a DMC, data need to be distributed among processors and explicit communication need to be incorporated in order to provide for the exchange of data among processors inherent in many programs. The processors in a DMC communicate by exchanging messages whenever a processor needs data which is located in some other processor's memory. This exchanging of data through software is commonly referred to in the literature as message passing and DMCs are usually known as message passing computers. Deciding when to insert messages in a program, thus implementing data and computation partitioning, and which partitioning of data is optimal are no easy tasks, and much effort has gone into developing ways to relieve the programmer from this burden.

Data and computation partitioning are at the heart of the compilation process or transformation of a single processor sequential program into a Single Program Multiple Data (SPMD) program to be run on a distributed memory machine. The main goal of parallelization of code is increased performance measured by reasonable speed-ups. However, if code is not properly parallelized, the result could be a parallelized code which may be even slower than sequential code as reported by Blume and Eigenmann [18]. One of the main sources of this undesirable degradation in execution time is the communication among processors and the overhead incurred by this communication. This is covered in detail by Stone [79]. In most situations communication is unavoidable due to the characteristics of the code, however it can be reduced in many instances.

A key component of the compilation process is the mapping of data to processors. Scientific computing is the application domain we concentrate on. We concentrate on arrays due to the fact that it is the predominant data structure used in scientific code and on loops because it is in them where most of the execution time of scientific programs is spent. This is done by a two-step process consisting of *alignment* followed by *distribution*; see Figure1.1. In the alignment phase, array elements are mapped to a template, which is an abstract multi-dimensional grid; this allows one to relate members of different arrays, and specify replication if needed. For example, in the case of arrays, this allows to specify the relative locations of elements of different arrays. The alignment is typically a function of only the data access patterns in the program, but not of the target machine architecture. In practice, the size of the template is much too large compared to the number of processors. The distribution phase partitions the template, and hence the array elements aligned to the template; this phase is a function of both the program and the target machine architecture.

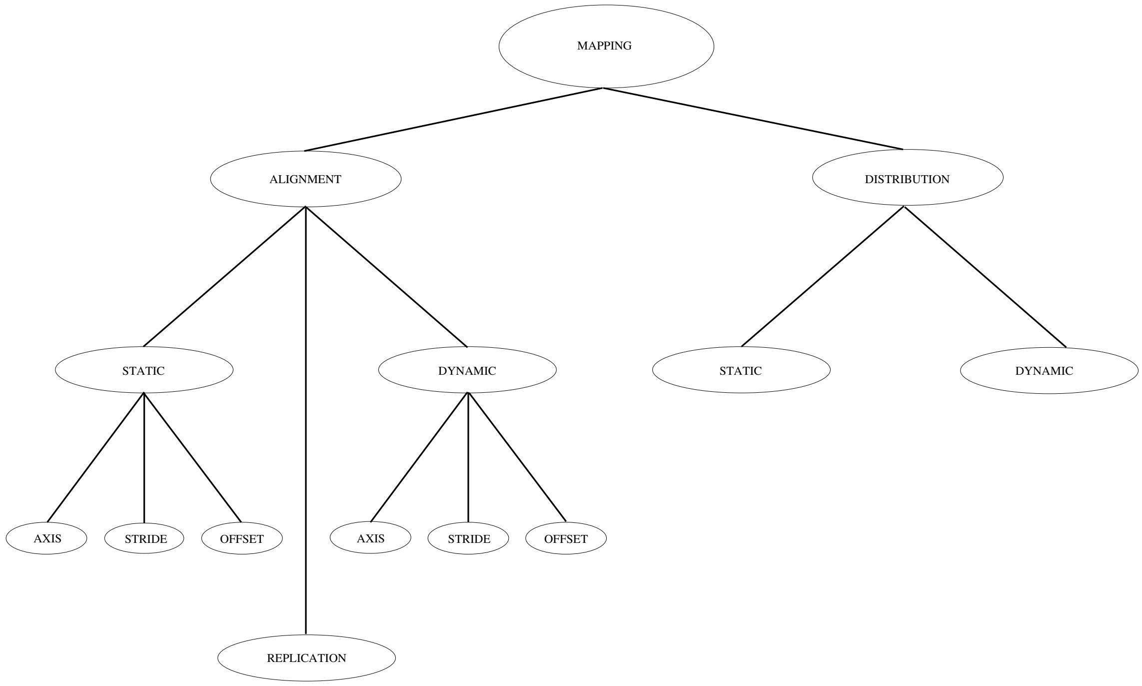


Figure 1.1: Structure of the Data Mapping Problem.

The focus of this research is the optimization of communication among processors in a DMC by properly aligning data and computation, by finding good distributions, and by applying transformations that will allow the use of message vectorization whenever possible. Our work automatically finds the alignment and distribution for the program thus relieving the programmer from this task. Our findings will then be inserted into the program using the alignment and distribution declarations which are later introduced in Section 1.1.1.

Throughout this dissertation we will be using several metrics to model the effect of moving array elements from one position to another. The distance between a point p and a point q in a k-dimensional space using the l_1 or Manhattan for is $d(p, q) = \sum_i |p_i - q_i|$, and using the l_2 or Euclidean metric it is $d(p, q) = \sqrt{\sum_i (p_i - p_i)^2}$, where $1 \leq i \leq k$. The l_1 metric is realistic for a one dimensional processor array whereas the l_2 metric is realistic for a grid of processors with nearest neighbor connections [31]. Other metrics which are also used in the literature are the l_∞ and the Hamming metrics which are realistic for a grid of processors with connections to their nearest neighbors and to their diagonal neighbors and for hypercubes, respectively [31].

Section 1.1 of this chapter presents the scope of our research. In Section 1.2 an overview of the most recent and important research, as it relates to ours, is given. The rest of this dissertation is organized as outlined below. Chapters 2, 3, and 4 relate to the alignment problem and they present solutions using a Linear Programming approach, a method using Lagrange Multipliers, and a method where the solution for over-constrained systems is obtained by relaxing some constraint(s), respectively. Chapter 5 presents a method to find distributions using matrices along with ways to reduce communication when this is unavoidable and Chapter 6 presents a method using a novel graph-based framework. Finally, Chapter 7 presents our conclusions, and summarizes our contributions and presents ideas for future research.

1.1 SCOPE OF OUR RESEARCH

As mentioned earlier, the focus of our current research is the optimization of the communication among processors by properly partitioning the data and computation. It includes not only the alignment problem, but also the distribution problem (as shown in Figure 1.1) and program transformations. This latter part is not included in the figure mentioned above. In this chapter we provide an overview of our research.

1.1.1 ALIGNMENT OVERVIEW

Alignment in data parallel programs, as illustrated in Figure 1.1, can take the form of static alignment, dynamic alignment, and replication of arrays. Static alignment refers to the alignment which is determined at compile time and dynamic alignment to the alignment determined at runtime. Both static and dynamic alignment can be further classified as *axis*, *stride and reversal*, and *offset* alignment. Static alignment is specified in the High Performance Fortran (HPF) standard using the `ALIGN` declaration, whereas dynamic alignment is specified through the executable statement `REALIGN` [41]. Similarly, static distribution is accomplished through the `DISTRIBUTE` declaration and refers to compile time distribution, and dynamic distribution via the `REDISTRIBUTE` executable statement at runtime.

It is clear by now why we need to align and distribute the arrays that are used in a data parallel program. But where does the need for realignment and redistribution comes from? The answer to this question is simple: It all comes from the change in data access patterns in programs. Some programs may access a particular array in one fashion during the execution of a loop nest and then access the same array in a different fashion. For example, we may have a loop inside which elements of an array X are computed as functions of the elements of an array Y such as $X[i] = f(Y[i])$; we may then have some computation performed on X and then another loop with an instruction $X[i] = g(Y[2i+5])$ as shown below. The notation above indicates that $X[i]$ is assigned a copy of some function f or g of some element of array Y.

DO $i = 1, N$
  $X[i] = Y[i]$
ENDDO
DO $i = 1, N$

$X[i] = Y[2i+5]$
ENDDO

For the first loop it is advantageous to align X and Y identically, but the second loop dictates a different alignment. In order to reduce communication, array Y needs to be realigned before the execution of the second loop.

A common case in scientific codes involving multidimensional arrays requires transposition of one of the arrays, e.g.

DO $i = 1, N$
$X[i,j] = Y[i,j]$
ENDDO
DO $i = 1, N$
$X[i,j] = Y[j,i]$
ENDDO

In this case array Y needs to be transposed between the loops. A redistribution may also arise, for example, because the programmer decided that it was better to distribute an array in a certain manner if the number of processors that were available was greater than or equal to some number and to distribute it in another manner if it was otherwise [80].

A program may also have a need for replication of arrays if doing so will result in a reduction of communication among processors or just simply because the programmer has specified it. For example, scalars and small read only arrays may be replicated onto the processors and, in so doing, completely eliminate the communication that could have arisen because a processor needed elements owned by some other processor. Also, we may have to replicate a one-dimensional array onto a multidimensional array for reasons similar to the ones previously stated. Consider the following code:

DO $i = 1, N$
DO $j = 1, M$
$X[i,j] = Y[i,j] * Z[i]$
ENDDO
ENDDO

In the above piece of code the one-dimensional read-only array Z could be replicated such that each processor owns a copy and thus can perform its computation without having to communicate, which would be the case if Z is not replicated properly. In HPF terminology, we could replicate Z along the rows or columns of a two-dimensional template to which both arrays X and Y are aligned with the result that each processor owning an element of X and Y will also own the entire array Z.

1.1.2 Distribution Overview

The distribution phase of the data mapping problem can be defined as the phase where the abstract template, and thus all the arrays aligned to it, are mapped onto the physical processors. This phase comes after the data structures have been aligned to the template. As with the alignment phase, the distribution phase can be subdivided into static distribution and dynamic distribution.

The most commonly used distributions, which are the only ones currently available in the High Performance Fortran (HPF) proposed standard [41], are the cyclic, cyclic(size), block, and block(size) distributions, where size is a parameter which specifies the number of data items from a template to be assigned to a processor. The cyclic distribution assigns one element to each processor in turn until all the processors assigned to that dimension of the template are exhausted, it then assigns a new element to each processor, and continues until all the elements on that dimension of the template are assigned. As explained by Gupta and Banerjee [33], this distribution is of special importance when load balancing needs to be achieved in the presence of iteration spaces where the lower or upper bound of an iteration variable is a function of an outer iteration variable, e.g. triangular iteration spaces. On the other hand, this type of distribution is not the best choice when there is a lot of nearest neighbor communication among processors, in which case a block distribution would be preferred [33]. See Figure 1.2 for examples of cyclic distributions and the code shown below for a triangular iteration space example. Note that the lower bound for loop j is an affine function of the outer loop index variable i.

DO $i = 1, N$
    DO $j = i, N$
        $\vdots$
    ENDDO
ENDDO

The cyclic(size) distribution provides the programmer with the ability of specifying the number of elements which the compiler should assign to each processor in a cyclic manner. Thus, cyclic(1) produces the same effect as cyclic.

The block distribution assigns a number of elements equal to the ceiling of the number of elements of the array in a particular dimension divided by the number of processors available for that dimension. Finally, the block(size) distribution assigns a programmer's specified number of elements to each processor. Examples are given in Figure 1.3. Note that both the block and the block(size) distributions can also be obtained from the cyclic(size) distribution. Block distributions are especially suited for rectangular iteration spaces and nearest neighbor (shift or offset) communication [33].

Skewed distributions are a more general class of distributions from which row, column, diagonal, parallelogram, etc. distributions could be derived. Both row and column distributions are one-dimensional distributions which can be obtained by skewing one dimension by a factor of zero with respect to another dimension. This factor has a non-zero value for diagonal distributions. These distributions are also referred to in the literature as hyperplanes. Skewed distributions, however general, are not currently supported by HPF [41].

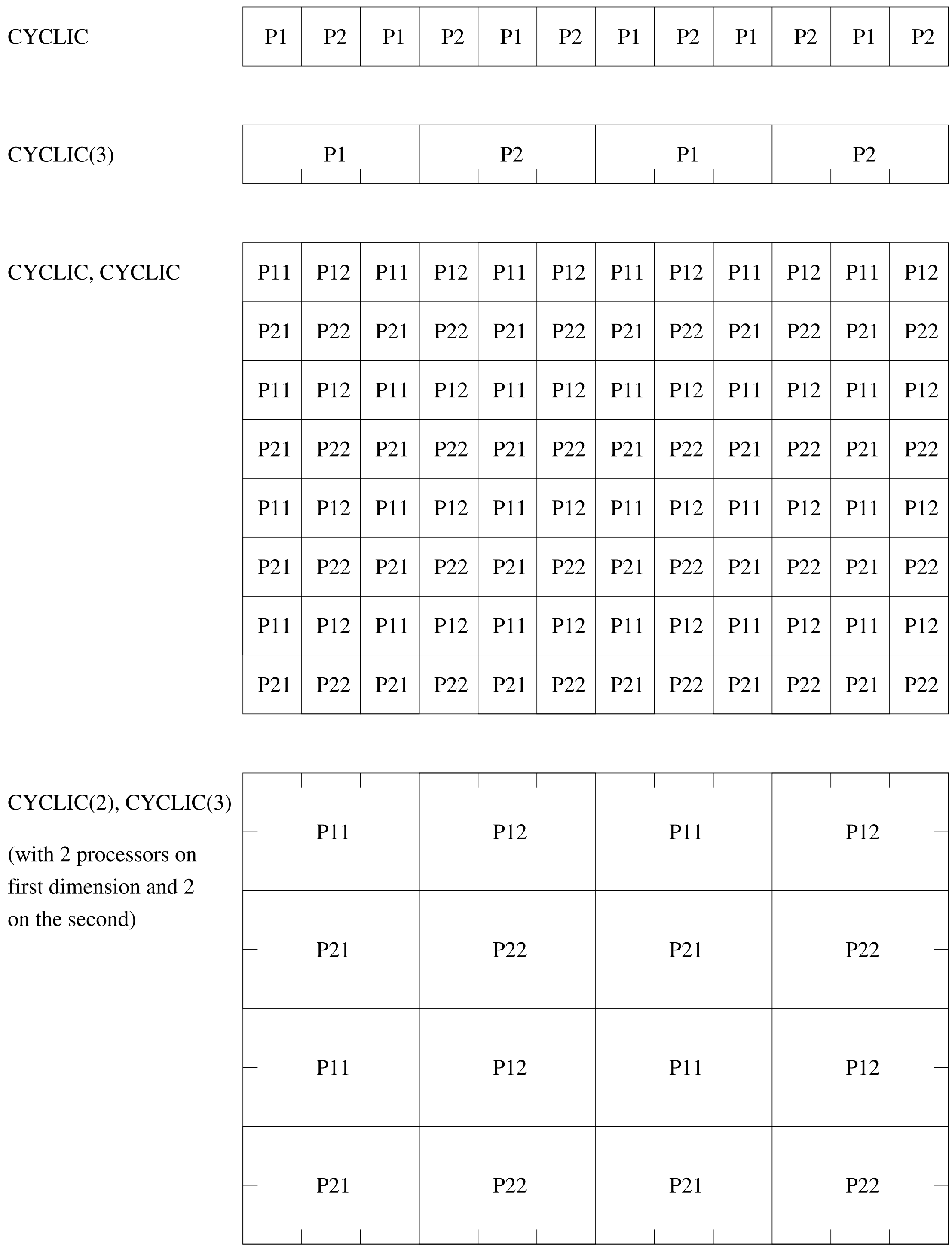


Figure 1.2: Cyclic and Cyclic(size) Distribution Examples.

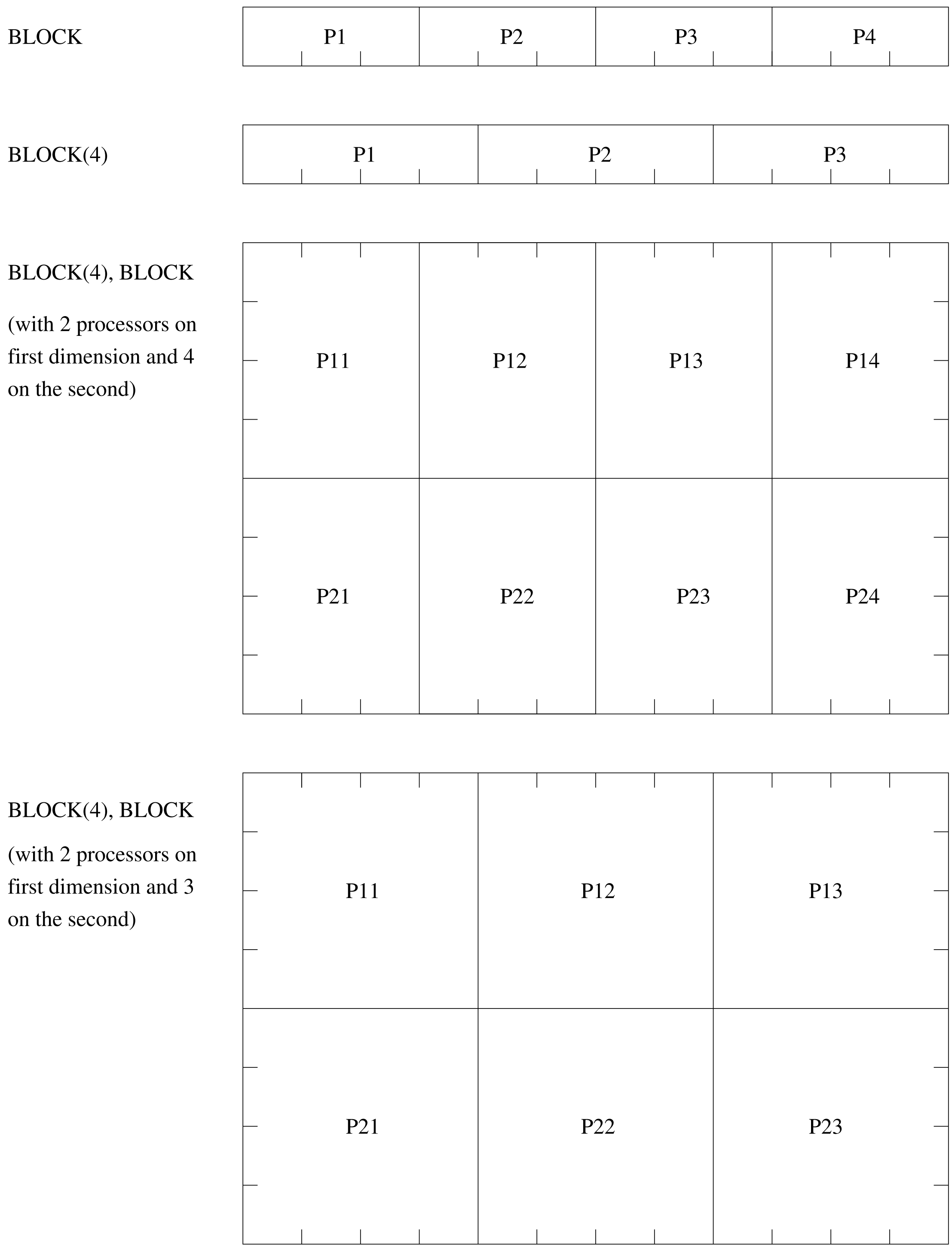


Figure 1.3: Block and Block(size) Distribution Examples.

1.2 BACKGROUND AND RELATED WORK

The component alignment problem has been proven to be NP-complete by Li and Chen [55]. Most of the effort in this research area is very recent. We review the work of several researchers on the problem and discuss how it relates to our work.

Ramanujam and Sadayappan [68], present a technique which applies to one fully parallel loop nest at a time. Only array data partitions defined by a family of parallel hyperplanes are considered. Communication-free data partitioning is the main subject in Ramanujam and Sadayappan [68]. However, a formulation is given for minimizing communication while balancing the workload, which is used as a constraint, among processors when communication-free partitioning is not possible. The work by Ramanujam and Sadayappan [68] is architecture-independent and assumes the owner-computes rule. The proposed alignment and the functions used are more general than those in the High Performance Fortran (HPF) standard [41], where at most one index variable per subscript expression is allowed. To illustrate the general idea behind the work by Ramanujam and Sadayappan [68] consider the following assignment statement where A, and B are two 2-dimensional arrays

$$B[f_l(i,j), g_l(i,j)] \leftarrow A[f_r(i,j), g_r(i,j)].$$

Assume that the assignment statement above is located inside a loop nest of depth 2 with i as the outermost loop index variable, j as the innermost one, and that the loop bounds are constant; also assume that f_l, f_r, g_l, and g_r are affine functions of i, and j. We can write the above functions as

$$\begin{aligned} i^{''} &= f_l(i,j) = b_{11}i + b_{12}j + b_{10} \\ j^{''} &= g_l(i,j) = b_{21}i + b_{22}j + b_{20} \\ i^{'} &= f_r(i,j) = a_{11}i + a_{12}j + a_{10} \\ j^{'} &= g_r(i,j) = a_{21}i + a_{22}j + a_{20} \end{aligned}$$

The subscript functions for array B define a family of lines given by $\alpha i^{''} + \beta j^{''} = c$ and those for array A define lines given by $\alpha^{'} i^{'} + \beta^{'} j^{'} = c^{'}$. From these equations we obtain the following condition for array B

$$i(b_{11}\alpha + b_{21}\beta) + j(b_{12}\alpha + b_{22}\beta) = c - b_{10}\alpha - b_{20}\beta,$$

and the following condition for array A

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

For communication-free partitioning, there must be a solution to the system of equations above, shown below in matrix form, where at most one of α and β is zero, and at most one of $\alpha^{'}$ and $\beta^{'}$ is zero.

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

Consider the following loop:

DO $i = lb_i, ub_i$
  DO $j = lb_j, ub_j$
    $A[i, j] = f(A[i, j], B[i-1, j], B[i, j-1])$
  ENDDO
ENDDO

where lb_i, ub_i, lb_j, and ub_j are the lower and upper loop bounds for loops i, and j. These lower and upper loop bounds are assumed to be constant values, i.e. they are known at compile time. Note that there are two distinct accesses to array B. From the $A[i, j]$ term on the *rhs* we have

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

from this system of equations we obtain $\alpha = \alpha$, $\beta = \beta$, $c = c$. Similarly, from $B[i-1, j]$ we obtain the following system of equations:

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

which yields $\alpha^{'} = \alpha$, $\beta^{'} = \beta$, $\alpha^{'} + c^{'} = c$. From this last equation we obtain $c^{'} = c - \alpha^{'}$. From $B[i, j-1]$ we obtain the system of equations shown below:

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

which yields the following equations $\alpha^{'} = \alpha$, $\beta^{'} = \beta$, $c^{'} = c - \beta$. For communication-free data partitioning, the system of equations shown above must have a solution, thus we find the solution $\alpha = \alpha^{'} = \beta^{'} = \beta = 1$, i.e. both arrays A and B should be partitioned into diagonals. Note that since $c^{'} = c - 1$ the corresponding partition for array B will be one line below that for array A. If the only solution had been $\alpha = \alpha^{'} = \beta = \beta^{'} = 0$, then no communication-free data partitioning would be possible other than mapping everything to just one processor, i.e. the trivial solution.

In our work we deal with alignment for both data and computation and we do not assume the owner-computes rule.

Huang and Sadayappan [45] focus on partitions of iterations and data arrays that eliminate data communication and considers partitions of iteration and data spaces along sets of hyperplanes. Since data elements are not to be accessed by different processors, even read-only data cannot be shared. All iterations belonging to an iteration hyperplane and all the data belonging to a data hyperplane are assigned to one processor, thus the owner-computes rule is implicit. A processor will execute iterations from the iteration hyperplanes which are assigned to it and in so doing it will access data from its assigned data hyperplanes.

The article presents no way of dealing with cases when communication-free partitioning, while maintaining parallelism, is not possible. It begins by presenting solutions for a single hyperplane partitioning for each iteration and data space and moves on to multiple (double) hyperplanes per space at which time they propose a heuristic. Huang and Sadayappan [45] derive necessary and sufficient conditions for communication-free hyperplane partitioning of both data and computation for fully parallel loop nests in the absence of flow and anti-dependences. Flow and anti-dependences are treated elsewhere and a list of articles which treat this subject is given later in this work. For communication-free single hyperplane partitioning of the iteration and data spaces the following must hold for an access function in the form of $A^i_{j,k}(I) + a^i_{j,k}$, which accesses the k-th reference to the j-th data array in the i-th nested loop, where I is used to denote the iteration vector, H and G are row vectors containing the iteration and data hyperplane coefficients (which are rational numbers), respectively, and α is nonzero.

1. $G_{j1}A^i_{j1,k_1} = G_{j2}A^i_{j2,k_2}$
2. $G_j a^i_{j,k_1} = G_j a^i_{j,k_2}$
3. $H_i = \alpha^i_j G_j A^i_{j,1}$
4. $\alpha^{i1}_{j1}\alpha^{i2}_{j2} = \alpha^{i1}_{j2}\alpha^{i2}_{j1}$
5. $\alpha^{i1}_{j1}G_{j1}(a^{i1}_{j1,1} - a^{i2}_{j1,1}) = \alpha^{i1}_{j2}G_{j2}(a^{i1}_{j2,1} - a^{i2}_{j2,1})$

An example which captures the essence of their work for the case of multiple arrays, multiple references, with single hyperplane partitioning is shown below for the same loop used previously, i.e.

DO $i = lb_i, ub_i$
  DO $j = lb_j, ub_j$
    $A[i, j] = f(A[i, j], B[i - 1, j], B[i, j - 1])$
  ENDDO
ENDDO

Here we find

$$A_1 = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}, a_1 = \begin{bmatrix} 0 \\ 0 \end{bmatrix}, A_2 = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}, a_2 = \begin{bmatrix} 0 \\ 0 \end{bmatrix},$$

$$B_1 = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}, b_1 = \begin{bmatrix} -1 \\ 0 \end{bmatrix}, B_2 = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}, b_2 = \begin{bmatrix} 0 \\ -1 \end{bmatrix}$$

from which, by applying the conditions stated previously, we get

$$G_A \begin{bmatrix} A_1 - A_2 & a_1 - a_2 \end{bmatrix} = \begin{bmatrix} g_{A1} & g_{A2} \end{bmatrix} \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} = \begin{bmatrix} 0 & 0 & 0 \end{bmatrix}$$

and

$$G_A \begin{bmatrix} B_1 - B_2 & b_1 - b_2 \end{bmatrix} = \begin{bmatrix} g_{B1} & g_{B2} \end{bmatrix} \begin{bmatrix} 0 & 0 & -1 \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 0 & 0 & 0 \end{bmatrix}.$$

From the last equation we find that $g_{B1} = g_{B2}$. The other set of equations is found from

$$\begin{bmatrix} G_A & G_B \end{bmatrix} \begin{bmatrix} A_1 \\ -B_1 \end{bmatrix} = \begin{bmatrix} g_{A1} & g_{A2} & g_{B1} & g_{B2} \end{bmatrix} \begin{bmatrix} 1 & 0 \\ 0 & 1 \\ -1 & 0 \\ 0 & -1 \end{bmatrix} = \begin{bmatrix} 0 & 0 \end{bmatrix}$$

which yields $g_{A1} = g_{B1}$, and $g_{A2} = g_{B2}$. Therefore, we can choose

$$G_A = G_B = \begin{bmatrix} 1 & 1 \end{bmatrix}$$

which is the same result we obtained using the method in Ramanujam and Sadayappan [68]. Additionally,

$$H = \alpha_A G_A A_1 = \begin{bmatrix} 1 & 1 \end{bmatrix}$$

for $\alpha_A = 1$.

The work in Huang and Sadayappan [45] does not assume anything about the architecture of the machine, and implicitly assumes the owner-computes rule. As mentioned earlier, no attempt is made to deal with the problem when communication is unavoidable. The alignment obtained and the access functions allowed are more general than what is allowed in the current HPF standard [41].

In our work we are also interested in obtaining communication-free data and computation alignment, but we do not assume the owner-computes rule; in addition, rather than stopping when zero communication is impossible, we use a method for reducing the overall communication when communication is unavoidable. For example, consider a loop where all the array elements on the *rhs* of the statement are in processors which are different from the processor which owns the *lhs* element. In the work by Huang and Sadayappan [45], because communication is unavoidable, the computation would have to be sequentialized even though it may be possible to parallelize the loop. However, our method finds the alignment that would minimize communication in such a way that the final computation is carried out at the processor at which communication is found to be minimum and then this processor sends the final result to the owner of the *lhs* element.

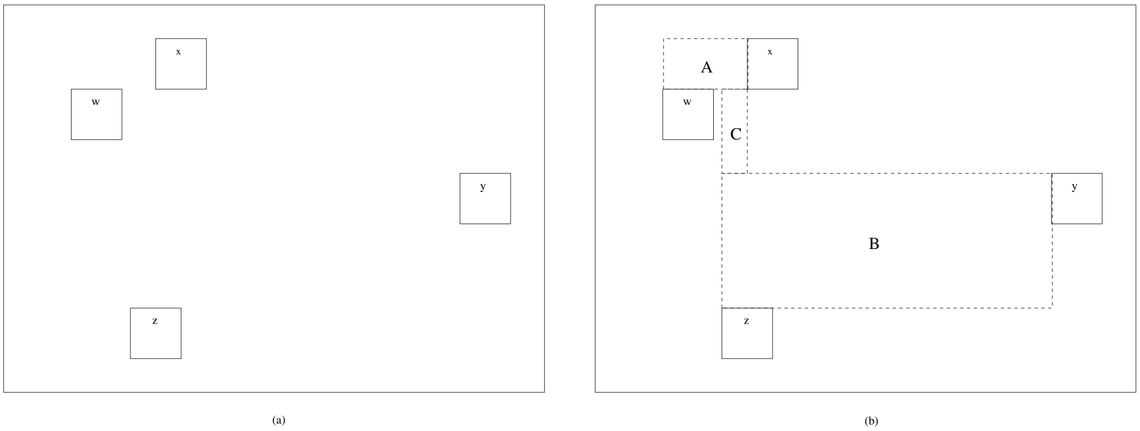


Figure 1.4: Example from Gilbert and Schreiber [31].

Gilbert and Schreiber [31], propose a method which considers only one expression at a time. The minimum cost of computing an arbitrary expression is found for architectures with *robustness*, e.g. hypercubes, linear arrays, meshes, etc. on which realistic metrics could be used. As Gilbert and Schreiber [31] explain, a given metric describes the cost of moving an array from one position to another within a machine. For example the l_1 (Manhattan), l_2 (Euclidean), l_∞, and Hamming metrics are realistic for a one dimensional processor array, a grid of processors with connections to their nearest neighbors, a grid of processors with connections to their nearest neighbors and their diagonal neighbors, and for hypercubes, respectively. In the l_1 or Manhattan metric the distance d from a point x to a point y on a k-dimensional space is given by

$$d(x, y) = \sum_i |x_i - y_i| \, , \; 1 \leq i \leq k$$

whereas in the l_∞ metric we have that

$$d(x, y) = max_i \, |x_i - y_i| \, , \; 1 \leq i \leq k.$$

The cost of the expression is evaluated by embedding its rooted binary tree onto the architecture and then finding the minimum cost of evaluating it using an specific metric. Subexpressions needed to evaluate an expression are in turn evaluated where doing so is cheapest, i.e. at the closest processors among a set of processors at which the evaluation of the subexpression is possible, to the processor which will evaluate the expression. The authors do not assume the owner-computes rule. As an example of what is presented by Gilbert and Schreiber [31] we have in Figure 1.4(a) four arrays to be combined in the expression $(w \oplus x) \otimes (y \odot z)$, where $\oplus$, $\otimes$, and $\odot$ are array operators. Each point in Figure 1.4 is a processor and so we could think of a grid of processors as the architecture which is used. In Figure 1.4(b) we have the result of applying this method to the expression above. Region A in Figure 1.4(b) represents the set of processors that should evaluate

$w \oplus x$, i.e. the set of processors for which the cost of evaluating $w \oplus x$ is minimal. Similarly, region B represents the set of processors that should evaluate subexpression $y \odot z$, and region C the set of processors that should evaluate the final expression, i.e. the root. Assume that processor p in region C is chosen to evaluate the final expression among all processors that can evaluate it. Then the processor in A which is closest to processor p is chosen to evaluate $w \oplus x$, and the processor in B which is closest to processor p is chosen to evaluate $y \odot z$. They both send their partial results to p which then evaluates the final expression.

Gilbert and Schreiber [31] use an approach where the processors at which the expression under consideration, as well as its subexpressions, should be evaluated to minimize cost are found. The article does not deal with neither data, nor computation decompositions. The work pertaining to a computation is performed by several processors. The work by Gilbert and Schreiber [31] is different to our work in that we consider all statements within a loop nest, rather than just one statement at a time. They relax the owner-computes rule and use the l_1-metric, which is in this context, robust and realistic for a grid of processors with nearest neighbor connections. We are concerned with partitioning of both data and computation, vectorization of messages, and mapping transformations to machine communication primitives. We also relax the owner-computes rule, but we assume that the work performed during one iteration is performed by only one processor.

Gupta and Banerjee [33], present a method restricted to partitioning of arrays, i.e. no computation partitioning. In their method Gupta and Banerjee select important segments of code to determine distribution of various arrays based on some constraints. Quality measures are used to choose among contradicting constraints. These quality measures may require user intervention. The compiler *tries* to combine constraints for each array in a consistent manner to minimize overall execution time and the entire program is considered. Small arrays are assumed to be replicated on all processors. The distribution of arrays is by rows, columns, or blocks. This work uses heuristic algorithms to determine the alignment of dimensions, i.e. component alignment, of various arrays since the problem has been shown to be NP-complete. The owner-computes rule is assumed and issues concerning the best way to communicate messages among processors, such as aggregate communication introduced in the work by Tseng [80], are dealt with. Communication costs are determined by Gupta and Banerjee [33] after identifying the pairs of dimensions that should be aligned. Consideration is given to when it would be best to replicate a dimension rather that to distribute it.

The idea is that the algorithm will build the Component Affinity Graph (CAG) developed by Li and Chen [55], as shown in Figure 1.5, and decide to align the first dimension of each of the arrays and also the second dimension since it would be too costly to do otherwise. That is, the cheapest way to partition the node set into $D = 2$ disjoint subsets is by grouping A_1, and B_1 into one subset, and A_2 and B_2 into another subset, where D is the dimensionality of the arrays. In this way the total weight of the edges going from one subset to the other is zero. The cost of choosing a cyclic distribution should make it favorable for the algorithm to choose a contiguous distribution for both dimensions. The

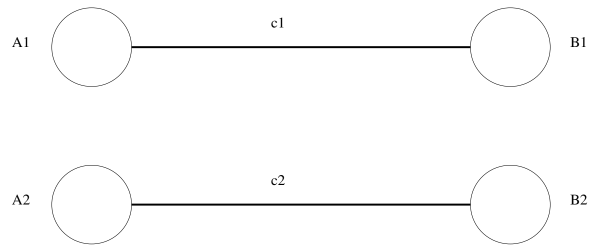


Figure 1.5: Component Affinity Graph (CAG) Partitioned by Classes of Dimensions.

alignment done is in terms of which dimensions should be aligned but it does not calculate how to best aligned them.

The nodes of the CAG represent array dimensions. An edge is added between two nodes for every constraint in the alignment of two dimensions. The weight of the edge is equal to the quality measure of the constraint.

The work by Gupta and Banerjee [33] uses the owner-computes rule, requires user intervention, and does not attempt to compute alignments beyond alignment of dimensions. In our work we address both data and computation alignment, relaxing the owner-computes rule. We address cases of axis alignment, stride and reversal alignments, and offset alignment. We do agree in that small arrays, as scalars, should be replicated and also in optimizing the communication by moving it outside the innermost loop whenever possible.

Bau et al. [14] use elementary matrix methods to determine communication-free alignment of code and data. They also deal with the problem of replicating read-only data to eliminate communication. They incorporate data dependences in their proposed solution to the problem, but the owner-computes rule is assumed. Replication of data is also incorporated into their proposed solution. This method will be discussed in detail in Chapter 4.

Amarasinghe et al. [5], show how to find partitions for *doall* and *doacross* parallelism and, in order to minimize communication across loop nests, they use a greedy algorithm that tries to avoid the largest amounts of potential communication. They give examples of how to obtain parallelism by incurring some communication when this is the only way to run in parallel.

Chatterjee et al. [23] and [24] provide an algorithm that obtains alignments which are more general that the owner-computes rule by decomposing alignment functions into several components. Chatterjee [23] et al. investigate the problem of evaluating Fortran 90 style array expressions on massively parallel distributed-memory machines. They present algorithms based on dynamic programming. There are a number of other researchers who have also made contributions to this problem. Kim and Wolfe [50] show how to find and operate on the communication *pattern matrix* from user-aligned references. Our approach generates the alignment of data and computation and frees the user from this task. Li and Pingali [56] start with user specified data distributions and develop a systematic loop transformation strategy identified by them as *access normalization* which restructures loop nests to exploit locality and block transfers whenever possible. Although we are also interested in maintaining locality our approach and theirs are different. We develop the data and computation distributions based on our findings, the user does not have to specify them.

O'Boyle [61] proposed an automatic data partition algorithm based on the analysis of four distinct factors. We concur with him in his view that automatic data partitioning is possible and that it must be considered in the context of the whole compilation process rather than be left to the programmer. He does not consider partitioning of computation along with that of data and he is not concerned with finding the alignment that will minimize communication as we are in our work. Wakatani and Wolfe [81] address the problem of minimizing communication overhead but from a different context than ours. They are concerned with the communication arising from the redistribution of an array and proposed a technique called *strip mining redistribution*. They are not concerned with automatically generating the alignments as we are in order to free the programmer from this task and achieve minimum communication while preserving parallelism.

Chatterjee et al. [20] and Sheffer et al. [76] deal with determining both static and dynamic distributions. They use the Alignment-Distribution Graph (ADG) whose nodes represent program operations, the ports in the nodes represent array object manipulated by the program, and the edges connect array definitions to their respective uses. The ADG is a directed edge-weighed graph although it is used as an undirected graph. Communication occurs when the alignment or distribution at the end points of an edge is different. The completion time of a program is modeled as the sum of the cost over all the nodes (which accounts for computation and realignment) plus the sum over all the edges of the redistribution time (which takes into account the cost per data item of all-to-all personalized communication, the total data volume, and the discrete distance between distributions).

Ayguadé et al.'s [8] main effort is directed toward intra-procedural data mappings. Candidate distributions are used to build a search space from which to determine, based on profitability analyses, the points at which to realign or redistribute the arrays in order to improve the performance by reducing the total data movement. The Component Affinity Graph (CAG) of Li and Chen [55] is used to determine the best local distribution for a particular phase of the code. All the arrays in a phase are distributed identically. Control flow information is used for phase sequencing identification. An intra-procedural remapping algorithm is provided.

Garcia et al. [30] present an approach to automatically perform static distribution using a constraint based model on the Communication-Parallelism Graph (CPG). The CPG contains edges representing both communication and parallelization constraints. The constraints are formulated and solved using a linear 0-1 integer programming model and solver. They obtain solution for one-dimensional array distributions, i.e. only one dimension of the arrays is distributed, and use an iterative approach for the multi-dimensional problem.

Kremer [52] proves the dynamic remapping problem NP-complete. Kremer et al. [53] and Kremer [51] consider the profitability of dynamic remapping and use an interactive tool for automatic data layout, respectively.

Kennedy and Kremer [48, 49] deal with dynamic remapping in Fortran D [80] and HPF [41]. The work by Kennedy and Kremer propose a way to solve the NP-complete inter-dimensional alignment problem [52] using a state-of-the-art general purpose integer programming solver [49]. Thus Kennedy and Kremer [49] formulate the inter-dimensional alignment problem as a 0-1 integer programming problem. The same is done by Bixby et al. [17].

Palermo and Banerjee [63] deal with dynamic partitioning by building the Communication Graph. In this graph the nodes correspond to statements in the program and the edges are flow dependences between the statements. The weight on these edges reflect communication. Maximal cuts are used to remove largest communication constraints and recursively divide the graph or subgraphs until chunks of code (phases) that should share the same partitioning schemes are grouped together. Thus remapping may be inserted between phases and not within a particular phase to reduce communication between phases.

Although we do not intend to go over all the issues related to parallelizing compilers in this work, we do believe that providing a good reference list will help those unfamiliar with the literature coverage on this subject. On the general theory of dependence analysis and vectorization the reader is referred to Padua and Wolfe [62], Wolf and Lam [82], Wolfe [83, 84, 85, 86, 87, 88, 89], Banerjee [10, 11, 12, 13], Goff [32], Maydan et al. [59], Allen et al. [3], Allen and Kennedy [2, 4], Cytron [26], Moldovan and Fortes [60], Irigoin and Triolet [46], Ramanujam and Sadayappan [69], Zima and Chapman [91], Blume and Eigenmann [18]. For cache and locality issues Gannon et al. [29], Gallivan et al. [28], Anderson and Lam [6], Fang and Lu [27]. For a good coverage of number theory the books by Rosen [73], and by Zima and Chapman [91]. In the articles by Chatterjee et al. [21, 24], and by Stichnoth [78] the reader can find an introduction to the issues related with the assignment of array elements to the local memory of processors and how these are accessed. Alignment is treated by Chatterjee et al. [21, 22, 24]. Parallel machines and algorithms are covered in the books by Quinn [66], JáJá [47], and Stone [79]. High Performance Fortran and related issues are covered in [41], Hiranandani et al. [42, 43, 44]. Communication-free compiling is the main topic of Huang and Sadayappan [45], Ramanujam and Sadayappan [68], and Fang and Lu [27] while data-flow analysis is treated in the books by Aho et al. [1], Parsons [64], and the article by Maydan et al. [58]. On inter procedural analysis issues the reader is referred to Aho et al. [1], Hall et al. [35, 37, 38], Hall [36], Hall and Kennedy [39], Callahan et al. [19], Havlak and Kennedy [40], Richardson and Ganapathi [71, 72], Shah [75], Cooper et al. [25], and Sebesta [74]. Finally, compiling for distributed memory machines is the topic of Tseng [80], Hiranandani et al. [43, 44], Bal et al. [9], Zima and Chapman [92], Gupta et al. [34].

CHAPTER 2

USING LINEAR PROGRAMMING TO SOLVE THE ALIGNMENT PROBLEM

In this chapter we present solutions to the alignment problem by first modeling it as a general linear programming (LP) problem and then using an LP tool to solve it. Our method will determine a non-trivial communication-free solution if it exists. Otherwise, our method will determine a solution that minimizes communication. The problem is modeled using the Manhattan or l_1 metric. Solutions are presented for the offset alignment problem along with some cases of the axis and stride alignment problems. Results are included for several benchmarks including: Jacobi, Alternate-Direction-Implicit (ADI), Disper, Livermore 18, Livermore 23, and Shallow. This chapter is organized as follows: Section 2.1 presents the stride and reversal alignment and how to solve a class of these problems using Linear Programming techniques. Section 2.2 formulates the offset alignment problem, shows how to represent and solve this problem as a linear programming problem, and solutions to some real life problems. In Section 2.3 a class of the axis alignment problem and its solution are presented. Section 2.4 deals with replication alignment. Section 2.5 talks about the type of solutions we will obtain using the Linear Programming approach. Section 2.6 presents other work in this subject and Section 2.7 a chapter summary.

2.1 STRIDE AND REVERSAL ALIGNMENT

When the coefficients of the loop index variables in the subscript expressions of the array references in a program are greater than unity, we have what has been termed as stride alignment. If any of these coefficients is negative, then it is called reversal alignment. Reversal alignment corresponds to mapping the reflection of the array onto the template Tseng [80]. Stride alignment is generated by statements similar to the following

$$\texttt{ALIGN}\ X[i]\ \texttt{WITH}\ T1[\alpha_X i + \beta_X]$$

where α_X is a positive number, and β_X can be any number. A multidimensional array example is given in the statement

$$\texttt{ALIGN}\ Y[i,j]\ \texttt{WITH}\ T2[\alpha_{Y_1} i + \beta_{Y_1}, \alpha_{Y_2} j + \beta_{Y_2}].$$

An example of reversal alignment is shown in the following statement

$$\texttt{ALIGN}\ Y[i]\ \texttt{WITH}\ T3[-i].$$

Consider the following loop, where array Y is not replicated onto the available processors:

```
ALIGN X(i) WITH T(3i+4)
ALIGN Y(i) WITH T(2i+1)
ALIGN i WITH T(6i+1)
DO i = 1, N
    X(2i-1) = Y(3i-1) + Y(3i) + Y(3i+1)
ENDDO
```

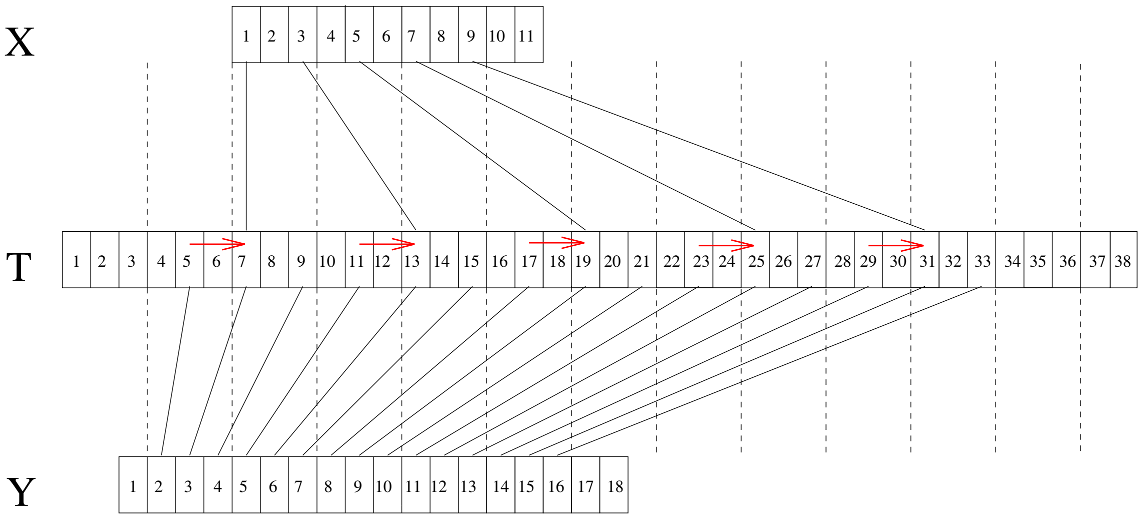


Figure 2.1: Relative Alignment of Arrays X and Y with Respect to Template T Using Computation Alignment.

DO $i = 1, N$
$$X[2i-1] = Y[3i-1] + Y[3i] + Y[3i+1]$$
ENDDO.

Figures 2.1 and 2.2 show two ways of aligning arrays X and Y. These alignments were found by inspection. With the alignment shown in Figure 2.1, each processor computing an iteration will need an element of Y which is held by the processor to its left, assuming a linear array and a block distribution with block size equal to 3, except of course for the processor at the leftmost position.

Using the owner-computes rule we can obtain the alignment shown in Figure 2.2. This would require that each processor send two elements to the processor on its left (except for the last processor) assuming the same configuration and distribution as before (see Figure 2.2). Note that there are infinitely many ways to align the arrays in the problem above. We are, however, concerned with determining the one that results in the least interprocessor communication and this is the reason why it is important to relax the owner-computes rule. If performing the computation at some processor will result in

```
ALIGN X(i) WITH T(3i+1)
ALIGN Y(i) WITH T(2i+1)
DO i = 1, N
      X(2i-1) = Y(3i-1) + Y(3i) + Y(3i+1)
ENDDO
```

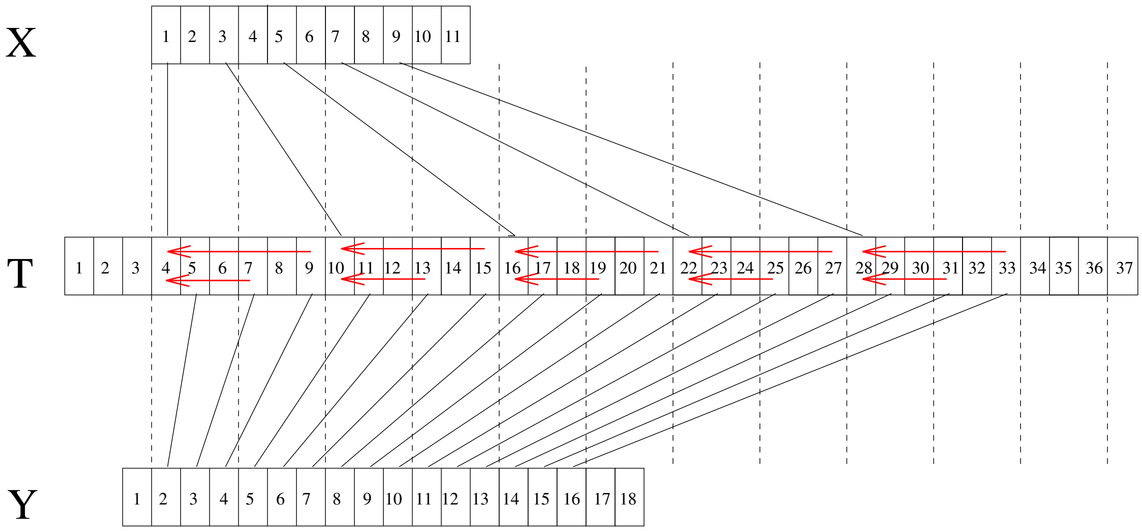


Figure 2.2: Relative Alignment of Arrays X and Y with Respect to Template T without Using Computation Alignment.

the least communication, then that processor should indeed be the one carrying out the computation regardless of whether or not it owns the *lhs* element.

We will now develop a formulation for the stride alignment problem as a general linear programming (LP) problem and show how to use an LP tool to solve it. Later on we will use this formulation to derive the offset alignment formulation. This is because the offset alignment case can be viewed as an special case of the stride alignment problem. Consider the following piece of code where the arrays X and Y (which are not replicated) are aligned to a template T as specified above and α_X, β_X, α_Y, β_Y, α_I, and β_I are to be determined.

$\texttt{ALIGN}\ X[i]\ \texttt{WITH}\ T[\alpha_X i + \beta_X]$
$\texttt{ALIGN}\ Y[i]\ \texttt{WITH}\ T[\alpha_Y i + \beta_Y]$
$\texttt{ALIGN}\ i\ \texttt{WITH}\ T[\alpha_I i + \beta_I]$
DO $i = 1, N$
$\quad X[a_1 i + b_1] = Y[c_1 i + d_1] + Y[c_2 i + d_2] + \cdots + Y[c_r i + d_r]$
ENDDO

Consider iteration i only. Since we are concerned with alignment, we assume as many processors as needed. We want to minimize the distance from the processor(s) holding the elements of arrays X and Y that are needed to perform the computation of the element on the left hand side, to the processor which will be performing the computation during iteration i. Using the alignment specified above we find that the processor which holds the element on the *lhs* is processor $\alpha_X(a_1 i + b_1) + \beta_X$. Similarly, the processor holding the first term of array Y is processor $\alpha_Y(c_1 i + d_1) + \beta_Y$, the one holding the second term is $\alpha_Y(c_2 i + d_2) + \beta_Y$, and so on. Thus the distance between the processor which holds the *lhs* element and the processor which performs the computation during iteration i is given by

$$|[\alpha_X(a_1 i + b_1) + \beta_X] - [\alpha_I i + \beta_I]| .$$

Similarly we find the distance from the processor(s) holding each one of the elements on the right hand side to be

$$|[\alpha_Y(c_j i + d_j) + \beta_Y] - [\alpha_I i + \beta_I]| , \; 1 \leq j \leq r.$$

Combining all the terms shown in these two expressions we find the sum of the distances of each processor holding an element of X and each processor holding and element of Y from the processor which performs the computation during iteration i as follows:

$$|(\alpha_X(a_1 i + b_1) + \beta_X) - (\alpha_I i + \beta_I)| + \sum_{j=1}^{r} |\alpha_Y (c_j i + d_j) + \beta_Y - (\alpha_I i + \beta_I)| .$$

We now include all the iterations to come up with the following equation:

$$\begin{aligned} \text{total distance} \quad &= \quad \sum_{i=1}^{N} |(\alpha_X(a_1 i + b_1) + \beta_X) - (\alpha_I i + \beta_I)| \\ &+ \quad \sum_{j=1}^{r}\sum_{i=1}^{N} |\alpha_Y (c_j i + d_j) + \beta_Y - (\alpha_I i + \beta_I)| . \end{aligned} \tag{2.1}$$

We want to minimize this sum of distances, i.e. ,

$$\begin{aligned} \text{minimize} \quad &\left\{ \sum_{i=1}^{N} |(\alpha_X(a_1 i + b_1) + \beta_X) - (\alpha_I i + \beta_I)| \right. \\ &\left. + \sum_{j=1}^{r}\sum_{i=1}^{N} |\alpha_Y (c_j i + d_j) + \beta_Y - (\alpha_I i + \beta_I)| \right\} . \end{aligned} \tag{2.2}$$

Collecting terms and rearranging we obtain the following:

$$\text{minimize} \quad \left\{ \sum_{i=1}^{N} |(\alpha_X a_1 - \alpha_I) i + \alpha_X b_1 + \beta_X - \beta_I| \right.$$

$$+\sum_{j=1}^{r}\sum_{i=1}^{N}|(\alpha_Y c_j-\alpha_I)i+\alpha_Y d_j+\beta_Y-\beta_I|\Bigg\}. \tag{2.3}$$

We can generalize the above expression to the case when we have an arbitrary number l of loop nests, an arbitrary number w of statements over the various loop nests (w^g is the number of statements in loop nest g), and q is the total number of arrays in the program which are actually used. In this way we can obtain a general expression for the minimization of the total distance over the entire program. Note that only the arrays which are actually used in the program are considered here.

The generalized expression is:

$$\text{minimize}\ \left\{\sum_{g=1}^{l}\sum_{u=1}^{w^g}\left\{\sum_{k=1}^{q}\sum_{j=1}^{r_k}\sum_{i=1}^{N}\left|\left(\alpha_{Y_k}c^{g,u}_{jY_k}-\alpha_I\right)i+\alpha_{Y_k}d^{g,u}_{jY_k}+\beta_{Y_k}-\beta_I\right|\right.\right.$$
$$\left.\left.+\sum_{i=1}^{N}\left|(\alpha_X a^{g,u}-\alpha_I)\,i+\alpha_X b^{g,u}+\beta_X-\beta_I\right|\right\}\right\} \tag{2.4}$$

where $r_k \geq 0$ and represents the number of terms of an array Y_k that appear on the right hand side of statement u in loop nest g, and $q \geq 1$. Note that in Equation 2.4, X is used for the array which appears on the *lhs* of a statement u in loop nest g, and Y_k is used for the k^{th} occurrence of an array Y which appears on the *rhs* of statement u in loop nest g, including X.

Adopting the convention that Y_1 corresponds to the array on the lhs of statement u in loop nest g, and accounting for the term Y_1 in r_1, we can rewrite Equation 2.4 as shown below

$$\text{minimize}\ \left\{\sum_{g=1}^{l}\sum_{u=1}^{w^g}\sum_{k=1}^{q}\sum_{j=1}^{r_k}\sum_{i=1}^{N}\left|\left(\alpha_{Y_k}c^{g,u}_{jY_k}-\alpha_I\right)i+\alpha_{Y_k}d^{g,u}_{jY_k}+\beta_{Y_k}-\beta_I\right|\right\}. \tag{2.5}$$

Note that for any array Y_k for which $r_k = 1$ we can reduce its contribution to the equation above to zero by choosing $\alpha_I = \alpha_{Y_k}c_{1_{Y_k}}$, and $\beta_I = \beta_{Y_k} + \alpha_{Y_k}d_{1_{Y_k}}$. This will also be the case if $r_k > 1$ and the subscript expressions for array Y_k are always the same. In this case we can use these equations as constraints on the values of both α_I and β_I and we can also use it to impose constraints on the values of α_{Y_k} and β_{Y_k}. This constraints are easily added to our model using the method outlined below.

Assume for now that we have only one loop nest, i.e. $g = 1$, and one statement, i.e. $w^g = 1$, at a time. Let us also assume that the stride terms for each distinct array are the same. Then each of the c coefficients for a particular array will be equal, i.e. $c_{1_{Y_k}} = c_{2_{Y_k}} = \cdots = c_{r_{Y_k}}$ so that our equation becomes

$$\text{minimize}\ \left\{N\sum_{k=1}^{q}\sum_{j=1}^{r_k}\left|\alpha_{Y_k}d_{jY_k}+\beta_{Y_k}-\beta_I\right|\right\} \tag{2.6}$$

or

$$\text{minimize } \left\{ N \sum_{k=1}^{q} \sum_{j=1}^{r_k} z_{kj} \right\} \tag{2.7}$$

where

$$z_{kj} = \left| \alpha_{Y_k} d_{jY_k} + \beta_{Y_k} - \beta_I \right| . \tag{2.8}$$

We want to solve this problem using a linear programming (LP) solver, however, LP solvers do not accept absolute values of variables. We solve this problem by adding the following pair of constraints for each z_{kj} term in the objective function

$$z_{kj} - d_{jY_k} \alpha_{Y_k} - \beta_{Y_k} + \beta_I \geq 0$$

and

$$z_{kj} + d_{jY_k} \alpha_{Y_k} + \beta_{Y_k} - \beta_I \geq 0.$$

This results in

$$z_{kj}(optimal) \ = \left| \alpha_{Y_k} d_{jY_k} + \beta_{Y_k} - \beta_I \right| .$$

Thus, our problem could be formulated as

$$\text{minimize } \left\{ N \sum_{k=1}^{q} \sum_{j=1}^{r_k} z_{kj} \right\} \tag{2.9}$$

subject to

$$z_{kj} - d_{jY_k} \alpha_{Y_k} - \beta_{Y_k} + \beta_I \geq 0 \tag{2.10}$$

$$z_{kj} + d_{jY_k} \alpha_{Y_k} + \beta_{Y_k} - \beta_I \geq 0 \tag{2.11}$$

$$\alpha_I - c_{Y_k} \alpha_{Y_k} = 0 \tag{2.12}$$

and

$$z_{kj},\ \alpha_I,\ \beta_I,\ \alpha_{Y_k},\ \beta_{Y_k} \geq 0 \tag{2.13}$$

for all values of k, and j.

Equation 2.9 together with the constraints 2.10, 2.11, and 2.13 describe a linear programming problem. Equation 2.12 is added as a new constraint on the values of α_I and α_{Y_k}.

To allow our variables to take on either positive or negative values we will use the concept of the positive and negative parts of a real number. Banerjee [11] defines the positive part a^+ and the negative part a^- of a real number a as $a^+ = max(a, 0)$ and $a^- = max(-a, 0)$ such that $a^+ = a$ and $a^- = 0$ for $a \geq 0$, and $a^+ = 0$ and $a^- = -a$ for $a \leq 0$. Note that $a = a^+ - a^-$.

Replacing $\alpha^+_{Y_k} - \alpha^-_{Y_k}$, $\alpha^+_I - \alpha^-_I$, $\beta^+_{Y_k} - \beta^-_{Y_k}$, and $\beta^+_I - \beta^-_I$ for α_{Y_k}, α_I, β_{Y_k}, and β_I, respectively, and adding the constraints

$$\alpha^+_{Y_k},\ \alpha^-_{Y_k},\ \beta^+_{Y_k},\ \beta^-_{Y_k},\ \alpha^+_I,\ \alpha^-_I,\ \beta^+_I,\ \beta^-_I \geq 0$$

the new constraints for Equation 2.9 become

$$z_{kj} - d_{jY_k}(\alpha^+_{Y_k} - \alpha^-_{Y_k}) - (\beta^+_{Y_k} - \beta^-_{Y_k}) + (\beta^+_I - \beta^-_I) \geq 0 \tag{2.14}$$

$$z_{kj} + d_{jY_k}(\alpha^+_{Y_k} - \alpha^-_{Y_k}) + (\beta^+_{Y_k} - \beta^-_{Y_k}) - (\beta^+_I - \beta^-_I) \geq 0 \tag{2.15}$$

$$\alpha^+_I - \alpha^-_I - c_{Y_k}(\alpha^+_{Y_k} - \alpha^-_{Y_k}) = 0 \tag{2.16}$$

$$\alpha^+_I - \alpha^-_I \geq 1,\ \alpha^+_{Y_k} - \alpha^-_{Y_k} \geq 1 \tag{2.17}$$

and

$$z_{kj},\ \alpha^+_{Y_k},\ \alpha^-_{Y_k},\ \beta^+_{Y_k},\ \beta^-_{Y_k},\ \alpha^+_I,\ \alpha^-_I,\ \beta^+_I,\ \beta^-_I \geq 0 \tag{2.18}$$

for all values of j and k. If we do not want a specific variable to be negative we can just add the constraint that the corresponding negative part be equal to zero. For example, if we do not want reversal alignment we would add constraints specifying that the negative parts of the values for the α's are set to zero.

Polynomial time algorithms have been discovered that solve the general linear programming problem, although the general integer programming problem is still NP-hard [47]. We use one of these LP solvers to solve for the unknown variables that will minimize Equation 2.9 subject to the conditions stated in Equations 2.14 thru 2.18. Note that with this method we add a new variable for each original equation and that each original constraint equation is replaced by two new equations. This method can be used for both stride and reversal alignment as specified previously.

After running the model for the example problem presented earlier we obtained the following results: $\alpha_I = 3$, $\alpha_X = 3/2$, $\alpha_Y = 1$, $\beta_I = \beta_Y = 0$, and $\beta_X = 3/2$. Note that these are all rational numbers. In order to convert them to integers we simply multiply by the least common multiple of the denominators; note that an integer i is written as $\frac{i}{1}$. For this example we use two, and obtain $\alpha_I = 6$, $\alpha_X = 3$, $\alpha_Y = 2$, $\beta_I = \beta_Y = 0$, and $\beta_X = 3$. These integer values are very similar to the values we obtained by inspection. The basic

```
ALIGN X(i) WITH T(3i+3)
ALIGN Y(i) WITH T(2i)
ALIGN i WITH T(6i)
DO i = 1, N
    X(2i-1) = Y(3i-1) + Y(3i) + Y(3i+1)
ENDDO
```

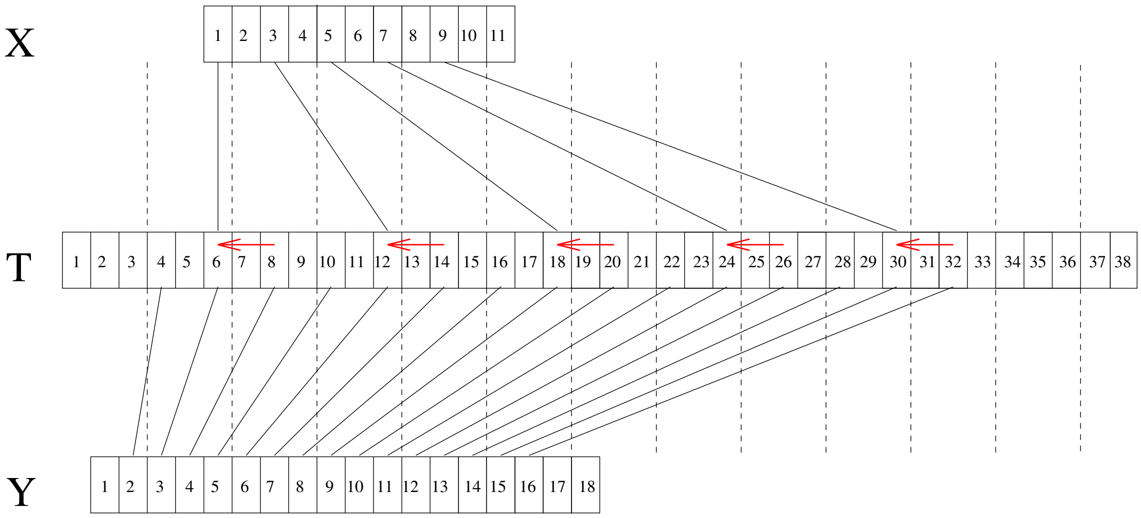


Figure 2.3: Relative Alignment of Arrays X and Y with Respect to Template T Using Computation Alignment and Our LP Method.

difference is in the value of the constant offset coefficients. We note that we still have communication arising from these constant offset terms. As shown in Figure 2.3 each processor needs one element from the processor to its right, except for the last processor. This is assuming the same block distribution of three elements per processor as before. This type of communication, however, is not too expensive if the block size exceeds the maximum absolute value of the offsets since it is between nearest neighbors. But we have taken care of a more expensive type of communication, i.e. that which arises from the stride terms which may not be among nearest neighbors. In terms of cost, the most expensive type of communication among processors is the one due to axis misalignment since it involves all processors across more than one dimension, followed by stride which is within a dimension, then replication from one to many processors, and offset.

2.2 Offset Alignment

Among all the different types of alignment the most common type is offset alignment. This is because nonzero coefficients of loop indexes in most subscripts expressions are either 1 or -1 [77]. All of the benchmarks used in this dissertation are examples of offset alignment. Offset alignment can be viewed as an special case of stride alignment where all the stride coefficients are equal to one. An example will help illustrate how this type of alignment arises. Suppose that we have the loop shown below:

DO $i = 1, N$
    $X[i] = Y[i+2]$
ENDDO.

Note that the coefficients that multiply the index variable in the subscript expressions have a value of one. Remember that, in order for arrays X and Y to be aligned they must be mapped to the same template, and that it is their relative positions to this template that will determine how they are aligned with respect to each other. For offset alignment we have alignment statements of the form

$$\texttt{ALIGN}\ X[i]\ \texttt{WITH}\ T1[i-1]$$

and

$$\texttt{ALIGN}\ Y[i]\ \texttt{WITH}\ T1[i+4].$$

Figures 2.4 and 2.5 illustrate several ways in which these arrays could be aligned to a template T. If we align the arrays as shown in Figure 2.4(a), then we find that two elements need to be sent to the processor on the left, assuming a linear array and a block distribution with block size three. This communication is illustrated in Figure 2.4(b). If on the other hand we choose to align arrays X and Y as shown in Figure 2.5, we obtain perfect alignment of the arrays. As a result the communication among processors is zero and the code can be executed in parallel.

There is a basic difference in the way the alignment is performed in Figure 2.5(a) and (b). In Figure 2.5(a) computation alignment is used to map the iterations of the loop to the processor that will minimize communication because of its position in the processor array with respect to the position of the processors holding the *rhs* operands. In Figure 2.5(b) the owner-computes rule is used and the computation is always performed by the processor holding the *lhs* element. This is done without any regard to the position of the elements on the *rhs*. Although for this particular example either method can result in zero communication, the advantage of the computation alignment method over the owner-computes rule was illustrated in Section 2.1.

Now consider the following code:

$\texttt{ALIGN}\ X[i]\ \texttt{WITH}\ T[i+\beta_X]$
$\texttt{ALIGN}\ Y[i]\ \texttt{WITH}\ T[i+\beta_Y]$
$\texttt{ALIGN}\ i\ \texttt{WITH}\ T[i+\beta_I]$
DO $i = 1, N$
    $X[i+b_1] = Y[i+d_1] + Y[i+d_2] + \cdots + Y[i+d_r]$
ENDDO

```
ALIGN X(i) WITH T(i)
ALIGN Y(i) WITH T(i)
DO i = 1, N
    X(i) = Y(i+2)
ENDDO
```

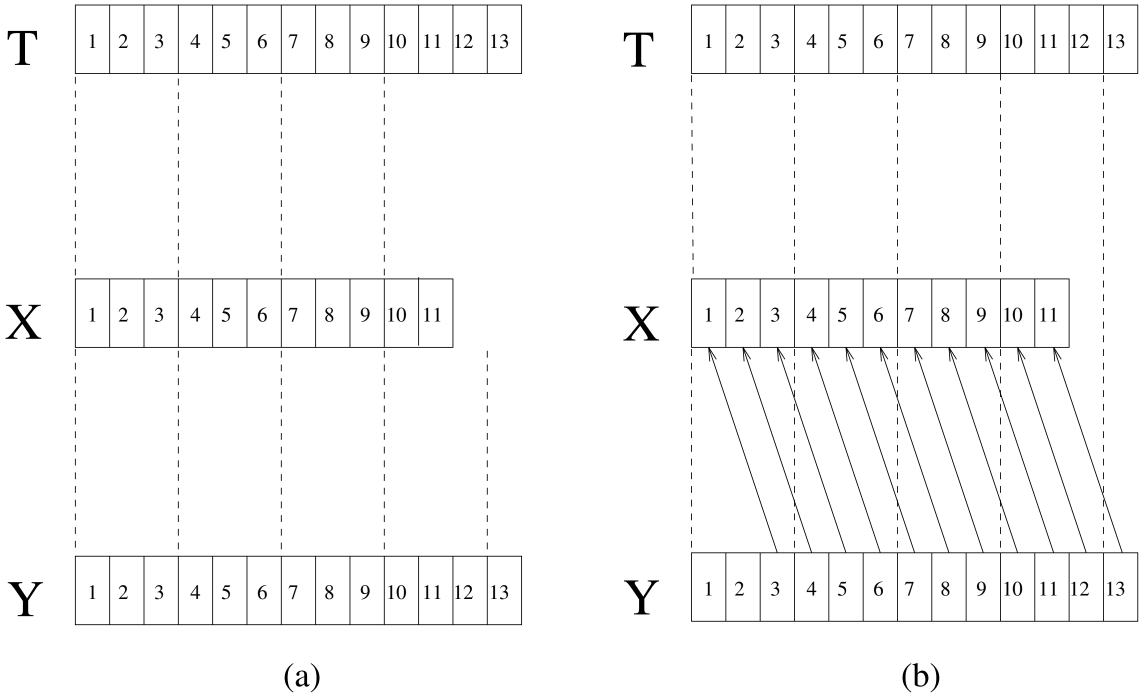


Figure 2.4: (a) Relative Alignment of Arrays X and Y with Respect to Template T (b) Elements of Array Y That Need to Be Copied Onto X.

```
ALIGN X(i) WITH T(i+4)
ALIGN Y(i) WITH T(i+2)
ALIGN i WITH T(i+4)
DO i = 1, N
    X(i) = Y(i+2)
ENDDO
```

```
ALIGN X(i) WITH T(i)
ALIGN Y(i) WITH T(i-2)
DO i = 1, N
    X(i) = Y(i +2)
ENDDO
```

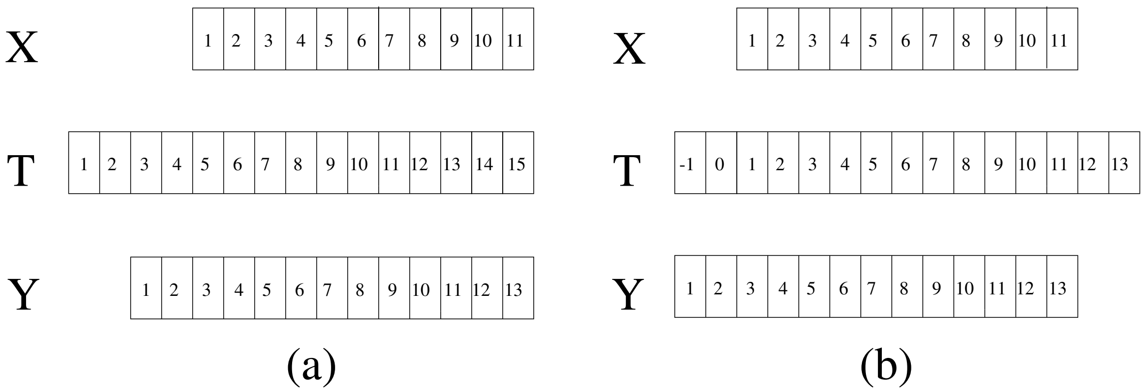


Figure 2.5: Perfect alignment of arrays X and Y (a) Using Computation Alignment and (b) without Computation Alignment.

where arrays X and Y (not replicated) are aligned to a template T as specified above and β_X, β_Y, and β_I are to be determined.

Note that this is the same problem we have already dealt with in Section 2.1, but in this case the stride coefficients c are all equal to one. Thus, we arrive at the following equation by replacing each alignment coefficient α in Equation 2.6 with one.

$$\text{minimize} \quad \left\{ N \sum_{k=1}^{q} \sum_{j=1}^{r_k} \left| d_{jY_k} + \beta_{Y_k} - \beta_I \right| \right\} \tag{2.19}$$

or

$$\text{minimize} \quad \left\{ N \sum_{k=1}^{q} \sum_{j=1}^{r_k} z_{kj} \right\} \tag{2.20}$$

where

$$z_{kj} = \left| d_{jY_k} + \beta_{Y_k} - \beta_I \right| . \tag{2.21}$$

To solve this problem using a linear programming (LP) solver we introduce additional variables and constraints as needed and arrive at the following equations:

$$z_{kj} - d_{jY_k} - (\beta_{Y_k}^+ - \beta_{Y_k}^-) + (\beta_I^+ - \beta_I^-) \geq 0 \tag{2.22}$$

$$z_{kj} + d_{jY_k} + (\beta_{Y_k}^+ - \beta_{Y_k}^-) - (\beta_I^+ - \beta_I^-) \geq 0 \tag{2.23}$$

and

$$z_{kj},\ \beta_{Y_k}^+,\ \beta_{Y_k}^-,\ \beta_I^+,\ \beta_I^- \geq 0 \tag{2.24}$$

for all values of j and k. Again, two variables are used for each original variable and two constraint equations for each original equation. Thus, we have doubled the number of original equations and variables. As explained previously, if we do not want a specific variable to be negative we can just add the constraint that the corresponding negative part be equal to zero.

Shown below are several examples of benchmark program segments from Kremer [51] and Tseng [80] and the result of applying our LP method of determining the offset alignment.

The Jacobi algorithm [80] is shown in Figure 2.6. We have labeled the two statements S1 and S2. After applying our LP method to each one of the statements we obtained the results shown in Table 2.1.

The code for the Alternating-Direction-Implicit (ADI) program [51] from is shown here as Figure 2.7. The results after applying our LP method to the ADI program segment are as shown in Table 2.2.

DO $j = 2, 99$
    DO $i = 2, 99$
S1: $A[i,j] = \mathcal{F}(B[i,j-1], B[i-1,j], B[i+1,j], B[i,j+1])$
    ENDDO
ENDDO
DO $j = 1, 99$
    DO $i = 1, 99$
S2: $B[i,j] = A[i,j]$
    ENDDO
ENDDO

Figure 2.6: Jacobi Program Segment.

Table 2.1: Constant Offsets (β's) Found Using LP Method on Jacobi.

STATE-MENT	ARRAY NAME	DIMENSION	
		1ST	2ND
S1	I	0	0
	A	0	0
	B	0	0
S2	I	0	0
	A	0	0
	B	0	0

```
      DO j = 2, N
          DO i = 1, N
S1:           x[i, j] = F1(x[i, j], x[i, j − 1], a[i, j], b[i, j − 1])
S2:           b[i, j] = F2(b[i, j], a[i, j], b[i, j − 1])
          ENDDO
      ENDDO
      DO i = 1, N
S3:           x[i, N] = F3(x[i, N], b[i, N])
      ENDDO
      DO j = N − 1, 1, −1
          DO i = 1, N
S4:           x[i, j] = F4(x[i, j], a[i, j + 1], x[i, j + 1], b[i, j])
          ENDDO
      ENDDO
      DO j = 1, N
          DO i = 2, N
S5:           x[i, j] = F5(x[i, j], x[i − 1, j], a[i, j], b[i − 1, j])
S6:           b[i, j] = F6(b[i, j], a[i, j], b[i − 1, j])
          ENDDO
      ENDDO
      DO j = 1, N
S7:           x[N, j] = F7(x[N, j], b[N, j])
      ENDDO
      DO j = 1, N
          DO i = N − 1, 1, −1
S8:           x[i, j] = F8(x[i, j], a[i + 1, j], x[i + 1, j], b[i, j])
          ENDDO
      ENDDO
```

Figure 2.7: Alternating-Direction-Implicit (ADI) Program Segment.

Table 2.2: Constant Offsets (β's) Found Using LP Method on ADI.

STATE-MENT	ARRAY NAME	DIMENSION ONE	DIMENSION TWO	STATE-MENT	ARRAY NAME	DIMENSION ONE	DIMENSION TWO
S1	I	0	0	S5	I	0	0
	A	0	0		A	0	0
	B	0	1		B	1	0
	X	0	0		X	0	0
S2	I	0	0	S6	I	0	0
	A	0	0		A	0	0
	B	0	0		B	0	0
S3	I	0	-	S7	I	-	0
	B	0	-		B	-	0
	X	0	-		X	-	0
S4	I	0	1	S8	I	1	0
	A	0	0		A	0	0
	B	0	1		B	1	0
	X	0	1		X	1	0

The code for Disper [80] is shown in Figure 2.8 and the results after applying our method are shown in Table 2.3

A program segment is shown for Livermore 18 in Figure 2.9. The results of applying our methods are shown in Table 2.4. The results were as shown in Table 2.4.

A program segment is shown for Livermore 23 in Figure 2.10 [80]. The results of applying our methods are shown in Table 2.5.

The code for Red Black SOR (Successive Over Relaxation) is shown in Figure 2.11 and the results of applying our method on Table 2.6.

We have also obtained the weather prediction program Shallow from [80]. Shallow is a 200 line benchmark that uses stencil computation that applies finite-difference methods

Table 2.3: Constant Offsets (β's) Found Using LP Method on Disper.

STATE-MENT	ARRAY NAME	DIMENSION 1ST	2ND	3RD	4TH	5TH
S3	I	0	0	0	0	0
	GRADY	0	-	-	-	-
	PFMR	0	0	0	0	0
	DDY	0	0	0	-	-

```
⋮
{* compute dispersion terms *}
DO j = 2, 4
        DO i3 = 1, 8
                DO i2 = 1, 8
                        DO i1 = 1, 256
                              ⋮
S3:                           grady[i1] = (pfmr[i1 + 1, i2, i3, j, k]−
                                   pfmr[i1 − 1, i2, i3, j, k]))/
                                   (0.5 ∗ (ddy[i1 + 1, i2, i3]+
                                   ddy[i1 − 1, i2, i3])+
                                   ddy[i1, i2, i3])
                              ⋮
                        ENDDO
                ENDDO
        ENDDO
ENDDO
```

Figure 2.8: Disper: Oil Reservoir Simulation Program Segment.

```
⋮
DO l = 1, time
    DO k = 2, 99
        DO j = 2, 99
S1:         ZA[j,k] = F1(ZP[j-1,k], ZQ[j-1,k], ZM[j-1,k],
                ZR[j-1,k], ZZ[j-1,k], ZA[j-1,k], ZU[j-1,k],
                ZV[j-1,k], ZB[j-1,k])
S2:         ZB[j,k] = F2(ZP[j-1,k], ZQ[j-1,k], ZM[j-1,k],
                ZR[j-1,k], ZZ[j-1,k], ZA[j-1,k], ZU[j-1,k],
                ZV[j-1,k], ZB[j-1,k])
        ENDDO
    ENDDO
    DO k = 2, 99
        DO j = 2, 99
S3:         ZU[j,k] = F3(ZZ[j-1,k], ZZ[j+1,k], ZA[j-1,k],
                ZA[j+1,k], ZU[j-1,k], ZU[j+1,k], ZV[j-1,k],
                ZV[j+1,k], ZB[j-1,k], ZB[j+1,k])
S4:         ZV[j,k] = F4(ZZ[j-1,k], ZZ[j+1,k], ZA[j-1,k],
                ZA[j+1,k], ZU[j-1,k], ZU[j+1,k], ZV[j-1,k],
                ZV[j+1,k], ZB[j-1,k], ZB[j+1,k])
        ENDDO
    ENDDO
    DO k = 2, 99
        DO j = 2, 99
S5:         ZR[j,k] = F5(ZR[j,k], ZU[j,k])
S6:         ZZ[j,k] = F6(ZZ[j,k], ZV[j,k])
        ENDDO
    ENDDO
ENDDO
```

Figure 2.9: Livermore 18 Program Segment.

Table 2.4: Constant Offsets (β's) Found Using LP Method on Livermore 18.

STATE-MENT	ARRAY	DIMENSION ONE	DIMENSION TWO	STATE-MENT	ARRAY	DIMENSION ONE	DIMENSION TWO
S1	I	0	0	S3	I	0	0
	ZA	1	0		ZA	0	0
	ZB	1	0		ZB	0	0
	ZM	1	0		ZU	0	0
	ZP	1	0		ZV	0	0
	ZQ	1	0		ZZ	0	0
	ZR	1	0	S4	I	0	0
	ZU	1	0		ZA	0	0
	ZV	1	0		ZB	0	0
	ZZ	1	0		ZU	0	0
S2	I	0	0		ZV	0	0
	ZA	1	0		ZR	0	0
	ZB	1	0	S5	I	0	0
	ZM	1	0		ZR	0	0
	ZP	1	0		ZU	0	0
	ZQ	1	0	S6	I	0	0
	ZR	1	0		ZV	0	0
	ZU	1	0		ZZ	0	0
	ZV	1	0				
	ZZ	1	0				

⋮

DO $l = 1, time$
  DO $j = 2, 99$
    DO $k = 2, 99$
S1: $QA = \mathcal{F}_1(ZA[k, j+1], ZA[k, j-1], ZA[k+1, j], ZA[k-1, j])$
S2: $ZA[k, j] = \mathcal{F}_2(ZA[k, j], QA)$
    ENDDO
  ENDDO
ENDDO

Figure 2.10: Livermore 23 Program Segment.

Table 2.5: Constant Offsets (β's) Found Using LP Method on Livermore 23.

STATE-MENT	ARRAY NAME	DIMENSION 1ST	2ND
S1	I	0	0
	ZA	0	0
S2	I	0	0
	ZA	0	0

$\vdots$
DO $l = 1, time$
    $\{*\ compute\ red\ points\ *\}$
    DO $j = 3, 999, 2$
        DO $i = 3, 999, 2$
S1: $V[i,j] = \mathcal{F}(V[i,j-1], V[i-1,j], V[i,j+1], V[i+1,j])$
        ENDDO
    ENDDO
    DO $j = 2, 998, 2$
        DO $i = 2, 998, 2$
S2: $V[i,j] = \mathcal{F}(V[i,j-1], V[i-1,j], V[i,j+1], V[i+1,j])$
        ENDDO
    ENDDO
    $\{*\ compute\ black\ points\ *\}$
    DO $j = 2, 998, 2$
        DO $i = 3, 999, 2$
S3: $V[i,j] = \mathcal{F}(V[i,j-1], V[i-1,j], V[i,j+1], V[i+1,j])$
        ENDDO
    ENDDO
    DO $j = 3, 999, 2$
        DO $i = 2, 998, 2$
S4: $V[i,j] = \mathcal{F}(V[i,j-1], V[i-1,j], V[i,j+1], V[i+1,j])$
        ENDDO
    ENDDO
ENDDO

Figure 2.11: Red Black SOR Program Segment.

Table 2.6: Constant Offsets (β's) Found Using LP Method on Red Black SOR.

STATE-MENT	ARRAY NAME	DIMENSION 1ST	DIMENSION 2ND
S1	I	0	0
	V	0	0
S2	I	0	0
	V	0	0
S3	I	0	0
	V	0	0
S4	I	0	0
	V	0	0

to solve shallow-water equations and is a representative of a large class of existing supercomputer applications. The program segment for Shallow is shown in Figure 2.12. The results were as shown in Table 2.7.

With this we conclude our benchmarks results. We will be using these same benchmarks throughout this dissertation and will be referring back to this section for the program segments.

2.3 Axis Alignment

The most expensive type of interprocessor communication can be avoided through axis alignment. Axis alignment arises when we have a dimension permutation in the alignment statement for multidimensional arrays. For example, with

$$\texttt{ALIGN}\ X[i,j]\ \texttt{WITH}\ T1[j,i]$$

we have that each row i of array X is aligned with column i of template $T1$. Likewise, each column j of X is aligned with row j of template $T1$. In other words, the first dimension of array X is aligned with the second dimension of template $T1$ and the second dimension of X is aligned with the first dimension of $T1$ (see Figure 2.13). Other examples include

$$\texttt{ALIGN}\ Z[k,j,i]\ \texttt{WITH}\ T3[i,j,k]$$

where the first, second, and third dimension of array Z are aligned with the third, second, and first dimension of template $T3$, respectively.

Consider the following code segment (where the two-dimensional arrays X and Y are not replicated)

DO $i = 1, N$
  DO $j = 1, N$
    $X[i,j] = Y[i,j] + Y[j,i]$

```
⋮
DO j = 1, N − 1
        DO i = 1, N − 1
S1:             u[i + 1, j] = F_1(psi[i + 1, j + 1], psi[i + 1, j])
S2:             v[i, j + 1] = F_2(psi[i + 1, j + 1], psi[i, j + 1])
        ENDDO
ENDDO
⋮
DO j = 1, N − 1
        DO i = 1, N − 1
S3:             cu[i + 1, j] = F_3(p[i + 1, j], p[i, j], u[i + 1, j])
S4:             cv[i, j + 1] = F_4(p[i, j + 1], p[i, j], u[i, j + 1])
S5:             z[i + 1, j + 1] = F_5(v[i + 1, j + 1], v[i, j + 1], u[i + 1, j + 1],
                        u[i + 1, j], p[i, j], p[i + 1, j], p[i + 1, j + 1], p[i, j + 1])
S6:             h[i, j] = F_6(p[i, j], u[i + 1, j], u[i, j], v[i, j + 1], v[i, j])
        ENDDO
ENDDO
⋮
DO j = 1, N − 1
        DO i = 1, N − 1
S7:             unew[i + 1, j] = F_7(uold[i + 1, j], z[i + 1, j + 1], z[i + 1, j],
                        cv[i + 1, j + 1], cv[i, j + 1], cv[i, j], cv[i + 1, j],
                        h[i + 1, j], h[i, j])
S8:             vnew[i, j + 1] = F_8(vold[i, j + 1], z[i + 1, j + 1], z[i, j + 1],
                        cu[i + 1, j + 1], cu[i, j + 1], cu[i, j], cu[i + 1, j],
                        h[i, j + 1], h[i, j])
S9:             pnew[i, j + 1] = F_9(pold[i, j], cu[i + 1, j], cu[i, j],
                        cv[i, j + 1], cv[i, j])
        ENDDO
ENDDO
```

Figure 2.12: Shallow Program Segment.

Table 2.7: Constant Offsets (β's) Found Using LP Method on Shallow.

STATEMENT	ARRAY	DIMENSION ONE	DIMENSION TWO	STATEMENT	ARRAY	DIMENSION ONE	DIMENSION TWO
S1	I	1	0		U	0	0
	U	0	0		V	0	0
	PSI	0	0	S7	I	1	0
S2	I	0	1		UNEW	0	0
	V	0	0		UOLD	0	0
	PSI	0	0		Z	0	0
S3	I	1	0		CV	0	0
	CU	0	0		H	1	0
	P	1	0	S8	I	0	1
	U	0	0		VNEW	0	0
S4	I	0	1		VOLD	0	0
	CV	0	0		Z	0	0
	P	0	1		CU	0	1
	V	0	0		H	0	1
S5	I	1	1	S9	I	0	0
	Z	0	0		PNEW	0	0
	V	1	0		POLD	0	0
	U	0	1		CU	0	0
	P	1	0		CV	0	0
S6	I	0	0				
	H	0	0				
	P	0	0				

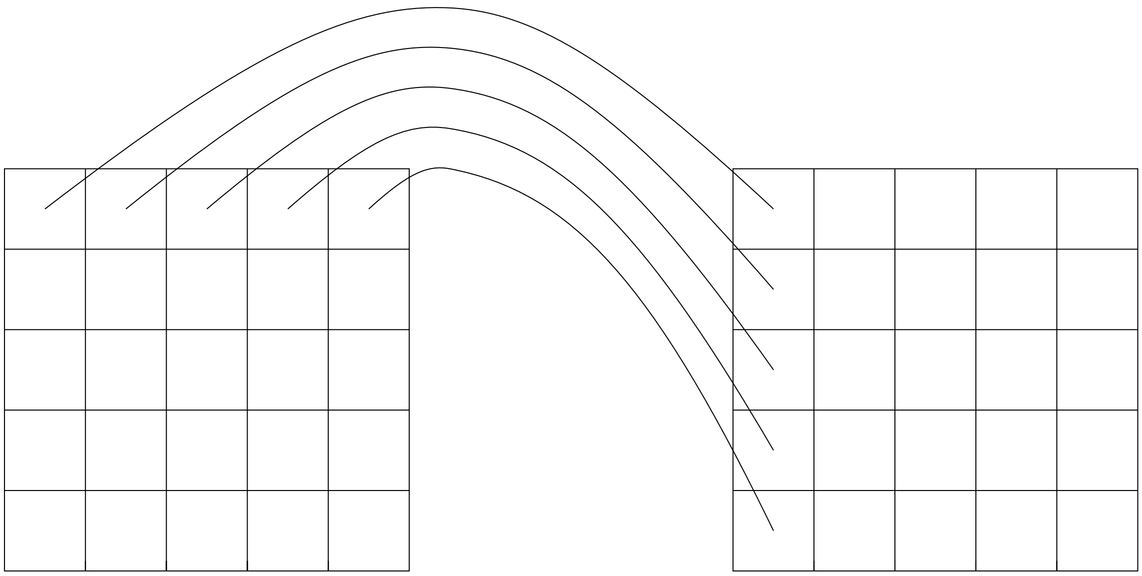


Figure 2.13: Axis Alignment of $X[i,j]$ with $T1[j,i]$.

ENDDO
ENDDO

and assume that the alignment directives for arrays X and Y and iteration i, j will be as follows:

$$\texttt{ALIGN}\ X[i,j]\ \texttt{WITH}\ T[\alpha_X^1 i + \beta_X^1, \alpha_X^2 j + \beta_X^2]$$
$$\texttt{ALIGN}\ Y[i,j]\ \texttt{WITH}\ T[\alpha_Y^1 i + \beta_Y^1, \alpha_Y^2 j + \beta_Y^2]$$
$$\texttt{ALIGN}\ i,j\ \texttt{WITH}\ T[\alpha_I^1 i + \beta_I^1, \alpha_I^2 j + \beta_I^2].$$

Using the l_1 metric the total distance from the processor performing iteration i, j to the processors holding each of the operands would be given by

$$\begin{aligned} \text{distance} &= \left|\left(\alpha_X^1 - \alpha_I^1\right) i + \beta_X^1 - \beta_I^1\right| + \left|\left(\alpha_X^2 - \alpha_I^2\right) j + \beta_X^2 - \beta_I^2\right| \\ &+ \left|\left(\alpha_Y^1 - \alpha_I^1\right) i + \beta_Y^1 - \beta_I^1\right| + \left|\left(\alpha_Y^2 - \alpha_I^2\right) j + \beta_Y^2 - \beta_I^2\right| \\ &+ \left|\alpha_Y^1 j - \alpha_I^1 i + \beta_Y^1 - \beta_I^1\right| + \left|\alpha_Y^2 i - \alpha_I^2 j + \beta_Y^2 - \beta_I^2\right|. \end{aligned} \tag{2.25}$$

Note the last two terms of the equation above. There is no non-trivial way of eliminating the i and the j from the equation above when we consider all the possible values that i and j can take on.

Here the problem is not the metric but the actual mapping. We have a mapping from a two-dimensional array space and a two-dimensional iteration space to a two-dimensional template space, and because of the nature of the problem itself, the approach we have used

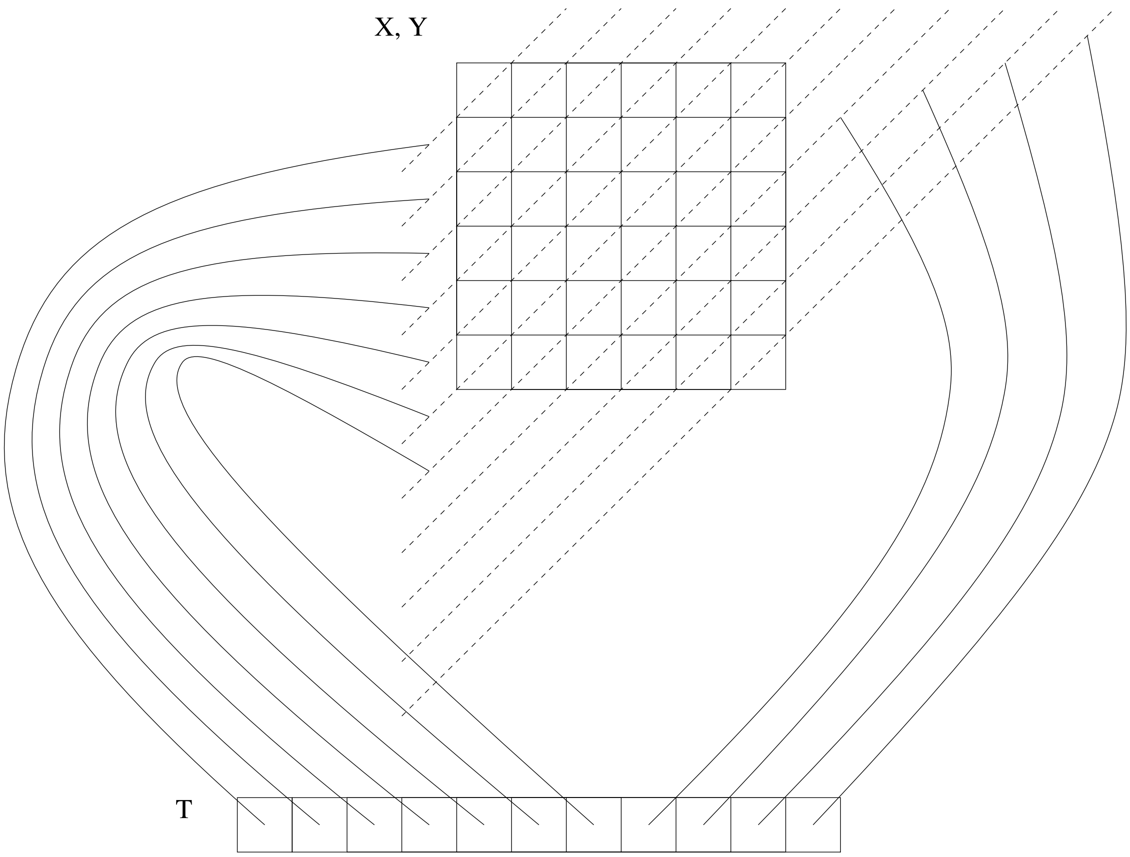


Figure 2.14: Alignment by Diagonals of Arrays X, and Y to Template T.

so far is not very useful for this particular problem. However, consider the same problem but with a different mapping, namely the hyperplane mapping. In particular, consider that arrays X and Y, and the iterations of the loops will be aligned using the directives shown below

$$\texttt{ALIGN}\ X[i,j]\ \texttt{WITH}\ T[\alpha_X^1 i + \beta_X^1 + \alpha_X^2 j + \beta_X^2]$$
$$\texttt{ALIGN}\ Y[i,j]\ \texttt{WITH}\ T[\alpha_Y^1 i + \beta_Y^1 + \alpha_Y^2 j + \beta_Y^2]$$
$$\texttt{ALIGN}\ i,j\ \texttt{WITH}\ T[\alpha_I^1 i + \beta_I^1 + \alpha_I^2 j + \beta_I^2]$$

so that the two-dimensional array space and the two-dimensional iteration space are mapped onto a one-dimensional template. The distance function will then be given by

$$\begin{aligned} \text{distance} &= \left|\left(\alpha_X^1 - \alpha_I^1\right) i + \left(\alpha_X^2 - \alpha_I^2\right) j + \beta_X^1 - \beta_I^1 + \beta_X^2 - \beta_I^2\right| \\ &+ \left|\left(\alpha_Y^1 - \alpha_I^1\right) i + \left(\alpha_Y^2 - \alpha_I^2\right) j + \beta_Y^1 - \beta_I^1 + \beta_Y^2 - \beta_I^2\right| \\ &+ \left|\left(\alpha_Y^2 - \alpha_I^1\right) i + \left(\alpha_Y^1 - \alpha_I^2\right) j + \beta_Y^1 - \beta_I^1 + \beta_Y^2 - \beta_I^2\right|. \end{aligned} \tag{2.26}$$

In order to reduce this distance to zero we require that

$$\alpha^1_X = \alpha^1_I = \alpha^2_X = \alpha^2_I = \alpha^1_Y = \alpha^2_Y.$$

Then the equation will be as follows

$$\left|\beta^1_X - \beta^1_I + \beta^2_X - \beta^2_I\right| + \left|\beta^1_Y - \beta^1_I + \beta^2_Y - \beta^2_I\right| + \left|\beta^1_Y - \beta^1_I + \beta^2_Y - \beta^2_I\right| \quad (2.27)$$

which can be reduced to zero distance by allowing $\beta^1_I = \beta^1_X = \beta^1_Y$ and $\beta^2_I = \beta^2_X = \beta^2_Y$. This means that each diagonal of arrays X and Y, and each diagonal of the iteration space would be mapped to a point in the one-dimensional template. The resulting alignment is illustrated in Figure 2.14.

2.4 REPLICATION

Replication is a type of alignment which arises from the need for every processor to store copies of some data. Examples of replication are shown in Figure 2.15. In this figure each element of the one-dimensional arrays X and Y is aligned with each element of the corresponding template column and row, respectively. Therefore each processor that owns an element of the template will also own the corresponding element of the array. For example, the first part of Figure 2.15 shows the replication of the one-dimensional array X onto template T. As a result of this replication, element 1 of array X is replicated on column 1 of template T, element 2 on column 2, and so on. A processor which is assigned an element of column 1 of template T will also be assigned element $X[1]$, if a processor is assigned an element of column 2 of T, then that processor will also be assigned element $X[2]$ and so on. Similarly, the second part of Figure 2.15 shows a one-dimensional array Y which is replicated on a two-dimensional template $T1$ and as a result each element $Y[i]$ is replicated on row i of $T1$ so that a processor which is assigned an element of row i of $T1$ will also be assigned element $Y[i]$.

In order to illustrate how our method could be used to solve the replication problem we will use an example from Bau et al. [14]. Consider the code shown below

```
   DO i = 1, N
      DO j = 1, N
S:          X[i, j] = X[i, j] + Y[i] * Z[j]
      ENDDO
   ENDDO
```

Note that Y and Z are one-dimensional arrays and that X is a two-dimensional array. Also note that all the elements of Z that are accessed during an iteration of the outer loop are also accessed during the next iteration of the outer loop and that the same element of Y is accessed during all the iterations of the inner loop for a particular iteration of the outer loop. In other words, $Y[i]$ is used to compute all the elements in row i of X and $Z[j]$ is used to compute all the elements in column j of X. Lastly, note that elements of arrays Y and Z are referenced (read) but not modified (written) during each execution of statement S above.

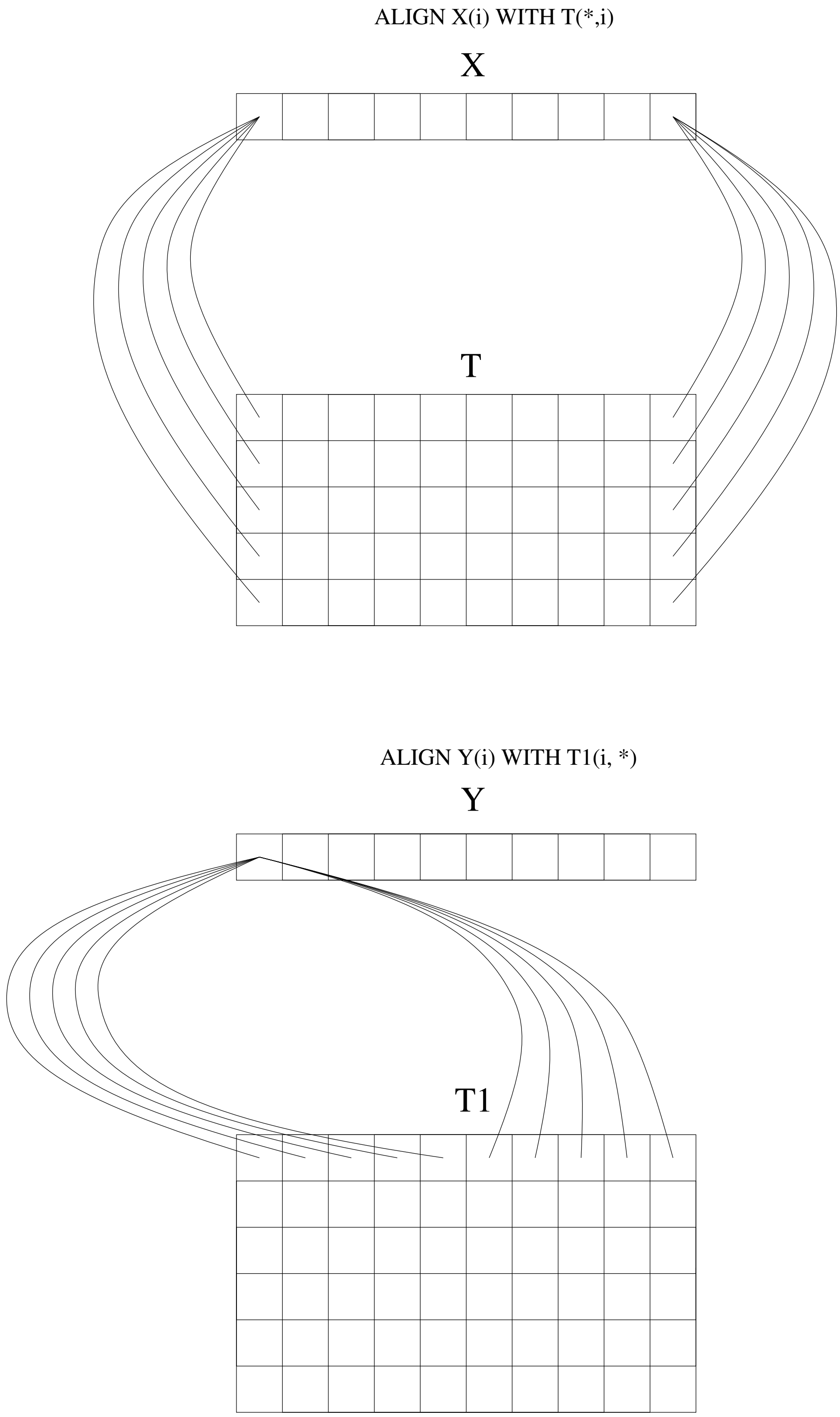


Figure 2.15: Replication of Array X along the Rows of Template T and of Array Y along the Columns of Template $T1$.

As discussed earlier, arrays are aligned to each other by aligning them to a common template. We have seen many examples in which we have aligned arrays which are all of the same dimension. In this case, however, we have arrays of different dimensionality used in the same statement. Since $Y[i]$ is used to compute all the elements in row i of X and $Z[j]$ is used to compute all the elements in column j of X, we would like to align X, Y, and Z such that communication is minimized. In this case we can reduce the communication to zero by replicating arrays Y and Z along the columns and along

We find how to align the arrays using the LP method developed in Section 2.2 for offset alignment. This is because the example does not include stride coefficients greater than one nor does it include axis alignment. In this case arrays Y and Z can be replicated so that communication is reduced to zero. The resulting alignment directives are shown below (note the use of $T[i,*]$ and of $T[*,j]$ to indicate row i and column j, respectively). These results are illustrated in Figure 2.16 for arrays Y and Z.

$$\texttt{ALIGN } X[i,j] \texttt{ WITH } T[i,j]$$
$$\texttt{ALIGN } Y[i] \texttt{ WITH } T[i,*]$$
$$\texttt{ALIGN } Z[j] \texttt{ WITH } T[*,j]$$
$$\texttt{ALIGN } i,j \texttt{ WITH } T[i,j].$$

This means that array Y is replicated along the columns of X and array Z is replicated along the rows of X.

2.5 Solutions Using Linear Programming

The solution to an optimization problem such as those in this chapter which we represented as general linear programming problems are found at the extreme points or vertices as stated in the corner principle of algebra. These extreme points are the intersections of linear equations which are formulated using integer coefficients. The solutions are therefore guaranteed to be rational numbers which we can convert into whole numbers by multiplying by the corresponding least common multiple.

2.6 Comparison With Other Work

The component alignment problem has been proven to be NP-complete by Li and Chen [55]. They introduced the Component Affinity Graph (CAG) and the idea of generating communication primitives based on the reference patterns found in the array subscript expressions. The CAG is an undirected, weighted graph whose nodes represent the components to be aligned. The nodes are grouped in columns. An edge joins two nodes if the nodes have affinity. The algorithm provided by Li and Chen is based on heuristics.

Gupta and Banerjee [33], present a method restricted to partitioning of arrays, i.e. no computation partitioning. Their method selects important segments of code to determine distribution of various arrays based on some constraints. Quality measures are used to choose among contradicting constraints. These quality measures may require user intervention. The compiler *tries* to combine constraints for each array in a consistent

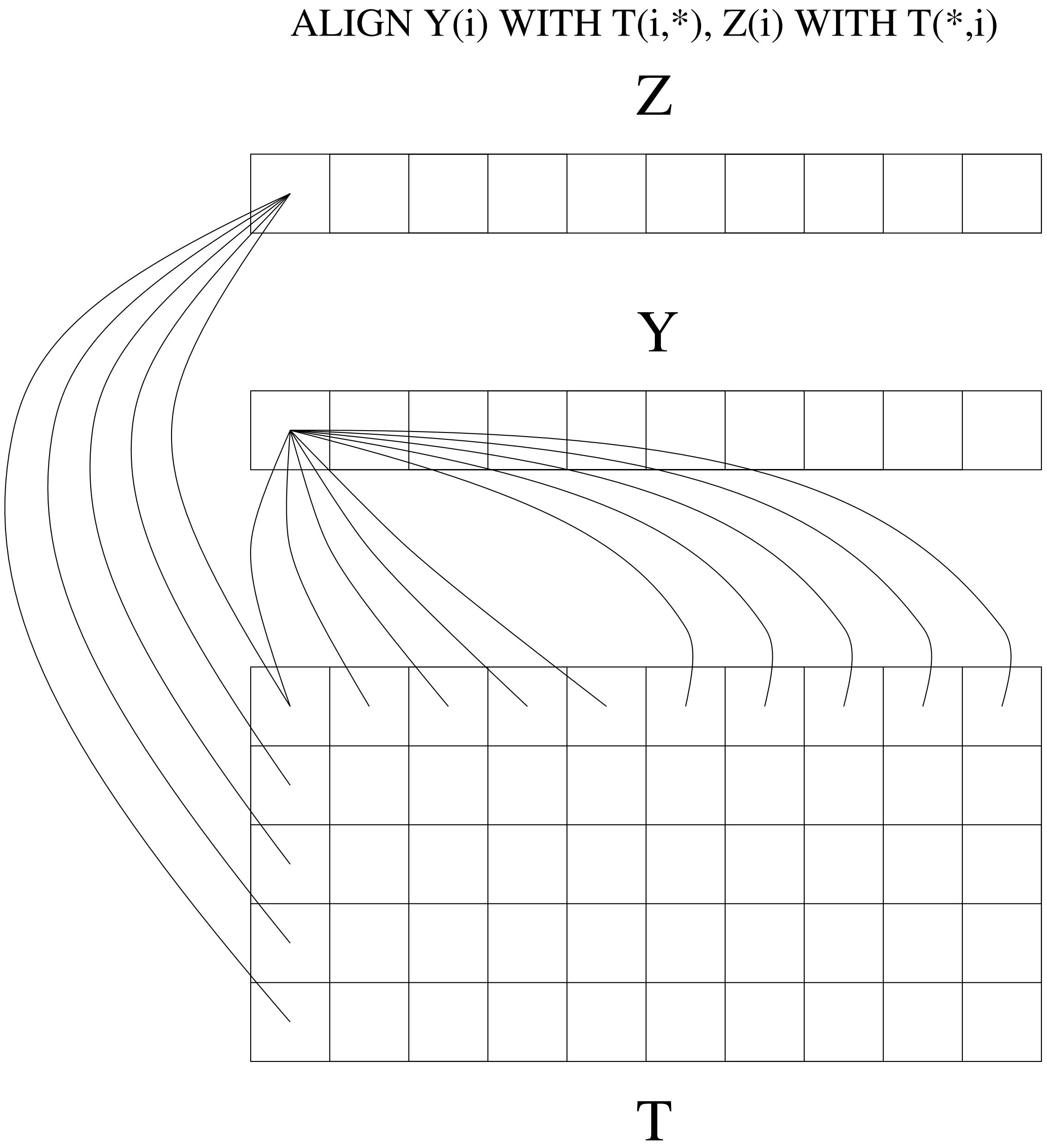


Figure 2.16: Replication of Array Z along the Rows of Template T and of Array Y along the Columns of Template T.

manner to minimize overall execution time and the entire program is considered. Small arrays are assumed to be replicated on all processors. The distribution of arrays is by rows, columns, or blocks. This work uses heuristic algorithms to determine the alignment of dimensions, i.e. component alignment, of various arrays since the problem has been shown to be NP-complete. The owner-computes rule is assumed and issues concerning the best way to communicate messages among processors, such as aggregate communication introduced in the work by Tseng [80], are dealt with. Communication costs are determined by Gupta and Banerjee [33] after identifying the pairs of dimensions that should be aligned. Consideration is given to when it would be best to replicate a dimension rather that to distribute it. The component affinity graph is used to determine alignment.

Bau et al. [14] use elementary matrix methods to determine communication-free alignment of code and data. They also deal with the problem of replicating read-only data to eliminate communication. Their work incorporates data dependences in their proposed solution to the problem and the owner-computes rule is assumed. Replication of data is also incorporated into their proposed solution. Their work treats all types of alignment as equal, e.g. it does not incorporate the notion that axis alignment is more important than stride, replication, or offset alignment. The Smith normal form of integers is used as part of their method. Alignment of both data and computation is determined by finding a basis for the null space of a known matrix.

Chatterjee et al. [23] and [24] provide an algorithm that obtains alignments which are more general that the owner-computes rule by decomposing alignment functions into several components. Chatterjee [23] et al. investigate the problem of evaluating Fortran 90 style array expressions on massively parallel distributed-memory machines. They present algorithms based on dynamic programming.

Kim and Wolfe [50] show how to find and operate on the communication *pattern matrix* from user-aligned references.

O'Boyle [61] proposed an automatic data partition algorithm based on the analysis of four distinct factors. He does not consider partitioning of computation along with that of data and he is not concerned with finding the alignment that will minimize communication.

Wakatani and Wolfe [81] address the problem of minimizing communication overhead but from a different context than ours. They are concerned with the communication arising from the redistribution of an array and proposed a technique called *strip mining redistribution*. They are not concerned with automatically generating the alignments in order.

Chatterjee et al. [20] and Sheffer et al. [76] deal with determining both static and dynamic distributions. They use the Alignment-Distribution Graph (ADG) whose nodes represent program operations, the ports in the nodes represent array object manipulated by the program, and the edges connect array definitions to their respective uses. The ADG is a directed edge-weighed graph although it is used as an undirected graph. Communication occurs when the alignment or distribution at the end points of an edge is different. At this time realignment and or redistribution may be needed. The completion time of a program is modeled as the sum of the cost over all the nodes (which accounts for computation and realignment) plus the sum over all the edges of the redistribution time (which takes into account the cost per data item of all-to-all personalized communication, the total data volume, and the discrete distance between distributions).

Kremer [52] proves the dynamic remapping problem NP-complete. Kremer et al. [53] and Kremer [51] consider the profitability of dynamic remapping and use an interactive tool for automatic data layout, respectively.

Kennedy and Kremer [48, 49] deal with dynamic remapping in Fortran D [80] and HPF [41]. The work by Kennedy and Kremer propose a way to solve the NP-complete inter-dimensional alignment problem [52] using a state-of-the-art general purpose integer programming solver [49]. Thus Kennedy and Kremer [49] formulate the inter-dimensional alignment problem as a 0-1 integer programming problem. The same is done by Bixby et al. [17].

2.7 Chapter Summary

We have modeled the alignment problem as a general linear programming problem using the l_1 metric. Constraints on the values of the stride ($\alpha' s$) and offset ($\beta' s$) coefficients were included as part of the model. Among other things these constraints reduce the contribution of single referenced arrays to zero and also make the problem independent of the loop index variable. An LP tool was used to solve the problem. A solution was presented for the offset alignment problem and specific solutions were given for the Jacobi, Alternating-Direction- Implicit, Disper, Livermore 18, Livermore 23, Red Black SOR, and Shallow benchmarks. Solutions were provided also for stride and axis alignment along with replication alignment. For axis alignment we have required that the an array diagonal be mapped to a cell template in order to solve the problem. Our method will determine a non-trivial communication-free solution if it exists. Otherwise, our method will determine a solution that minimizes communication.

Since the problem is modeled using linear constraints with integer coefficients and since the solution to this type of problems lies in one of the extreme points resulting from the intersection of the linear constraints, the results are guaranteed to be rational numbers. Multiplying by the least common multiple of the denominators will yield whole numbers. Linear programming tools are readily available for most computers. The methods provided here are intended to be used as part of the compiler. Our framework does not require user intervention.

CHAPTER 3

USING LAGRANGE MULTIPLIERS TO SOLVE THE ALIGNMENT PROBLEM

In this chapter we show how to solve the computation and data alignment problems using the Euclidean metric and the Lagrange Multiplier method. Once the problem has been modeled, we use the software Mathematica [90] to solve the constrained optimization problem. This method allows us to specify both equality and inequality constraints whenever necessary. We deal with the alignment problem in a uniform way, that is we formulate the stride, offset, axis, and replication alignment problems using the same constraint-based method and solve them using the Lagrange Multiplier method. Some researches have focused on the communication-free solution to the alignment problem and provide no way of dealing with the problem when no communication-free solution exists. Our approach deals with the problem when communication-free solutions exist and also when no non-trivial communication-free solution can be found. If communication-free alignment is possible our method will find it. Otherwise, our method will determine the alignment that minimizes communication in case the communication is unavoidable. In order to determine is the solution is communication-free one can simply substitute the values found into the distance function and check whether this becomes zero. Our approach is intended as a tool to be used with the compiler.

This chapter is organized as follows. Section 3.1 shows how to use the Lagrange Multiplier method for the stride and reversal alignment problems. In Section 3.2 we use the Lagrange Multiplier method for solving the offset alignment problem. Section 3.2 also presents the formulation and present solutions for several benchmark codes. In Section 3.3 we provide a solution to the axis alignment problem. Section 3.4 presents how to use this approach for replication alignment. Section 3.5 compares what other researchers have done. Finally, Section 3.6 summarizes the chapter.

3.1 STRIDE AND REVERSAL ALIGNMENT

When the coefficients of the loop index variables in the subscript expressions of the array references in a program are greater than unity, we have what has been termed as stride alignment. If any of these coefficients is negative, then it is called reversal alignment. Reversal alignment corresponds to mapping the reflection of the array onto the template [80]. Stride alignment is generated by statements similar to the following

$$\texttt{ALIGN}\ X[i]\ \texttt{WITH}\ T1[2i+3].$$

A multidimensional array example is given in the statement

$$\texttt{ALIGN}\ Y[i,j]\ \texttt{WITH}\ T2[2i+1,3j+2].$$

An example of reversal alignment is shown in the following statement

$$\texttt{ALIGN}\ Y[i]\ \texttt{WITH}\ T3[-i].$$

Consider the following piece of code:

DO $i = 1, N$
$\quad X[a_1 i + b_1] = Y[c_1 i + d_1] + Y[c_2 i + d_2] + \cdots + Y[c_r i + d_r]$
ENDDO

where arrays X and Y (which is not replicated) will be aligned to a template T as shown below

$\texttt{ALIGN}\ X[i]\ \texttt{WITH}\ T[\alpha_X i + \beta_X]$
$\texttt{ALIGN}\ Y[i]\ \texttt{WITH}\ T[\alpha_Y i + \beta_Y]$

and the iterations of the loop will be aligned with the following declaration

$\texttt{ALIGN}\ i\ \texttt{WITH}\ T[\alpha_I i + \beta_I]$

where α_X, β_X, α_Y, β_Y, α_I, and β_I are to be determined. Whenever any coefficient of the loop index variables in the assignment statement are greater than one our formulation will yield values for the α's which will be greater than or equal to one.

Let us consider for now iteration i only. We want to minimize the distance from the processor(s) holding the elements of arrays X and Y that are needed to perform the computation of the element on the left hand side to the processor which will be performing the computation during iteration i. Using the alignment as specified above we can find that the processor which holds the element on the *lhs* is processor $\alpha_X(a_1 i + b_1) + \beta_X$. Similarly, the processor holding the first term of array Y is processor $\alpha_Y(c_1 i + d_1) + \beta_Y$, the one holding the second term is $\alpha_Y(c_2 i + d_2) + \beta_Y$, and so on. Using the l_2 or Euclidean metric the distance from the processor which holds the *lhs* element to the processor which performs the computation during iteration i is $\left[([\alpha_X(a_1 i + b_1) + \beta_X] - [\alpha_I i + \beta_I])^2\right]^{\frac{1}{2}}$. Similarly we find the distance from the processor(s) holding each one of the elements on the right hand side is $\left[([\alpha_Y(c_j i + d_j) + \beta_Y] - [\alpha_I i + \beta_I])^2\right]^{\frac{1}{2}}$, $1 \leq j \leq r$.

Combining all the terms shown in the last two equations above we find the sum of the distances from each processor holding an element of X and each processor holding and element of Y to the processor which performs the computation during iteration i to be

$$\left[((\alpha_X(a_1 i + b_1) + \beta_X) - (\alpha_I i + \beta_I))^2 + \sum_{j=1}^{r} (\alpha_Y\,(c_j i + d_j) + \beta_Y - (\alpha_I i + \beta_I))^2\right]^{\frac{1}{2}}.$$

If we now consider all the iterations of the loop nest, then the equation above becomes

$$\text{distance} \quad = \quad \left[\sum_{i=1}^{N} ((\alpha_X(a_1 i + b_1) + \beta_X) - (\alpha_I i + \beta_I))^2\right.$$

$$+ \quad \left.\sum_{j=1}^{r}\sum_{i=1}^{N}\left(\alpha_Y\left(c_j i + d_j\right) + \beta_Y - \left(\alpha_I i + \beta_I\right)\right)^2\right]^{\frac{1}{2}}. \tag{3.1}$$

Collecting terms and rearranging the above equation we obtain the following

$$\begin{aligned} \text{distance} &= \left[\sum_{i=1}^{N}\left((\alpha_X a_1 - \alpha_I)i + \alpha_X b_1 + \beta_X - \beta_I\right)^2\right. \\ &+ \left.\sum_{j=1}^{r}\sum_{i=1}^{N}\left((\alpha_Y c_j - \alpha_I)i + \alpha_Y d_j + \beta_Y - \beta_I\right)^2\right]^{\frac{1}{2}}. \end{aligned} \tag{3.2}$$

To find an expression for the distance over the entire program we generalize the expressions above to the case when we have an arbitrary number l of loop nests, an arbitrary number w of statements over the various loop nests (w^g is the number of statements in loop nest g), and q is the total number of arrays in the program which are actually used. Note that we do not account for arrays which have been declared but are not used. In other words

$$\begin{aligned} \text{distance} &= \left[\sum_{g=1}^{l}\sum_{u=1}^{w^g}\left\{\sum_{k=1}^{q}\sum_{j=1}^{r_k}\sum_{i=1}^{N}\left(\left(\alpha_{Y_k} c_{jY_k}^{g,u} - \alpha_I\right)i + \alpha_{Y_k} d_{jY_k}^{g,u} + \beta_{Y_k} - \beta_I\right)^2\right.\right. \\ &+ \left.\left.\sum_{i=1}^{N}\left((\alpha_X a^{g,u} - \alpha_I)\,i + \alpha_X b^{g,u} + \beta_X - \beta_I\right)^2\right\}\right]^{\frac{1}{2}} \end{aligned} \tag{3.3}$$

where $r_k \geq 0$ represents the number of terms that involve array Y_k appearing on the right hand side of statement u in loop nest g, and $q \geq 1$. Note that in Equation 3.3, X is used for the array which appears on the *lhs* of a statement u in loop nest g, and Y_k is used for the k^{th} occurrence of an array Y which appears on the *rhs* of statement u in loop nest g, including X.

Adopting the convention that Y_1 corresponds to the array on the lhs of statement u in loop nest g, and accounting for the term Y_1 in r_1, we rewrite Equation 3.3 as shown below.

$$\text{distance} = \left[\sum_{g=1}^{l}\sum_{u=1}^{w^g}\sum_{k=1}^{q}\sum_{j=1}^{r_k}\sum_{i=1}^{N}\left(\left(\alpha_{Y_k} c_{jY_k}^{g,u} - \alpha_I\right)i + \alpha_{Y_k} d_{jY_k}^{g,u} + \beta_{Y_k} - \beta_I\right)^2\right]^{\frac{1}{2}} \tag{3.4}$$

Note that for any array Y_k for which $r_k = 1$ we can reduce its contribution to the above equation to zero by choosing

$$\alpha_I = \alpha_{Y_k} c_{1_{Y_k}}$$

and

$$\beta_I = \beta_{Y_k} + \alpha_{Y_k} b_{1_{Y_k}}.$$

This will also be the case if $r_k > 1$ and the subscript expressions for array Y_k are always the same. In this case we use these equations as constraints on the values of both α_I and β_I and we also use it to impose constraints on the values of α_{Y_k} and β_{Y_k}.

In order to solve for the unknowns we require that $\alpha_{Y_k} c^{g,u}_{j_{Y_k}} - \alpha^{g,u}_I = 0$. In this way we eliminate the terms that are multiplied by i in Equation 3.4 and arrive at the following equation

$$\text{distance} = \left[\sum_{g=1}^{l} \sum_{u=1}^{w^g} \sum_{k=1}^{q} \sum_{j=1}^{r_k} \sum_{i=1}^{N} \left(\alpha_{Y_k} d^{g,u}_{j_{Y_k}} + \beta_{Y_k} - \beta_I \right)^2 \right]^{\frac{1}{2}}. \quad (3.5)$$

To solve for the unknowns in Equation 3.5 we will use the Lagrange Multiplier method as reported by Avriel [7], Bazaraa et al. [15], Bertsekas [16], Kuhn and Tucker [54], Pike [65], and Reklaitis et al. [70]. To minimize the Euclidean distance function in Equation 3.5 subject to the conditions $f_i(x) \leq 0, i = 1, 2, \cdots, h$, and $f_i(x) = 0, i = h+1, h+2, \cdots, m$, the necessary conditions for the existence of a relative minimum at a point x^* are:

1. $\frac{\partial L}{\partial x_j}(x^*) + \sum\limits_{i=1}^{h} \lambda_i \frac{\partial L}{\partial x_j}(x^*) + \sum\limits_{i=h+1}^{m} \lambda_i \frac{\partial L}{\partial x_j}(x^*) = 0 \quad \text{for } j = 1, 2, \cdots, n$
2. $f_i(x^*) \leq 0 \quad \text{for } i = 1, 2, \cdots, h$
3. $f_i(x^*) = 0 \quad \text{for } i = h+1, h+2, \cdots, m$
4. $\lambda_i f_i(x^*) = 0 \quad \text{for } i = 1, 2, \cdots, h$
5. $\lambda_i \geq 0 \quad \text{for } i = 1, 2, \cdots, h$
6. λ_i is unrestricted in sign for $i = h+1, h+2, \cdots, m$

where n is the number of unknowns, h is the number of inequality constraints, m ($n \geq m$) is the total number of constraints including equality constraints, and L is the Lagrangian function formed by adding the constraints and their corresponding multipliers (λ's) to the distance function [65]. Note that if we do not want to allow a variable to be negative we can specify this by adding the appropriate constraint. For example, if we do not want to allow reversal alignment we can constrain the values on the α's to be positive.

The first condition sets the first partial derivatives of the Lagrangian function L with respect to x_i, $i = 1, 2, \cdots, n$ equal to zero to locate the Kuhn-Tucker point x^*. Conditions 2 and 3 are the inequality and equality constraints, respectively, that must be met at the minimum point found by solving the system of equations obtained from Condition 1. The fourth condition comes from setting the partial derivatives of the Lagrangian with respect to the slack variables equal to zero. Condition 5 arises from the fact that the rate of change of the distance function with respect to the parameters on the *rhs* of the constraints is equal to the negative of the corresponding Lagrange multiplier. By increasing the *rhs* of a constraint the constraint region would be enlarged, which could not result in a larger value for the distance function evaluated at x^* but could result in a lower value. Thus the Lagrange multiplier must be positive to satisfy the rate of change mentioned above [65, 70]. Condition 6 is due to a proof that the Lagrange multipliers associated with the equality constraints are not restricted in sign [65]. Note that a new variable is added for each equality constraint and that two variables are added for each inequality constraint.

To illustrate the Lagrange Multiplier method applied to the stride alignment problem we will use the code shown below which is the same code we used in a previous example.

DO $i = 1, N$
$X[2i-1] = Y[3i-1] + Y[3i] + Y[3i+1]$
ENDDO

The Lagrangian function in this case is given by

$$\begin{aligned} L &= \left(N\left((\beta_X - \alpha_X - \beta_I)^2 + (\beta_Y - \beta_I)^2 + (\beta_Y - \alpha_Y - \beta_I)^2 + (\beta_Y + \alpha_Y - \beta_I)^2\right)\right)^{\frac{1}{2}} \\ &+ \lambda_1\left(1 + s^2 - \alpha_I\right) + \lambda_2\left(2\alpha_X - \alpha_I\right) + \lambda_3\left(3\alpha_Y - \alpha_I\right) \end{aligned} \tag{3.6}$$

and the corresponding system of equations is

1. $\frac{\partial L}{\partial \alpha_I} = \lambda_1 + \lambda_2 + \lambda_3 = 0$

2. $\frac{\partial L}{\partial \alpha_X} = \frac{\sqrt{N}(-1(-\alpha_X - \beta_I + \beta_X))}{((-\alpha_X-\beta_I+\beta_X)^2+(-\beta_I+\beta_Y)^2+(-\alpha_Y-\beta_I+\beta_Y)^2+(\alpha_Y-\beta_I+\beta_Y)^2)^{\frac{1}{2}}} + 2\lambda_2 = 0$

3. $\frac{\partial L}{\partial \alpha_Y} = \frac{\sqrt{N}(0.5(-2(-\alpha_Y-\beta_I+\beta_Y)+2(\alpha_Y-\beta_I+\beta_Y))}{((-\alpha_X-\beta_I+\beta_X)^2+(-\beta_I+\beta_Y)^2+(-\alpha_Y-\beta_I+\beta_Y)^2+(\alpha_Y-\beta_I+\beta_Y)^2)^{\frac{1}{2}}} + 3\lambda_3 = 0$

4. $\frac{\partial L}{\partial \beta_I} = \frac{(0.5(-2(-\alpha_X-\beta_I+\beta_X)-2(\beta_I+\beta_Y)-2(\alpha_Y-\beta_I+\beta_Y))}{((-\alpha_X-\beta_I+\beta_X)^2+(-\beta_I+\beta_Y)^2+(-\alpha_Y-\beta_I+\beta_Y)^2+(\alpha_Y-\beta_I+\beta_Y)^2)^{\frac{1}{2}}} = 0$

5. $\frac{\partial L}{\partial \beta_X} = \frac{(-\alpha_X-\beta_I+\beta_X)}{((-\alpha_X-\beta_I+\beta_X)^2+(-\beta_I+\beta_Y)^2+(-\alpha_Y-\beta_I+\beta_Y)^2+(\alpha_Y-\beta_I+\beta_Y)^2)^{\frac{1}{2}}} = 0$

6. $\frac{\partial L}{\partial \beta_Y} = \frac{(0.5(2(-\beta_I+\beta_Y)+2(-\alpha_Y-\beta_I+\beta_Y)+2(\alpha_Y-\beta_I+\beta_Y)}{((-\alpha_X-\beta_I+\beta_X)^2+(-\beta_I+\beta_Y)^2+(-\alpha_Y-\beta_I+\beta_Y)^2+(\alpha_Y-\beta_I+\beta_Y)^2)^{\frac{1}{2}}} = 0$

7. $\frac{\partial L}{\partial \lambda_1} = 1 - \alpha_I + s^2 = 0$

8. $\frac{\partial L}{\partial \lambda_2} = -\alpha_I + 2\alpha_X = 0$

9. $\frac{\partial L}{\partial \lambda_2} = -\alpha_I + 3\alpha_Y = 0$

10. $\frac{\partial L}{\partial s} = \lambda_1 s = 0$

The solutions that we obtain are $\lambda_1 = 0.47\sqrt{N}$, $\lambda_2 = 0$, $\lambda_3 = -0.47\sqrt{N}$, $s = 0$, $\alpha_I = 1$, $\alpha_Y = 1/3$, $\alpha_X = 1/2$, $\beta_Y = \beta_I$, and $\beta_X = \beta_I + 1/2$. Note that these are rational numbers. In order to convert these values to integer values we multiply by the least common multiple of the denominators; note that an integer i can be written as

$\frac{i}{1}$. Since $\lambda_1 \neq 0$, and $s = 0$ the equality holds. Note that for this example we had added two equality constraints and one inequality constraint to the Euclidean distance function in order to form the Lagrangian function. Two variables, the Lagrange multiplier and the slack variable, are added for each inequality constraint and one variable, the Lagrange multiplier, for each equality constraint. The total time that takes an IBM RISC System/6000 to solve this system of equations is 3.6 seconds. We should point out that, for this example, we obtained several sets of possible solutions which makes for the excess time. This is the computer we used for all the examples given in this chapter.

Let $\alpha_I = 6$, $\alpha_X = 3$, $\alpha_Y = 2$, $\beta_I = 0$, and $\beta_X = 3$. With this alignment, each processor computing an iteration will need an element of Y which is held by the processor to its left, assuming a linear array and a block distribution with block size equal to 3, except of course for the processor at the leftmost position (see Figure 2.3). Note that the communication is with only one processor and that only one element needs to be passed from the sending processor to the receiving processor.

Using the owner-computes rule we obtain $\beta_X = 1$ for the same values of $\alpha_Y = 2$, $\alpha_X = 3$, and $\beta_Y = 1$. This would require each processor to send two elements to the processor on its right, except for the last processor, assuming the same configuration and distribution as before, see Figure 2.2. Again note that the communication is with only one processor, but this time two messages must be communicated.

3.2 OFFSET ALIGNMENT

Offset alignment can be viewed as a special case of stride alignment where the stride coefficients are equal to one. Consider the following piece of code:

ALIGN $X[i]$ WITH $T[i + \beta_X]$
ALIGN $Y[i]$ WITH $T[i + \beta_Y]$
ALIGN i WITH $T[i + \beta_I]$
DO $i = 1, N$
    $X[i + b_1] = Y[i + d_1] + Y[i + d_2] + \cdots + Y[i + d_r]$
ENDDO

where β_X, β_Y, and β_I are to be determined. Consider iteration i. We want to minimize the distance from the processor(s) holding the elements of arrays X and Y that are needed to perform the computation of the element on the left hand side to the processor which will be performing the computation during iteration i. Using the alignment specified above we find that the processor which holds the element on the *lhs* is processor $i + b_1 + \beta_X$. Similarly, the processor holding the first term of array Y is processor $i + d_1 + \beta_Y$, the one holding the second term is $i + d_2 + \beta_Y$, and so on.

The distance function is

$$\left[\sum_{g=1}^{l} \sum_{u=1}^{w^g} \sum_{k=1}^{q} \sum_{j=1}^{r_k} \sum_{i=1}^{N} \left(d_{jY_k}^{g,u} + \beta_{Y_k} - \beta_I \right)^2 \right]^{\frac{1}{2}} \tag{3.7}$$

where l is the number of loops, w^g is the number of statements in loop nest g, q is the total number of arrays, and Y_1 corresponds to the array on the lhs of statement u in loop nest g.

To solve for the unknowns in Equation 3.7 we will use the Lagrange Multiplier method. Though this method allows us to include in our system of equations a set of equality and inequality constraints, we do not need to do it for the offset alignment case. To illustrate this method let us consider the following code segment:

DO $i = 1, N$
    $X[i] = Y[i+2]$
ENDDO.

The first step is to form the function L,

$$L = ((\beta_X - \beta_I)^2 + (2 + \beta_Y - \beta_I)^2)^{\frac{1}{2}}.$$

Here we repeat the necessary conditions for the existence of a relative minimum at a point x^* for the Euclidean distance function shown in Equation 3.7 and subject to the conditions $f_i(x) \leq 0, i = 1, 2, \cdots, h$, and $f_i(x) = 0, i = h+1, h+2, \cdots, m$, where n is the number of unknowns, h is the number of inequality constraints, and m $(n \geq m)$ is the total number of constraints including equality constraints [65]. These are the same conditions introduced and explained in Section 3.1.

1. $\frac{\partial L}{\partial x_j}(x^*) + \sum_{i=1}^{h} \lambda_i \frac{\partial L}{\partial x_j}(x^*) + \sum_{i=h+1}^{m} \lambda_i \frac{\partial L}{\partial x_j}(x^*) = 0 \quad$ for $j = 1, 2, \cdots, n$
2. $f_i(x^*) \leq 0 \quad$ for $i = 1, 2, \cdots, h$
3. $f_i(x^*) = 0 \quad$ for $i = h+1, h+2, \cdots, m$
4. $\lambda_i f_i(x^*) = 0 \quad$ for $i = 1, 2, \cdots, h$
5. $\lambda_i \geq 0 \quad$ for $i = 1, 2, \cdots, h$
6. λ_i is unrestricted in sign for $i = h+1, h+2, \cdots, m$

By solving the system of equations given by condition 1 above, the general solution for our example would be given by $\beta_X = \beta_I$, and $\beta_Y = \beta_I - 2$.

We now use the method developed in this section to find the alignment for several benchmark programs, i.e. Jacobi, Alternating-Direction-Implicit (ADI), Disper, Livermore 18, Livermore 23, Red Black SOR, and Shallow.

The code for Jacobi was presented in Figure 2.6. After applying our method to each one of the statements we obtained the results shown in Table 3.1.

The results of applying this method to the ADI program segment shown in Figure 2.7 are shown in Table 3.2.

The code for Disper is shown in Figure 2.8 and the results after applying our method are shown in Table 3.3

Table 3.1: Constant Offsets (β's) Found Using Lagrange Method on Jacobi.

STATE-MENT	ARRAY NAME	DIMENSION 1ST	DIMENSION 2ND
S1	I	0	0
	A	0	0
	B	0	0
S2	I	0	0
	A	0	0
	B	0	0

Table 3.2: Constant Offsets (β's) Found Using Lagrange Method on ADI.

STATE-MENT	ARRAY NAME	DIMENSION 1ST	DIMENSION 2ND	STATE-MENT	ARRAY NAME	DIMENSION 1ST	DIMENSION 2ND
S1	I	0	0	S5	I	0	0
	A	0	0		A	0	0
	B	0	1		B	1	0
	X	0	1/3		X	1/3	0
S2	I	0	0	S6	I	0	0
	A	0	0		A	0	0
	B	0	1/3		B	1/3	0
S3	I	0	-	S7	I	-	0
	B	0	-		B	-	0
	X	0	-		X	-	0
S4	I	0	0	S8	I	0	0
	A	0	-1		A	-1	0
	B	0	0		B	0	0
	X	0	-1/3		X	-1/3	0

Table 3.3: Constant Offsets (β's) Found Using Lagrange Method on Disper.

STATE-MENT	ARRAY NAME	DIMENSION 1ST	DIMENSION 2ND	DIMENSION 3RD	DIMENSION 4TH	DIMENSION 5TH
S3	I	0	0	0	0	0
	GRADY	0	-	-	-	-
	PFMR	0	0	0	0	0
	DDY	0	0	0	-	-

Table 3.4: Constant Offsets (β's) Found Using Lagrange Method on Livermore 18.

STATE-MENT	ARRAY	DIMENSION ONE	DIMENSION TWO	STATE-MENT	ARRAY	DIMENSION ONE	DIMENSION TWO
S1	I	-1	0	S3	I	0	0
	ZA	-1/2	0		ZA	0	0
	ZB	0	0		ZB	0	0
	ZM	0	0		ZU	0	0
	ZP	0	0		ZV	0	0
	ZQ	0	0		ZZ	0	0
	ZR	0	0	S4	I	0	0
	ZU	0	0		ZA	0	0
	ZV	0	0		ZB	0	0
	ZZ	0	0		ZU	0	0
S2	I	-1	0		ZV	0	0
	ZA	0	0		ZR	0	0
	ZB	-1/2	0	S5	I	0	0
	ZM	0	0		ZR	0	0
	ZP	0	0		ZU	0	0
	ZQ	0	0	S6	I	0	0
	ZR	0	0		ZV	0	0
	ZU	0	0		ZZ	0	0
	ZV	0	0				
	ZZ	0	0				

A program segment for Livermore 18 is shown in Figure 2.9. The results of applying our methods are shown in Table 3.4. The program segment for Livermore 23 is shown in Figure 2.10 and the results of applying our method are shown in Table 3.5.

The code for Red Black SOR (Successive Over Relaxation) is shown in Figure 2.11 and the results of applying our method on Table 3.6.

Shallow is a 200 line benchmark that uses stencil computation that applies finite-difference methods to solve shallow-water equations and is a representative of a large class of existing supercomputer applications. Table 3.7 shows the result of applying our method to Shallow. The code for Shallow is shown in Figure 2.12.

With this we conclude the report on the benchmarks after applying the Lagrange Multipliers method. Table 3.8 summarizes the time it took our computer to solve the equations for the applications mentioned above.

3.3 Axis Alignment

As explained earlier, axis alignment arises when we have a dimension permutation in the alignment statement for multidimensional arrays. For example, in the statement

Table 3.5: Constant Offsets (β's) Found Using Lagrange Method on Livermore 23.

STATE-MENT	ARRAY NAME	DIMENSION 1ST	DIMENSION 2ND
S1	I	0	0
	ZA	0	0
S2	I	0	0
	ZA	0	0

Table 3.6: Constant Offsets (β's) Found Using Lagrange Method on Red Black SOR.

STATE-MENT	ARRAY NAME	DIMENSION 1ST	DIMENSION 2ND
S1	I	0	0
	V	0	0
S2	I	0	0
	V	0	0
S3	I	0	0
	V	0	0
S4	I	0	0
	V	0	0

Table 3.7: Constant Offsets (β's) Found Using Lagrange Method on Shallow.

STATE-MENT	ARRAY NAME	DIMENSION		STATE-MENT	ARRAY NAME	DIMENSION	
		1ST	2ND			1ST	2ND
S1	I	1	0		U	-1/2	0
	U	0	0		V	0	-1/2
	PSI	0	-1/2	S7	I	1	0
S2	I	0	1		UNEW	0	0
	V	0	0		UOLD	0	0
	PSI	-1/2	0		Z	0	-1/2
S3	I	1	0		CV	1/2	-1/2
	CU	0	0		H	1/2	0
	P	1/2	0	S8	I	0	1
	U	0	0		VNEW	0	0
S4	I	0	1		VOLD	0	0
	CV	0	0		Z	-1/2	0
	P	0	1/2		CU	-1/2	1/2
	V	0	0		H	0	1/2
S5	I	1	1	S9	I	0	0
	Z	0	0		PNEW	0	0
	V	1/2	0		POLD	0	0
	U	0	1/2		CU	-1/2	0
	P	1/2	1/2		CV	0	-1/2
S6	I	0	0				
	H	0	0				
	P	0	0				

Table 3.8: Time (seconds) to Solve the System of Equations for the Different Applications.

PROGRAM	TIME
Jacobi	0.18
ADI	1.67
Disper	0.23
Livermore 18	5.98
Livermore 23	0.12
Red Black SOR	0.24
Shallow	1.96

$$\texttt{ALIGN } X[i,j] \texttt{ WITH } T1[j,i]$$

we have that each row i of array X is aligned with column i of template $T1$. Likewise, each column j of X is aligned with row j of template $T1$. In other words, the first dimension of array X is aligned with the second dimension of template $T1$, and the second dimension of X is aligned with the first dimension of $T1$ (see Figure 2.13).

In what follows assume that arrays X and Y are two-dimensional arrays which are not replicated onto the available processors. Also assume that T is a two-dimensional template. Consider the following code segment:

ALIGN $X[i,j]$ WITH $T[\alpha_X^1 i + \beta_X^1, \alpha_X^2 j + \beta_X^2]$
ALIGN $Y[i,j]$ WITH $T[\alpha_Y^1 i + \beta_Y^1, \alpha_Y^2 j + \beta_Y^2]$
ALIGN i,j WITH $T[\alpha_I^1 i + \beta_I^1, \alpha_I^2 j + \beta_I^2]$
DO $i = 1, N$
  DO $j = 1, N$
    $X[i,j] = Y[i,j] + Y[j,i]$
  ENDDO
ENDDO

Using the l_2 metric the total distance from the processor performing iteration i,j to the processors holding each of the operands would be given by

$$\begin{aligned}
\text{distance} &= \Big[\Big(\Big(\alpha_X^1 - \alpha_I^1\Big) i + \beta_X^1 - \beta_I^1\Big)^2 + \Big(\Big(\alpha_X^2 - \alpha_I^2\Big) j + \beta_X^2 - \beta_I^2\Big)^2 \\
&+ \Big(\Big(\alpha_Y^1 - \alpha_I^1\Big) i + \beta_Y^1 - \beta_I^1\Big)^2 + \Big(\Big(\alpha_Y^2 - \alpha_I^2\Big) j + \beta_Y^2 - \beta_I^2\Big)^2 \\
&+ \Big(\alpha_Y^1 j - \alpha_I^1 i + \beta_Y^1 - \beta_I^1\Big)^2 + \Big(\alpha_Y^2 i - \alpha_I^2 j + \beta_Y^2 - \beta_I^2\Big)^2\Big]^{\frac{1}{2}}. \qquad (3.8)
\end{aligned}$$

Note the last two terms of the equation above. There is not a non-trivial way of eliminating the i and the j from the above equation when we consider all the possible values that i, and j can take on.

Here the problem is not the metric but the actual mapping. We have a mapping from a two-dimensional array space and a two-dimensional iteration space to a two-dimensional template space, and because of the nature of the problem itself, this mapping approach we have used so far is not very useful for this particular problem. However, consider the same problem but with a different mapping. In particular, consider that arrays X and Y, and iteration i,j will be aligned using

ALIGN $X[i,j]$ WITH $T[\alpha_X^1 i + \beta_X^1 + \alpha_X^2 j + \beta_X^2]$
ALIGN $Y[i,j]$ WITH $T[\alpha_Y^1 i + \beta_Y^1 + \alpha_Y^2 j + \beta_Y^2]$
ALIGN i,j WITH $T[\alpha_I^1 i + \beta_I^1 + \alpha_I^2 j + \beta_I^2]$

so that the two-dimensional array space and the two-dimensional iteration space are mapped onto a one-dimensional template. The distance function will then be given by

$$\text{distance} = \Big[\Big(\Big(\alpha_X^1 - \alpha_I^1\Big) i + \Big(\alpha_X^2 - \alpha_I^2\Big) j + \beta_X^1 - \beta_I^1 + \beta_X^2 - \beta_I^2\Big)^2$$

$$+ \quad \left(\left(\alpha_Y^1 - \alpha_I^1\right) i + \left(\alpha_Y^2 - \alpha_I^2\right) j + \beta_Y^1 - \beta_I^1 + \beta_Y^2 - \beta_I^2\right)^2$$
$$+ \quad \left(\left(\alpha_Y^2 - \alpha_I^1\right) i + \left(\alpha_Y^1 - \alpha_I^2\right) j + \beta_Y^1 - \beta_I^1 + \beta_Y^2 - \beta_I^2\right)^2\Bigg]^{\frac{1}{2}}. \quad (3.9)$$

We require that

$$\alpha_X^1 = \alpha_I^1 = \alpha_X^2 = \alpha_I^2 = \alpha_Y^1 = \alpha_Y^2$$

and thus the distance function reduces to

$$\left[\left(\beta_X^1 - \beta_I^1 + \beta_X^2 - \beta_I^2\right)^2 + \left(\beta_Y^1 - \beta_I^1 + \beta_Y^2 - \beta_I^2\right)^2 + \left(\beta_Y^1 - \beta_I^1 + \beta_Y^2 - \beta_I^2\right)^2\right]^{\frac{1}{2}} \quad (3.10)$$

which is reduced to zero distance by allowing $\beta_X^1 = \beta_I^1 + \beta_I^2 - \beta_X^2$ and $\beta_Y^1 = \beta_I^1 + \beta_I^2 - \beta_Y^2$ which is the result we obtain when using our method. This means that each diagonal of arrays X and Y, and each diagonal of the iteration space would be mapped to a point in the one-dimensional template. The resulting alignment is illustrated in Figure 2.14. It took the computer 0.11 seconds to solve the system of equations and arrive at the result presented above.

3.4 REPLICATION

As explained in a previous chapter, replication is a type of alignment which arises from the need for every processor to store copies of some data. In this section we will use the example from Bau et al. [14] that we have used previously in Section 2.4. Consider the code shown below:

DO $i = 1, N$
  DO $j = 1, N$
S: $X[i,j] = X[i,j] + Y[i] * Z[j]$
  ENDDO
ENDDO.

Note that Y and Z are one-dimensional arrays and that X is a two-dimensional array. Also note that all the elements of Z that are accessed during an iteration of the outer loop are also accessed during the next iteration of the outer loop and that the same element of Y is accessed during all the iterations of the inner loop for a particular iteration of the outer loop. In other words, $Y[i]$ is used to compute all the elements in row i of X and $Z[j]$ is used to compute all the elements in column j of X. Lastly, note that elements of arrays Y and Z are referenced (read) but not modified (written) during each execution of statement S above.

Since $Y[i]$ is used to compute all the elements in row i of X and $Z[j]$ is used to compute all the elements in column j of X, we would like to align X, Y, and Z such that communication is minimized. In this case we can reduce the communication to zero by replicating arrays Y and Z along the columns and along

We find how to align the arrays using the Lagrange method developed in Section 3.2 for offset alignment. This is because the example does not include stride coefficients greater than one nor does it include axis alignment. In this case arrays Y and Z can be replicated so that communication is reduced to zero. Note that array Y is replicated along the columns of X and array Z is replicated along the rows of X. These results are illustrated in Figure 2.16 for arrays Y and Z. The resulting alignment is as shown in the following directives (note the use of $T[i,*]$ and of $T[*,j]$ to indicate row i and column j, respectively):

ALIGN $X[i,j]$ WITH $T[i,j]$
ALIGN $Y[i]$ WITH $T[i,*]$
ALIGN $Z[j]$ WITH $T[*,j]$
ALIGN i,j WITH $T[i,j]$.

3.5 Comparison With Other Work

The component alignment problem has been proven to be NP-complete by Li and Chen [55]. They introduced the Component Affinity Graph (CAG) and the idea of generating communication primitives based on the reference patterns found in the array subscript expressions. The CAG is an undirected, weighted graph whose nodes represent the components to be aligned. The nodes are grouped in columns. An edge joins two nodes if the nodes have affinity. The algorithm provided by Li and Chen is based on heuristics.

Gupta and Banerjee [33], present a method restricted to partitioning of arrays, i.e. no computation partitioning. Their method selects important segments of code to determine distribution of various arrays based on some constraints. Quality measures are used to choose among contradicting constraints. These quality measures may require user intervention. The compiler *tries* to combine constraints for each array in a consistent manner to minimize overall execution time and the entire program is considered. Small arrays are assumed to be replicated on all processors. The distribution of arrays is by rows, columns, or blocks. This work uses heuristic algorithms to determine the alignment of dimensions, i.e. component alignment, of various arrays since the problem has been shown to be NP-complete. The owner-computes rule is assumed and issues concerning the best way to communicate messages among processors, such as aggregate communication introduced in the work by Tseng [80], are dealt with. Communication costs are determined by Gupta and Banerjee [33] after identifying the pairs of dimensions that should be aligned. Consideration is given to when it would be best to replicate a dimension rather that to distribute it. The component affinity graph is used to determine alignment.

Bau et al. [14] use elementary matrix methods to determine communication-free alignment of code and data. They also deal with the problem of replicating read-only data to eliminate communication. Their work incorporates data dependences in their proposed solution to the problem and the owner-computes rule is assumed. Replication of data is also incorporated into their proposed solution. Their work treats all types of alignment as equal, e.g. it does not incorporate the notion that axis alignment is more important than stride, replication, or offset alignment. The Smith normal form of integers is used as part of their method. Alignment of both data and computation is determined by finding a basis for the null space of a known matrix.

Chatterjee et al. [23] and [24] provide an algorithm that obtains alignments which are more general that the owner-computes rule by decomposing alignment functions into several components. Chatterjee [23] et al. investigate the problem of evaluating Fortran 90 style array expressions on massively parallel distributed-memory machines. They present algorithms based on dynamic programming.

Kim and Wolfe [50] show how to find and operate on the communication *pattern matrix* from user-aligned references.

O'Boyle [61] proposed an automatic data partition algorithm based on the analysis of four distinct factors. He does not consider partitioning of computation along with that of data and he is not concerned with finding the alignment that will minimize communication.

Wakatani and Wolfe [81] address the problem of minimizing communication overhead but from a different context than ours. They are concerned with the communication arising from the redistribution of an array and proposed a technique called *strip mining redistribution*. They are not concerned with automatically generating the alignments in order.

Chatterjee et al. [20] and Sheffer et al. [76] deal with determining both static and dynamic distributions. They use the Alignment-Distribution Graph (ADG) whose nodes represent program operations, the ports in the nodes represent array object manipulated by the program, and the edges connect array definitions to their respective uses. The ADG is a directed edge-weighed graph although it is used as an undirected graph. Communication occurs when the alignment or distribution at the end points of an edge is different. The completion time of a program is modeled as the sum of the cost over all the nodes (which accounts for computation and realignment) plus the sum over all the edges of the redistribution time (which takes into account the cost per data item of all-to-all personalized communication, the total data volume, and the discrete distance between distributions).

Kremer [52] proves the dynamic remapping problem NP-complete. Kremer et al. [53] and Kremer [51] consider the profitability of dynamic remapping and use an interactive tool for automatic data layout, respectively.

Kennedy and Kremer [48, 49] deal with dynamic remapping in Fortran D [80] and HPF [41]. The work by Kennedy and Kremer propose a way to solve the NP-complete inter-dimensional alignment problem [52] using a state-of-the-art general purpose integer programming solver [49]. Thus Kennedy and Kremer [49] formulate the inter-dimensional alignment problem as a 0-1 integer programming problem. The same is done by Bixby et al. [17].

3.6 Chapter Summary

We have presented a method for solving the alignment problem by using the Lagrange Multipliers method on a constrained optimization problem modeled using the Euclidean metric. Both equality and inequality constraints can be added to our framework as needed in which case the Lagrange multipliers will also be part of the function. The constraints are on the values permitted for the stride coefficients ($\alpha' s$) and are such that the problem is independent of the loop index variable. The software Mathematica was used to solve the system of equations obtained from our model. Solutions were provided for stride, offset, axis, and replication alignment using the same constrained-based method. For axis alignment we have required that an entire diagonal from the arrays be mapped to a single template cell. If communication-free alignment is possible our method will find it. Otherwise, our method will determine the alignment that minimizes communication in case the communication is unavoidable.

This is the first time this method is used to solve the alignment problem. We have provided the results for the Jacobi, ADI, Disper, Livermore 18, Livermore 23, Red Black SOR, and Shallow benchmarks and the average time it took to find them is also included. Our framework does not require user intervention.

CHAPTER 4

RELAXING CONSTRAINTS IN THE ALIGNMENT PROBLEM

In this chapter we deal with the problem of determining which constraint or constraints to leave unsatisfied when we have an over-constrained system of equations from which we want to determine the alignment for both computation and data. The system of equations is obtained by using the framework by Bau et al. [14], which uses elementary linear algebra methods to determine a non-trivial communication-free solution to the computation and data alignment problem. This chapter is organized as follows: Section 4.1 is a review of the method presented by Bau et al. [14]. Section 4.2 presents our method of determining which constraint(s) to leave unsatisfied when the system is over-constrained. Section 4.3 reviews the work that other researchers have done and Section 4.4 is a summary of our findings in this chapter.

4.1 REVIEW OF BAU ET AL.'S METHOD

To illustrate the method developed by Bau et al. [14] we will use the following example:

DO $i = lb_i, ub_i$
  DO $j = lb_j, ub_j$
    $A[i,j] = f(A[i,j], B[i-1,j], B[i,j-1])$
  ENDDO
ENDDO.

From the subscript expressions for arrays A, and B we obtain the following, where F_A is the access matrix for array A, F_{B_1} and F_{B_2} are the access matrices for the first and second terms of array B, respectively, f_A, f_{B_1}, and f_{B_2} their corresponding constant terms, and C, D_A, and D_B are the mapping for the iterations (computations) and data elements of the arrays onto a virtual template, and $\hat{F_X}$ is formed in order to account for the constant offset term f_X in the same equations where the access matrix F_X is used. Note that this is done in order to account for the constant offset coefficients found in the array references. The resulting functions are called *affine* functions.

$$\hat{F}_A = \begin{bmatrix} F_A & f_A \\ 0 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix},$$

$$\hat{F}_{B_1} = \begin{bmatrix} F_{B_1} & f_{B_1} \\ 0 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & -1 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix},$$

$$\hat{F}_{B_2} = \left[\begin{array}{cc} F_{B_2} & f_{B_2} \\ 0 & 1 \end{array} \right] = \left[\begin{array}{ccc} 1 & 0 & 0 \\ 0 & 1 & -1 \\ 0 & 0 & 1 \end{array} \right],$$

$$\hat{C} = \hat{D}_A \hat{F}_A,\ \hat{C} = \hat{D}_B \hat{F}_{B_1},\ \hat{C} = \hat{D}_B \hat{F}_{B_2},$$

and

$$\hat{U} = \left[\begin{array}{ccc} \hat{C} & \hat{D}_A & \hat{D}_B \end{array} \right],$$

$$\hat{V} = \left[\begin{array}{cccc} I & I & I & I \\ -\hat{F}_A & -\hat{F}_A & 0 & 0 \\ 0 & 0 & -\hat{F}_{B_1} & -\hat{F}_{B_2} \end{array} \right]$$

i.e.

$$\hat{V} = \left[\begin{array}{cccccccccccc} 1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 \\ -1 & 0 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & -1 & 0 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & -1 & 0 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & -1 & 0 & 1 & -1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & -1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & -1 \end{array} \right].$$

It can be seen from the above that $\hat{V}^T \hat{U}^T = 0$ so that the problem is reduced to finding a basis for the null space of $\hat{V}^T$. An algorithm is then applied to $\hat{V}$ so that it is diagonalized into its Smith Normal Form, i.e. positive diagonal entries, to reveal its rank, e.g. $T = H\hat{V}G$, where H and G represent the elementary row and column operations, respectively. In other words:

$$T = H\hat{V}G = \left[\begin{array}{cc} \Sigma & 0 \\ 0 & 0 \end{array} \right].$$

Since H and G represent elementary operations they are unimodular matrices. In this dissertation a unimodular matrix is a matrix whose determinant is ± 1. This type of matrices are used extensively in parallelizing compilers. The only matrix which is important for the calculations is H since it is this matrix which we need to obtain the solution. This matrix H is premultiplied by a matrix $\hat{U}'$ to yield the solution matrix, i.e. , $\hat{U} = \hat{U}'H$ where matrix $\hat{U}'$ forms a basis for the range of the orthogonal complement of the first $r = \text{rank}(V)$ columns of matrix T:

$$\hat{U}' = \left[\begin{array}{ccccccccc} 0 & 0 & \cdots & 0 & 1 & 0 & \cdots & 0 \\ 0 & 0 & \cdots & 0 & 0 & 1 & \cdots & 0 \\ & & \vdots & & & & \vdots & \\ 0 & 0 & \cdots & 0 & 0 & 0 & \cdots & 1 \end{array} \right].$$

Note that the first r columns of $\hat{U}'$ are zero and that the last columns form an identity matrix. Premultiplying H by $\hat{U}'$ will choose those rows of H for which the corresponding rows of T are zero, i.e. the last $M-r$ rows, where M is the number of rows of $\hat{V}$, H, and T. Continuing with our example

$$\hat{U} = \begin{bmatrix} 1 & 1 & -1 & 1 & 1 & -1 & 1 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 \end{bmatrix}$$

for

$$H = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & -1 & 1 & 0 & -1 & 1 & 0 & -1 & 0 \\ 0 & 0 & 0 & -1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 \\ 1 & 1 & -1 & 1 & 1 & -1 & 1 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 \end{bmatrix}.$$

It is worth mentioning that we only need H out of all the matrices found when computing the Smith Normal Form since we find from $\hat{U}\hat{V} = 0$ that

$$\hat{U}H^{-1}H\hat{V}G = 0$$

or

$$\hat{U}' \begin{bmatrix} \Sigma & 0 \\ 0 & 0 \end{bmatrix} = 0$$

i.e.

$$\hat{U}' \begin{bmatrix} \Sigma \\ 0 \end{bmatrix} = 0$$

where $\hat{U}' = \hat{U}H^{-1}$.

Now, following the procedure by Bau et al. [14], in order to eliminate the extra dimension introduced when the constant (offset) terms were added, we need to find some $\beta_i \neq 0$ for which

$$\hat{U}^T\beta = \begin{bmatrix} 1 & 0 \\ 1 & 0 \\ -1 & 1 \\ 1 & 0 \\ 1 & 0 \\ -1 & 1 \\ 1 & 0 \\ 1 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} \beta_1 \\ \beta_2 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \\ 0 \\ 1 \\ 0 \\ 0 \\ 1 \end{bmatrix}$$

from which we obtain that $\beta_1 = 0$ and $\beta_2 = 1$. Since the second component of β is nonzero, we can eliminate the second row of $\hat{U}$. Thus

$$\hat{C} = \begin{bmatrix} 1 & 1 & -1 \\ 0 & 0 & 1 \end{bmatrix},\ \hat{D_A} = \begin{bmatrix} 1 & 1 & -1 \\ 0 & 0 & 1 \end{bmatrix},\ \hat{D_B} = \begin{bmatrix} 1 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix},$$

and

$$C = \begin{bmatrix} 1 & 1 & -1 \end{bmatrix},\ D_A = \begin{bmatrix} 1 & 1 & -1 \end{bmatrix},\ D_B = \begin{bmatrix} 1 & 1 & 0 \end{bmatrix},$$

or

$$C\vec{I} = \begin{bmatrix} 1 & 1 & -1 \end{bmatrix} \begin{bmatrix} i \\ j \\ 1 \end{bmatrix} = i + j - 1,$$

$$D_A\hat{F}_A\vec{I} = \begin{bmatrix} 1 & 1 & -1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} i \\ j \\ 1 \end{bmatrix} = i + j - 1,$$

$$D_B\hat{F}_{B_1}\vec{I} = \begin{bmatrix} 1 & 1 & 0 \end{bmatrix} \begin{bmatrix} 1 & 0 & -1 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} i \\ j \\ 1 \end{bmatrix} = i + j - 1$$

$$D_B\hat{F}_{B_1}\vec{I} = \begin{bmatrix} 1 & 1 & 0 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & -1 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} i \\ j \\ 1 \end{bmatrix} = i + j - 1.$$

This means that for a communication-free solution the iteration space and the data space for arrays A and B should be partitioned into diagonals as shown above.

4.2 Deciding on Which Constraint(s) to Leave Unsatisfied

When the system is over-constrained the only communication-free solution is the trivial solution, e.g. map each iteration and each data item to one processor [6, 14, 33, 45, 68]. Assume that we are using Bau et al.'s framework [14] which use elementary matrix methods to determine communication-free alignment of code and data. If we have an assignment statement such as

$$X[i,j] \;=\; \mathcal{F}(X[i,j], Y[i,j])$$

then, using Bau et al.'s method, we obtain

$$U = \begin{bmatrix} C & D_X & D_Y \end{bmatrix},$$

and

$$V = \begin{bmatrix} I & I \\ -F_X & 0 \\ 0 & -F_Y \end{bmatrix}.$$

In this case we obtain $C - D_X F_X = 0$ and $C - D_Y F_Y = 0$, i.e. $C = D_X F_X = D_Y F_Y \neq 0$. Otherwise, the only communication-free solution is to map both the computation and data to one processor so that $C = D_X F_X = D_Y F_Y = 0$. We could rewrite the equations from Bau et al. [14] in the following way where $\vec{I}$ represents the iteration vector

$$(C - D_X F_X)\vec{I} - D_X f_X = \mathcal{F}_1 \text{ and } (C - D_Y F_Y)\vec{I} - D_Y f_Y = \mathcal{F}_2.$$

We can think of the vectors $\mathcal{F}_1$ and $\mathcal{F}_2$ as estimates of the type of communication that will be incurred due to misalignment. In Bau et al.'s context for both a trivial and a non-trivial communication-free solution these functions are zero. If we look at them from another perspective, i.e. without fully applying Bau et al.'s [14] method, these functions could be thought of as including as part of their components, constant values, values that vary with some index variable, or values that change with more than one index variable depending on whether the misalignment is due to offset, stride, or axis misalignment, respectively.

If no non-trivial communication-free solution can be found we would then like the elements of vectors $\mathcal{F}_1$, and $\mathcal{F}_2$ to be constants and as many of them as possible to have a constant value of zero. In other words, if we can not obtain a non-trivial zero communication solution, we would then prefer to only have communication arising due to offset misalignment. This is the cheapest form of communication since it is usually between nearest neighbor processors. We should, in the other hand, avoid communication arising from axis misalignment because this type of communication indicates interprocessor communication among several processors along different dimensions, similar in some respect to a ManyToMany type of communication if we use Gupta and Banerjee's [33], and Garcia et al.'s [30] terminology.

With this in mind, when no non-trivial communication-free solution exists we would like to have the following $\mathcal{F}_1 =$ constant, $\mathcal{F}_2 =$ constant. For this to be possible we need $C = D_X F_X = D_Y F_Y$ to hold true so that the communication will be due solely to the constant offset terms used in the subscript expressions of the array references.

Thus, if it is not possible to find a solution that yields zero communication when using more than one processor, we should then attempt to find a solution that does not incur any interprocessor communication due to axis and stride misalignments, if at all possible, even if it means that we should have some communication due to offset misalignment. We need a structure that will have enough information to aid us in recognizing, among other things, which type of misalignment we could get if we do not satisfy a particular constraint. This structure should also give us information about the constant offset coefficients and possibly the distance vectors involved. The structure that we propose to use is what we call the *Reference Information Table* or RIT.

For example, assume that we have the following assignment statement

$$A[i,j] = \mathcal{F}(A[i,j], B[i,j], B[i-1,j-1], B[i-2,j-2], B[i-2,j+2]).$$

In this case we obtain the matrices

$$\widehat{F}_A = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}, \widehat{F}_{B_1} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}, \widehat{F}_{B_2} = \begin{bmatrix} 1 & 0 & -1 \\ 0 & 1 & -1 \\ 0 & 0 & 1 \end{bmatrix}, \widehat{F}_{B_3} = \begin{bmatrix} 1 & 0 & -2 \\ 0 & 1 & -2 \\ 0 & 0 & 1 \end{bmatrix},$$

$$\widehat{F}_{B_4} = \begin{bmatrix} 1 & 0 & -2 \\ 0 & 1 & 2 \\ 0 & 0 & 1 \end{bmatrix}, \widehat{U} = \begin{bmatrix} \widehat{C} & \widehat{D}_A & \widehat{D}_B \end{bmatrix},$$

and

$$\widehat{V} = \begin{bmatrix} I & I & I & I & I \\ -\widehat{F}_A & 0 & 0 & 0 & 0 \\ 0 & -\widehat{F}_{B_1} & -\widehat{F}_{B_2} & -\widehat{F}_{B_3} & -\widehat{F}_{B_4} \end{bmatrix}.$$

Using Bau et al.'s [14] framework we determine that the only communication-free solution is the trivial solution. Since there is no possibility of axis and stride misalignment for this example we concentrate on the offset misalignment case. Our structure should help us decide that it would be better to leave the constraint arising from the term $B[i-2,j+2]$ without enforcing in which case we can obtain a communication-free non-trivial solution for the rest of the terms. This solution is $\widehat{C} = \widehat{D}_A = \widehat{D}_B = \begin{bmatrix} 1 & -1 & 0 \end{bmatrix}$, i.e. map iteration $(i,j,1)$ and element $(i,j,1)$ from both arrays A and B to processor $(i-j)$, and it can be obtained using Bau et al.'s [14] method by not satisfying the constraint arising from the term $B[i-2,j+2]$, i.e. using

$$\widehat{V} = \begin{bmatrix} I & I & I & I \\ -\widehat{F}_A & 0 & 0 & 0 \\ 0 & -\widehat{F}_{B_1} & -\widehat{F}_{B_2} & -\widehat{F}_{B_3} \end{bmatrix}.$$

Thus we would incur offset misalignment communication due to only one term.

We notice that for the example above we have the nonzero offset vectors $(-1,-1)$, $(-2,-2)$, and $(-2,2)$. We also notice that offset vectors $(-1,-1)$ and $(-2,-2)$ are linearly dependent,i.e. they lie on the same plane (and in this case on the same line), and that offset vector $(-2,2)$ is orthogonal (perpendicular) or has no projection onto the plane where the other two nonzero offset vectors lie. The RIT for the example above is as shown in Table 4.1. The first column in the RIT is the array name. The second column is the constraint number which comes from how the constraint appears in the $\widehat{V}$ matrix. Constraint 1 affects array A only, whereas constraints 2, 3, 4, and 5 affect array B. These constraints are constructed following Bau et al.'s method. Constraint 1 tells us that iteration (i,j) must be performed at the processor holding element $A[i,j]$.

Table 4.1: RIT for $A[i,j] = \mathcal{F}(A[i,j], B[i,j], B[i-1,j-1], B[i-2,j-2], B[i-2,j+2])$

ARRAY NAME	CONS-TRAINT NUMBER	DIMENSION 1ST(i) STRIDE	DIMENSION 2ND(j) STRIDE	USES	OFFSET VECTOR	AXIS
A	1	0	0	2	$(0,0)$	0
B	2	0	0	1	$(0,0)$	0
	3	0	0	1	$(-1,-1)$	0
	4	0	0	1	$(-2,-2)$	0
	5	0	0	1	$(-2,2)$	0

Constraints 2, 3, 4, and 5 tell us that the same processor must also hold elements $B[i,j]$, $B[i-1,j-1]$,$B[i-2,j-2]$,$B[i-2,j+2]$. The third and fourth columns indicate whether there is a coefficient in any of the dimensions which is larger than one. In other words they are used to indicate the presence of stride in the subscript expression corresponding to the array and the constraint specified in the previous two columns. The fifth column indicates how many references are made to each term at the deepest loop level common to all the terms in the assignment statement. The use for this column will become more evident in a future example. Then the sixth column shows the constant offset vectors for the array references. The last column indicates the presence of axis misalignment in the constraint.

Using the information in this table we determine that there is no axis or stride misalignment. If it were otherwise, we would then try to satisfy the corresponding constraints giving priority to axis, then stride, and finally offset. Because there is no possibility for either axis or stride misalignment we then check the nonzero offset vectors and determine that we can group them into two linearly independent groups. Since one of the groups has two elements, i.e. $(-1,-1)$ and $(-2,-2)$ and the other one has only element $(-2,2)$ we determine to leave the constraint affecting the element $B[i-2,j+2]$ unsatisfied, i.e. constraint 5 rather than risking not satisfying the constraints on $B[i-1,j-1]$ and $B[i-2,j-2]$, i.e. constraints 3 and 4. Note that if we had chosen to leave either constraint 3 or constraint 4 without satisfying, we would have found out that we would still be unable to find a non-trivial communication-free solution.

The algorithm for deciding which constraints to relax is shown in Figure 4.1. To explain the algoritm we will apply it to the matrix multiplication example shown below. In this case we have a loop nest with depth of three, i.e. the number of loops is three.

DO $i = 1, n_i$
    DO $j = 1, n_j$
        DO $k = 1, n_k$
            $Z[i,j] = Z[i,j] + X[i,k] * Y[k,j]$
        ENDDO
    ENDDO

ENDDO

Using Bau et al.'s [14] method we obtain

$$F_X = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix}, F_Y = \begin{bmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \end{bmatrix}, F_Z = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix},$$

$$U = \begin{bmatrix} C & D_X & D_Y & D_Z \end{bmatrix}, V = \begin{bmatrix} I & I & I \\ -F_X & 0 & 0 \\ 0 & -F_Y & 0 \\ 0 & 0 & -F_Z \end{bmatrix}.$$

Note that $r = \text{rank}(V) = 9$, i.e. V is a full rank matrix, $M = 9$, and that $q = M - r = 0$. Thus, we can not use Bau et al.'s [14] method for this particular example as is. This is because the null space of V is empty and thus no basis for the right null space of V^T can be found.

Steps 1-3 Bau et al.'s method can not be applied.

Step 4 RIT is as shown in Table 4.2.

Step 5 As shown in RIT constraint 2 will result in axis misalignment if not satisfied. Constraints 1 and 3 will result in offset misalignment if not satisfied, but constraint 3 would result in the least communication.

Step 6 Apply Bau et al.'s method without considering constraint 3.

$$U = \begin{bmatrix} C & D_X & D_Y \end{bmatrix}, V = \begin{bmatrix} I & I \\ -F_X & 0 \\ 0 & -F_Y \end{bmatrix}$$

That is,

$$V = \begin{bmatrix} 1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 1 \\ -1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & -1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & -1 \\ 0 & 0 & 0 & 0 & -1 & 0 \end{bmatrix}.$$

In this case $r = \text{rank}(V) = 6$. Note that the number of rows of V is $M = 7$ and thus $q = M - r = 1$. Thus the right (column) null space (orthogonal complement) of V^T is non-empty. This means that we should be able to find a basis for column null space of V^T and therefore we should be able to find a solution to the problem. This basis will have a dimensionality of $q = 1$, i.e. the number of rows of the solution will be one. To do this we perform row operations on V to bring it to its Smith

Step 1: If there exists a non-trivial solution, then terminate.

Step 2: Ignore offset vectors and check if there exists a non-trivial solution.

Step 3: If there exists a non-trivial solution to Step 2 above and if it is not desired to reduce interprocessor communication due to offset misalignment, then terminate. If it is desired to do otherwise then go to Step 8.

Step 4: If there does not exist a non-trivial solution to Step 2 above, then form RIT.

Step 5: Determine which constraints may result in axis misalignment and which may result in offset misalignment if they are not satisfied.

Step 6: If there exist constraints which could result in axis misalignment if they are not satisfied, then perform Bau et al.'s [14] method considering only these constraints. If no non-trivial solution can be found, then rank the constraints according to an estimate of the amount of communication that could result from each and apply Bau et al.'s [14] method considering only the one that would result in the largest amount of communication. Then add the next constraint in terms of cost of communication and check if there is a non-trivial solution. If a trivial solution is found, then make the last non-trivial solution the current solution.

Step 7: If there are no constraints which could result in axis misalignment and if there are constraints that could result in stride misalignment when not satisfied, then perform the Step 6 above but for the stride constraints.

Step 8: If at this time it is not desired to try to reduce the interprocessor communication arising from the constant offset vectors, then terminate. Otherwise, classify offset vectors of the constraints which are being satisfied in sets of linearly dependent vectors.

Step 9: If a non-trivial solution can be found that takes into account the largest of the offset vector sets, then attempt to find a non-trivial solution that includes the next set and continue until no non-trivial solution can be found. Make last non-trivial solution the current solution.

Figure 4.1: Algorithm for Choosing which Constraint(s) to Leave Unsatisfied When the Problem is Over-constrained.

normal form. Note that we do not show the effect of the column operations nor are we concerned with them. The resulting V matrix after performing the necessary row operations on it and the corresponding H matrix are as shown below:

$$V = \begin{bmatrix} 1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}, \; H = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & -1 & 0 & -1 & -1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 1 & 1 & 0 \end{bmatrix}.$$

Thus

$$C = \begin{bmatrix} 0 & 0 & 1 \end{bmatrix}, D_X = \begin{bmatrix} 0 & 1 \end{bmatrix}, D_Y = \begin{bmatrix} 1 & 0 \end{bmatrix}$$

$$\Longrightarrow C\vec{I} = \begin{bmatrix} 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} i \\ j \\ k \end{bmatrix} = k,$$

$$D_X F_X \vec{I} = \begin{bmatrix} 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} i \\ j \\ k \end{bmatrix} = k,$$

and

$$D_Y F_Y \vec{I} = \begin{bmatrix} 1 & 0 \end{bmatrix} \begin{bmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} i \\ j \\ k \end{bmatrix} = k.$$

Step 7 There are no constraints resulting in stride misalignment if not satisfied.

Step 8 Done.

Keeping in mind that the mapping of the array elements is given by DF the results above indicate that the computation of iteration (i, j, k) and elements $X[i, k]$, and $Y[k, j]$ should be assigned to processor k. Intuitively we see that this is correct since arrays X and Y can be aligned so that the processor that has column k of X also has row k of Y and executes iteration k.

4.3 Comparison With Other Work

Amarasinghe et al. [5] and Anderson and Lam [6], show how to find partitions for *doall* and *doacross* parallelism and, in order to minimize communication across loop nests, they use a greedy algorithm that tries to avoid the largest amounts of potential communication. In order to find data and computation decompositions they express the problem using constraints in matrix notation and find the basis of the null space of known matrices. Their algorithm trades off extra degrees of parallelism to eliminate communication. For

Table 4.2: RIT for Matrix Multiplication

ARRAY NAME	CONS-TRAINT NUMBER	DIMENSION		USES	OFFSET VECTOR	AXIS
		1ST(i) STRIDE	2ND(j) STRIDE			
X	1	0	0	n_k	$(0, 0)$	0
Y	2	0	0	n_k	$(0, 0)$	1
Z	3	0	0	2	$(0, 0)$	0

example, it may convert a parallel loop into a sequential loop. If communication is needed the algorithm tries to reduce the most expensive communication to inexpensive communication by pipelining using tiling. To find dynamic decompositions they use the communication graph whose nodes correspond to the loops in the program and edges represent where data reorganization can occur.

Bau et al. [14] use elementary matrix methods to determine communication-free alignment of code and data. They also deal with the problem of replicating read-only data to eliminate communication. Their work incorporates data dependences in their proposed solution to the problem and the owner-computes rule is assumed. Replication of data is also incorporated into their proposed solution. Their work treats all types of alignment as equal, e.g. it does not incorporate the notion that axis alignment is more important than stride, replication, or offset alignment. The Smith normal form of integers is used as part of their method. Alignment of both data and computation is determined by finding a basis for the null space of a known matrix.

4.4 Chapter Summary

In this chapter we have presented a heuristics-based algorithm to deal with the problem of determining which constraints to leave unsatisfied when we have an over-constrained system of equations from which we want to determine the alignment for both computation and data. The framework we have used is based on the work developed by Bau et al. [14] which uses elementary linear algebra to determine both computation and data alignment for a program. Our method is used when their approach finds that the only communication-free solution is to map all iterations and all the arrays to a single processor, i.e. the trivial solution. Our method aligns the non-conflicting constraints and finds a communication-free solution based on them. Communication will only arise due to the constraints left unsatisfied.

The decision on which constraints to eliminate is based on the amount of communication which would result if the constraint is left unsatisfied. Constraints which result in higher communication are given preference and are kept over those which would result in less communication when left unsatisfied. This is as opposed to the Bau et al. framework which treats all the constraints, whether arising from axis, stride, replication, or offset alignment as equal.

CHAPTER 5

A MATRIX-BASED APPROACH TO FINDING DISTRIBUTIONS

This chapter presents a technique for finding good distributions of arrays and suitable loop restructuring transformations so that communication is minimized in the execution of nested loops on message passing machines. For each possible distribution (by one or more dimensions), we derive the best unimodular loop transformation that results in block transfers of data. As defined earlier in this dissertation, unimodular matrices have a determinant with a value of ± 1 and they are used extensively in parallelizing compilers. Unlike other work which focus on either data layout or on program transformations, this chapter combines both array distributions and loop transformations resulting in good performance. The techniques described here are suitable for dense linear algebra codes.

On a distributed memory machine, local memory accesses are much faster than accesses to non-local data. Inter-processor communication—resulting accesses to non-local data—is a major determinant of the performance of a parallel machine. When a number of non-local accesses are to be made between processors, it is preferable to send fewer but larger messages rather than several smaller messages more frequently (called *message vectorization* [80]). This is because the message setup cost is usually large. Even in shared memory machines, it is preferable to use block transfers.

We should point out that the interprocessor communication time can be modeled as $t = \alpha + \beta * \gamma$, where α and β are machine dependent and γ is the length of the message. Usually $\alpha >> \beta$ and thus it is desirable to communicate longer messages rather than short ones whenever possible.

Given a program segment, our aim is to determine the computation and data mapping onto processors. Parallelism can be exploited by transforming the loop nest suitably and then distributing the iterations of the transformed outermost loop onto the processors. The distribution of data onto processors may then result in communication and synchronization which counters the advantages obtained by parallelism. This chapter presents an algorithm which results in the optimal performance while simultaneously considering the conflicting goals of parallelism and data locality.

While a programmer can manually write code to enhance data locality by specifying data distribution among processors, we present a technique where we can automatically derive data distribution given the program structure. We present a method by which the program is restructured such that when the outer loop iterations are mapped onto the processors, it results in the least communication. Wherever communication is unavoidable, we restructure the inner loop(s) so that data can be transferred using block transfers; such an approach is referred to as *message vectorization.* Our approach relieves the programmer from having to specify the distribution of the arrays and from having to optimize the communication among processors in case this communication is unavoidable.

This chapter is organized as follows: Section 5.1 talks about the need for automatic distribution; Section 5.2 introduces our first algorithm for automatic distribution and vectorization of messages; Section 5.3 is a step by step application of the algorithm to several examples; Section 5.4 shows the advantage of relaxing the owner-computes rule when our algorithm does not find a solution; Section 5.5 presents our algorithm and applies it to an example in a step by step fashion; Section 5.6 applies tiling after the algorithm to obtain higher granularity in the communication; Section 5.7 is a brief presentation of what others have done; and Section 5.8 is our chapter summary.

5.1 AUTOMATIC DISTRIBUTION

In this chapter, we consider the cases where we allocate outer iterations to processors so that each outer loop iteration is done by a single processor. The data is then allocated so that there is minimum communication and all communication is done through block transfers. This chapter deals with an algorithm to restructure the program to enhance data locality while still enabling parallelism. We construct the entries of a legal invertible transformation matrix so that there is a one-to-one mapping from the original iteration space to the transformed iteration space. This transformation when applied to the original loop structure will do the following:

- Allow the outermost loop iteration to be distributed over the processors *i.e.,* an entire outermost iteration is mapped on to a single processor.
- Determine the data distribution (block or cyclic distribution of a single array dimension).
- Allow blocks transfers to be moved out of the innermost loop so that all the necessary data are transferred to the respective local memories before the execution of the innermost loop.

5.1.1 BACKGROUND AND TERMINOLOGY

The transformation matrix is derived from the *data reference matrix* of the array references. Given a loop nest with indices $i_1, i_2, \cdots, i_n$ which is represented by a column vector $\vec{I}$, we define a *data reference matrix*, $A_{\mathcal{R}}$, for each array reference A (distinct or non-distinct) in a loop nest such that the array reference can be written in the form $A_{\mathcal{R}}\vec{I} + \vec{b}$ where $\vec{b}$ is the offset vector. In what follows we assume that the arrays are not replicated onto the available processors.

Example 5.1 Consider the following loop nest.

DO $i = 1, N_1$
  DO $j = 1, N_2$
    DO $k = 1, N_3$

$B[i, j-i] = B[i, j-i] + A[i, j+k]$
ENDDO
ENDDO
ENDDO

In the example above , the data reference matrix for array B is

$$B_{\mathcal{R}} = \begin{bmatrix} 1 & 0 & 0 \\ -1 & 1 & 0 \end{bmatrix},$$

and the data reference matrix for array A is

$$A_{\mathcal{R}} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 1 \end{bmatrix}.$$

Note that there are two data reference matrices for array B though they are identical. For each array, we use only the distinct data reference matrices.

5.1.2 Effect of a Transformation

On applying a transformation T to a loop with index I, the transformed loop index becomes $\vec{I'} = T\vec{I}$ and the transformed data reference matrix becomes $A'_{\mathcal{R}} = A_{\mathcal{R}}T^{-1}$. The columns of T^{-1} determine the array subscripts of the references in the transformed loop. The key aspect of the algorithm presented in this chapter is that the entries of the inverse of the transformation matrix are derived using the data reference matrices.

5.1.3 Motivation

Consider Example 5.1 above which is similar to the one given by Li and Pingali [57]. There are two references to array B (though not distinct) and one reference to array A. Li and Pingali [57] assume that all arrays are distributed by columns and derive a transformation matrix that matches column distribution. In this case, the loop can be distributed in such a way that there is no communication incurred. Both the arrays can be distributed by rows, *i.e.,* each processor can be assigned an entire row of array A and an entire row of array B. This makes the loop run without any communication. We notice that the first row in the data reference matrix for arrays A and B are the same *i.e.,* $\begin{bmatrix} 1 & 0 & 0 \end{bmatrix}$. This allows the first dimension of both the *lhs* and *rhs* arrays to be distributed (*i.e.,* by rows) over the processors so that there is no communication. In the next section, we derive an algorithm to construct a transformation matrix, which determines the distribution of data.

5.1.4 Algorithm

We restrict our analysis to affine array references in loop nests whose upper and lower bounds are affine. We assume that the iterations of the outermost loop are distributed among processors. To exploit data locality and reduce communication among processors, we further look at transformations that facilitate block transfers so that the data elements which are referenced are brought to local memory in large chunks; this allows to amortize the high message start-up costs over large messages. We assume that the data can be distributed along any one dimension of the array (wrapped or blocked) and that the loop index variable appearing in the subscript expression of the distributed dimension of our base array and any array which is identically distributed is that corresponding to the outermost loop. The results can be generalized where data is distributed along multiple dimensions and block transfers set up in outer iterations. Again we assume that the arrays which are used in the iterations of the loop nest are not replicated onto the available processors.

5.1.5 Criteria for Choosing the Entries in the Transformation Matrix

Let the array indices of the original loop be $i_1, i_2, \ldots, i_n$. Let the array indices of the transformed loop be $j_1, j_2, \ldots, j_n$. We look for transformations such that the *lhs* array has the outermost loop index as the only element in any one of the dimensions of the array, *e.g.* $C[*, *, \ldots, j_1, \ldots, *]$ where j_1 is in the r^{th} dimension and "$*$" indicates a term independent of j_1. The *lhs* array can then be distributed along dimension r. This means that the data reference matrix $C'_{\mathcal{R}}$ of the transformed array reference C, has at least one row in which the first entry is non-zero and the rest are zero, *i.e.,* there is a row r in $C'_{\mathcal{R}}T^{-1} = [\alpha, 0, 0, \ldots, 0]$. For all arrays that appear on the right hand side:

- If a row in all the data reference matrices of an array is identical to a row in the reference matrix in the *lhs* array, then that array can be distributed in the same way as the *lhs* array. There is no communication due to that array, since they are always mapped onto the same processor. If all the references of all the arrays have a row in the data reference matrix identical to that of the *lhs* array, then the entire loop can be distributed along that dimension and there is no communication.

- If the condition above does not hold, choose the entries in T^{-1} such that the following conditions hold:

  1. some dimension of the *rhs* reference consists only of the transformed innermost loop index, *e.g.* $A[*, \ldots, j_n, \ldots, *]$; and
  2. all the other dimensions are independent of the innermost loop index (that is, "$*$" indicates a term independent of j_n).

  This means the transformed reference matrix must have only one non-zero in some row r, and that non-zero must occur in column n. If this condition is satisfied, then dimension r of the *rhs* array is not a distributed dimension; thus, we can move communication arising from that *rhs* reference outside the innermost loop. This allows a block transfer to the local memory before the execution of the innermost loop. This means that a row in the transformed data reference matrix $A'_{\mathcal{R}}$ has a row with all entries zero except in the last column, which is non-zero. Also, the last column of the $A'_{\mathcal{R}}$ has all remaining entries as zero.

- If communication could be moved out of the innermost loop, the previous step can be applied repeatedly starting with the deepest loop outside the innermost and working outward; this process can either stop at some level of the outside which communication can not be moved or when there are no more loops in the loop nest to be considered.

The transformation should also satisfy the condition that the determinant is ± 1 and must preserve the dependences in the program.

5.2 The Algorithm

Consider the following loop where n is the loop nesting level and d the dimension of the arrays.

DO $i_1 = 1, N_1$
    ...
        DO $i_n = 1, N_n$
            $L[C]\vec{I} + \vec{B}^l = R[A]\vec{I} + \vec{B}^r$
        ENDDO
    ...
ENDDO

where

$$C = \begin{bmatrix} c_{11} & \dots & c_{1n} \\ \ddots & & \\ c_{d1} & \dots & c_{dn} \end{bmatrix} \text{ and } A = \begin{bmatrix} a_{11} & \dots & a_{1n} \\ \ddots & & \\ a_{d1} & \dots & a_{dn} \end{bmatrix}$$

are the access or reference matrices for the *lhs* array L and *rhs* array R, respectively,

$$\vec{I} = \begin{bmatrix} i_1 \\ \vdots \\ i_n \end{bmatrix}$$

is the iteration vector, and

$$\vec{B}^l = \begin{bmatrix} b^l_1 \\ \vdots \\ b^l_d \end{bmatrix} \text{ and } \vec{B}^r = \begin{bmatrix} b^r_1 \\ \vdots \\ b^r_d \end{bmatrix}$$

are the constant offset vectors for the *lhs* and *rhs* arrays, respectively. Let the inverse of the transformation matrix be

$$Q = T^{-1} = \begin{bmatrix} q_{11} & \cdots & q_{1n} \\ \ddots & & \\ q_{n1} & \cdots & q_{nn} \end{bmatrix}.$$

The algorithm is shown in Figure 5.1. We use the notation $\vec{A}[i,:]$ to refer to the ith row of a matrix A, and $\vec{A}[:,j]$ to refer to the jth column of a matrix A.

5.3 Examples

We illustrate the use of the algorithm through several examples in this section. The reader is referred to the work by Ramanujam and Narayan [67] for a detailed discussion of the algorithm. In the following discussion, we refer to the matrix T^{-1} as the matrix Q.

Example 5.2 Matrix Multiplication

DO $i = 1, N$
  DO $j = 1, N$
    DO $k = 1, N$
      $C[i,j] = C[i,j] + A[i,k] * B[k,j]$
    ENDDO
  ENDDO
ENDDO

The reference matrices of the arrays are:

$$C_{\mathcal{R}} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix}, A_{\mathcal{R}} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix}, \text{ and } B_{\mathcal{R}} = \begin{bmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \end{bmatrix}.$$

Step 1: C distributed along first dimension. Set

$$\begin{aligned} \vec{C}_{\mathcal{R}}[1,:] &\cdot \vec{Q}[:,1] = 1 \\ \vec{C}_{\mathcal{R}}[1,:] &\cdot \vec{Q}[:,2] = 0 \\ \vec{C}_{\mathcal{R}}[1,:] &\cdot \vec{Q}[:,3] = 0 \end{aligned}$$

Therefore we have, $q_{11} = 1$, $q_{12} = 0$, and $q_{13} = 0$.

Step 1a: Derive distribution of array A. Since row 1 of A is the same as that of C, *i.e.,* $\vec{C}_{\mathcal{R}}[1,:] = \vec{A}_{\mathcal{R}}[1,:]$, distribute A and C identically.

Step 2.1: Derive distribution for array B. Check if you can find a matrix, $B_{\mathcal{R}}Q$ of the form

$$B_{\mathcal{R}}Q = \begin{bmatrix} 0 & 0 & 1 \\ ? & ? & 0 \end{bmatrix}$$

Step 0: If a row in the reference matrix of all the arrays are the same, then there is no communication involved. The data can be distributed along the respective dimension and all the data for the computation will be in local memory. (Initialize $i \leftarrow 1$).

Step 1: Distribute *lhs* array along dimension i, i.e. set $\vec{c}_i.[T^{-1}] = [1\ 0\ \cdots\ 0]$, where $\vec{c}_i$ represents row i of the *lhs* array C.

Step 2: Choose a *rhs* array which does not have a row in the reference matrix the same as that of *lhs* array. For each row j in turn, set: $\vec{a}^p_j.[T^{-1}] = [0\ 0\ \cdots\ 0\ 1]$ for a reference to that array and $\vec{a}^p_{k \neq j} \cdot \vec{q}_n = 0$, where $\vec{a}^p_j$ represents row j in the data reference matrix for the p^{th} *rhs* array A, and $\vec{q}_n$ is the n^{th} column of T^{-1}.

If a valid T^{-1} is found, check the determinant of T^{-1}. If non-zero block transfers are possible for that *rhs* array, (break) go to Step 3.

If there are no valid T^{-1} or the determinant of T^{-1} is zero, block transfers are not possible for dimension j on that array with the given distribution of the *lhs* array; therefore, increment j and go to Step 2.

Step 3: Repeat Step 2 for all the reference matrices of a particular array to check the results for that particular value of j.

Step 4: Repeat Step 2 for all distinct arrays on *rhs*. (Increment p)

Step 5: Check the number of arrays where block transfers are possible.

Step 6: Repeat Step 1 to Step 4 for *lhs* array distributed along each of the other dimensions in turn (Increment i).

Step 7: Compare the number of arrays that can have block transfers and distribute *lhs* array along the dimension which yields maximum number of block transfers for the arrays on the right hand side.

Figure 5.1: Algorithm for Data Distribution and Loop Transformations

where ? denotes entries we do not care about. Set

$$\begin{array}{rcl} \vec{B}_{\mathcal{R}}[1,:] & \cdot & \vec{Q}[:,1] = 0 \\ \vec{B}_{\mathcal{R}}[1,:] & \cdot & \vec{Q}[:,2] = 0 \\ \vec{B}_{\mathcal{R}}[1,:] & \cdot & \vec{Q}[:,3] = 1 \end{array}$$

Therefore we have, $q_{31} = 0$, $q_{32} = 0$, and $q_{33} = 1$. In addition, set $\vec{B}_{\mathcal{R}}[2,:] \cdot \vec{Q}[:,3] = 0$. This implies $q_{23} = 0$. Therefore, the first dimension of B is not distributed. Finally we have,

$$T^{-1} = Q = \begin{bmatrix} 1 & 0 & 0 \\ q_{21} & q_{22} & 0 \\ 0 & 0 & 1 \end{bmatrix}.$$

For a unimodular transformation, $q_{22} = \pm 1$. Note that the dependence vector is $[0\ 0\ 1]$, and therefore, there are no constraints on q_{21}. This results in the identity matrix as the transformation matrix, and thus nothing need be done. Distribute A and C by rows, and B by columns. The code shown next gives the best performance we can get in terms of parallelism and locality. Note that the communication is carried out outside of the innermost loop. In this way a coarser grain in the communication pattern is achieved by vectorizing the messages.

```
DO u = 1, N
    DO v = 1, N
        send B[*, u]
        receive B[*, v]
        DO w = 1, N
            C[u, v] = C[u, v] + A[u, w] * B[w, v]
        ENDDO
    ENDDO
ENDDO
```

We go ahead and complete the algorithm by looking at distributing the *lhs* array in the next dimension.

Step 1.1: C distributed along second dimension. Set

$$\begin{array}{rcl} \vec{C}_{\mathcal{R}}[2,:] & \cdot & \vec{Q}[:,1] = 1 \\ \vec{C}_{\mathcal{R}}[2,:] & \cdot & \vec{Q}[:,2] = 0 \\ \vec{C}_{\mathcal{R}}[2,:] & \cdot & \vec{Q}[:,3] = 0 \end{array}$$

Therefore we have, $q_{21} = 1$, $q_{22} = 0$, and $q_{23} = 0$.

Step 1.1a: Derive distribution for array B. Since second row of $B_{\mathcal{R}}$ is the same as the second row of $C_{\mathcal{R}}$ distribute B same as C.

Step 2.2: Derive distribution for array A. Check if you can find a matrix, $A_{\mathcal{R}}Q$ of the form

$$A_{\mathcal{R}}Q = \begin{bmatrix} 0 & 0 & 1 \\ ? & ? & 0 \end{bmatrix}$$

where ? denotes entries we do not care about. Set

$$\begin{aligned} \vec{A}_{\mathcal{R}}[1,:] &\cdot \vec{Q}[:,1] = 0 \\ \vec{A}_{\mathcal{R}}[1,:] &\cdot \vec{Q}[:,2] = 0 \\ \vec{A}_{\mathcal{R}}[1,:] &\cdot \vec{Q}[:,3] = 1 \end{aligned}$$

Therefore we have, $q_{11} = 0$, $q_{12} = 0$ and $q_{13} = 1$; and $\vec{A}_{\mathcal{R}}[2,:] \cdot \vec{Q}[:,3] = 0 \Longrightarrow q_{33} = 0$.

Finally we have,

$$T^{-1} = \begin{bmatrix} 0 & 0 & 1 \\ 1 & 0 & 0 \\ q_{31} & q_{32} & 0 \end{bmatrix}.$$

For a unimodular transformation, $q_{32} = \pm 1$. Therefore,

$$T^{-1} = \begin{bmatrix} 0 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix} \text{ and } T = \begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \end{bmatrix}.$$

Distribute arrays A, B, and C by columns. The transformed loop is given below:

```
DO u = 1, N
    DO v = 1, N
        send A[*, u]
        receive A[*, v]
        DO w = 1, N
            C[w, u] = C[w, u] + A[w, v] * B[v, u]
        ENDDO
    ENDDO
ENDDO
```

We see that the performance of the loop is similar in both cases. Therefore array C can either be distributed by columns with the above transformation, or by rows with no transformation for the same performance with respect to communication. Again notice that the communication is carried outside of the innermost loop.

Consider the Symmetric Rank 2K (SYR2K) code, from the Basic Linear Algebra Subroutines (BLAS) [56]), example shown below.

Example 5.3 SYR2K

DO $i = 1, N$
  DO $j = i, min(i + 2b - 2, N)$
    DO $k = max(i - b + 1, j - b + 1, 1), min(i + b - 1, j + b - 1, N)$
      $C[i, j - i + 1] = C[i, j - i + 1] + A[k, i - k + b] * B[k, j - k + b]$
        $+ A[k, j - k + b] * B[k, i - k + b]$
    ENDDO
  ENDDO
ENDDO

The reference matrices for the arrays are:

$$C_{\mathcal{R}} = \begin{bmatrix} 1 & 0 & 0 \\ -1 & 1 & 0 \end{bmatrix}, A^1_{\mathcal{R}} = B^2_{\mathcal{R}} = \begin{bmatrix} 0 & 0 & 1 \\ 1 & 0 & -1 \end{bmatrix}, \text{ and } A^2_{\mathcal{R}} = B^1_{\mathcal{R}} = \begin{bmatrix} 0 & 0 & 1 \\ 0 & 1 & -1 \end{bmatrix}.$$

Step 1: C row distributed. Set

$$\begin{aligned} \vec{C}_{\mathcal{R}}[1,:] &\cdot \vec{Q}[:,1] = 1 \\ \vec{C}_{\mathcal{R}}[1,:] &\cdot \vec{Q}[:,2] = 0 \\ \vec{C}_{\mathcal{R}}[1,:] &\cdot \vec{Q}[:,3] = 0. \end{aligned}$$

Therefore we have, $q_{11} = 1$, $q_{12} = 0$, and $q_{13} = 0$. None of the other references matrices have any row common with $C_{\mathcal{R}}$.

Step 1.1: Derive distribution of A for the first reference; check if first dimension of A can be not distributed. Check if you can find a matrix, $A^1_{\mathcal{R}}Q$ of the form

$$A^1_{\mathcal{R}}Q = \begin{bmatrix} 0 & 0 & 1 \\ ? & ? & 0 \end{bmatrix}$$

where ? denotes entries we do not care about. Set

$$\begin{aligned} \vec{A}^1_{\mathcal{R}}[1,:] &\cdot \vec{Q}[:,1] = 0 \\ \vec{A}^1_{\mathcal{R}}[1,:] &\cdot \vec{Q}[:,2] = 0 \\ \vec{A}^1_{\mathcal{R}}[1,:] &\cdot \vec{Q}[:,3] = 1. \end{aligned}$$

Therefore, $q_{31} = 0$, $q_{32} = 0$, and $q_{33} = 1$. In addition, $\vec{A}^1_{\mathcal{R}}[2,:] \cdot \vec{Q}[:,3] = 0$ implies $q_{13} - q_{33} = 0$, which is impossible. Therefore, the first dimension of A has to be distributed.

Step 1.2: Derive distribution of A using first reference; check if second dimension of A can be not distributed. Check if you can find a matrix, $A^1_{\mathcal{R}}Q$ of the form

$$A^1_{\mathcal{R}}Q = \begin{bmatrix} ? & ? & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

where ? denotes entries we do not care about. Set

$$\begin{aligned} \vec{A}^1_{\mathcal{R}}[2,:] &\cdot \vec{Q}[:,1] = 0 \\ \vec{A}^1_{\mathcal{R}}[2,:] &\cdot \vec{Q}[:,2] = 0 \\ \vec{A}^1_{\mathcal{R}}[2,:] &\cdot \vec{Q}[:,3] = 1 \end{aligned}$$

and $\vec{A}^1_{\mathcal{R}}[1,:] \cdot \vec{Q}[:,3] = 0$. Therefore, $q_{11} - q_{31} = 0 \Longrightarrow q_{13} = 1$; $q_{12} - q_{32} = 0 \Longrightarrow q_{32} = 0$; and $q_{13} - q_{33} = 1 \Longrightarrow q_{33} = -1$, which is impossible since $q_{33} = 1$. Thus the second dimension of A also has to be distributed. Based on an analysis of the first reference of A, every dimension of A must be distributed. A similar result follows from an analysis of the second reference to A as well. Since the reference matrix for array A and B are the same, there can be no block transfers for B as well.

Step 2.0: C column distributed. Set

$$\begin{aligned} \vec{C}_{\mathcal{R}}[2,:] &\cdot \vec{Q}[:,1] = 1 \\ \vec{C}_{\mathcal{R}}[2,:] &\cdot \vec{Q}[:,2] = 0 \\ \vec{C}_{\mathcal{R}}[2,:] &\cdot \vec{Q}[:,3] = 0. \end{aligned}$$

Therefore,

$$-q_{11} + q_{21} = 1 \Longrightarrow q_{11} = q_{21} - 1,$$

$$-q_{12} + q_{22} = 0 \Longrightarrow q_{12} = q_{22},$$

and

$$-q_{13} + q_{23} = 0 \Longrightarrow q_{13} = q_{23}.$$

Step 2.1a: Derive distribution of A; check if the first dimension of A can be not distributed. Set

$$\begin{aligned} \vec{A}^1_{\mathcal{R}}[1,:] &\cdot \vec{Q}[:,1] = 0 \\ \vec{A}^1_{\mathcal{R}}[1,:] &\cdot \vec{Q}[:,2] = 0 \\ \vec{A}^1_{\mathcal{R}}[1,:] &\cdot \vec{Q}[:,3] = 1. \end{aligned}$$

Therefore, $q_{31} = 0$, $q_{32} = 0$ and $q_{33} = 1$, and

$$\vec{A}^1_{\mathcal{R}}[2,:] \cdot \vec{Q}[:,3] = 0 \Longrightarrow q_{13} - q_{33} = 0 \Longrightarrow q_{13} = 1 \text{ and } q_{23} = 1.$$

This means that under a column distribution of array C, the first reference to array A, i.e. $A^1_{\mathcal{R}}$ allows A to be not distributed along its first dimension. We now check if the same result can be obtained with the second reference to array A, i.e. $A^2_{\mathcal{R}}$.

Step 2.1b: Second reference of A, check if the second reference allows the first dimension of A to be not distributed: Set

$$\begin{aligned} \vec{A}^2_{\mathcal{R}}[1,:] &\cdot \vec{Q}[:,1] = 0 \\ \vec{A}^2_{\mathcal{R}}[1,:] &\cdot \vec{Q}[:,2] = 0 \\ \vec{A}^2_{\mathcal{R}}[1,:] &\cdot \vec{Q}[:,3] = 1. \end{aligned}$$

Therefore, $q_{31} = 0$, $q_{32} = 0$ and $q_{33} = 1$,

$$\vec{A}^2_{\mathcal{R}}[2,:] \cdot \vec{Q}[:,3] = 0 \Longrightarrow q_{23} - q_{33} = 0$$

and $q_{23} = 1$.

Thus both references to A allow A to be not distributed by its first dimension. Thus, A can be column distributed (by its second dimension). Since B has identical array reference matrices as those of A, array B can also be distributed by columns. Recall, that we started out with a column distribution of C. Thus, we have the inverse of the transformation matrix as

$$T^{-1} = \begin{bmatrix} q_{11} & q_{12} & 1 \\ q_{21} & q_{22} & 1 \\ 0 & 0 & 1 \end{bmatrix}.$$

The only constraint on the unknown elements is that the resulting matrix be legal and unimodular.

Thus we choose the unknown values such that T is a legal unimodular transformation. A possible T^{-1} is as shown below.

$$T^{-1} = \begin{bmatrix} 0 & 1 & 1 \\ 1 & 1 & 1 \\ 0 & 0 & 1 \end{bmatrix} \Longrightarrow T = \begin{bmatrix} -1 & 1 & 0 \\ 1 & 0 & -1 \\ 0 & 0 & 1 \end{bmatrix}.$$

The transformed reference matrices are as follows:

$$C'_{\mathcal{R}} = \begin{bmatrix} 0 & 1 & 1 \\ 1 & 0 & 0 \end{bmatrix}, A'^{,1}_{\mathcal{R}} = \begin{bmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \end{bmatrix}, \text{ and } A'^{,2}_{\mathcal{R}} = \begin{bmatrix} 0 & 0 & 1 \\ 1 & 1 & 0 \end{bmatrix}.$$

Using the algorithm above we distribute arrays A, B, and C by columns. In this way we will have communication arising from A and B. Since we are using the owner-computes rule, the accesses to C are all local. We can thus move the communication outside the innermost loop. The transformed code with block transfers is as shown below:

DO $u = max(0, 2-2b), min(N-1, 2b-2)$
  DO $v = max(1-N, 1-b), min(N-1, b-1-u)$
    send $A[*, u]$, $B[*, u]$
    receive $A[*, v+b]$, $A[*, u+v+b]$, $B[*, u+v+b]$, $B[*, v+b]$

```
        DO w = max(1, 1 − v), min(N − u − v, N)
            C[v + w, u + 1] = C[v + w, u + 1] + A[w, v + b] ∗ B[w, u + v + b]
                            + A[w, u + v + b] ∗ B[w, v + b]
        ENDDO
    ENDDO
ENDDO.
```

5.4 RELAXING THE OWNER-COMPUTES RULE

So far we have relied on the use of the owner-computes rule and have thus assumed that the processor who owns the *lhs* element of the assignment statement is the one that performs the computation. There are cases, though, in which using the owner-computes rule will not allow block transfers. When this happens we can try the algorithm by relaxing the owner-computes rule.

If we apply the method presented earlier to the code shown in Example 5.4, we find that whether array C is distributed by rows or by columns both arrays A and B must be distributed in all of their dimensions and thus no block transfers are possible. The code shown in Example 5.4 is a variation of the code we have already seen in Example 5.3.

Example 5.4 Consider the following code:

```
DO i = 1, N
    DO j = 1, N
        DO k = 1, N
            C[i, j] = C[i, j] + A[k, i − k + b] ∗ B[k, j − k + b]
                    + A[k, j − k + b] ∗ B[k, i − k + b]
        ENDDO
    ENDDO
ENDDO
```

By relaxing the owner-computes rule and modifying the algorithm accordingly we find that block transfers are indeed possible. We could distribute arrays A and B by rows and array C by columns and obtain the following code

```
DO u = 1, N
   DO v = 1, N
      DO w = 1, N
         tmp[w] = tmp[w] + A[u, v − u + b] ∗ B[u, w − u + b]
                + A[u, w − u + b] ∗ B[u, v − u + b]
      ENDDO
      send tmp[∗]
      receive C[∗, u]
   ENDDO
ENDDO
```

where tmp is a temporary column vector used to store the column of C which is computed locally. This same column storage is used each time the processor needs to compute a column of C. Another alternative is to distribute arrays A, B, and C by rows instead and use the code shown below

DO $u = 1, N$
  DO $v = 1, N$
    DO $w = 1, N$
      $tmp[w] = tmp[w] + A[u, w - u + b] * B[u, v - u + b]$
        $+ A[u, v - u + b] * B[u, w - u + b]$
    ENDDO
    send $tmp[*]$
    receive $C[u, *]$
  ENDDO
ENDDO

where tmp is a temporary row vector used to store the row of C which is computed locally. This same row storage is used each time the processor needs to compute a row of C.

We notice that when the algorithm presented previously could not find a solution that would allow block transfers we could then, by relaxing the owner-computes rule, allow block transfers by allowing some other processor to perform the computation.

5.5 The Extended Algorithm

In order to explain the algorithm in Figure 5.2 we will use it to obtain the solution for the problem shown below. This is the code from Example 5.4.

DO $i = 1, N$
  DO $j = 1, N$
    DO $k = 1, N$
      $C[i, j] = C[i, j] + A[k, i - k + b] * B[k, j - k + b]$
        $+ A[k, j - k + b] * B[k, i - k + b]$
    ENDDO
  ENDDO
ENDDO

Notice that the accesses to the two-dimensional arrays A, B, and C are such that while C is being accessed along its second dimension it is the first dimension of arrays A and B which is being accessed. In other words this is an example of communication along distinct axes. Note that the alignment phase will not be able to eliminate the interprocessor communication in this case because the index variables that are used for the accesses along the dimensions are different. In other words, the first dimension of A and B are indexed with a variable distinct to the variable used for the accesses along the second dimension of array C.

Step 0 thru Step 7: These steps are the same as in Figure 5.1.

Step 8: If no block transfers are possible, then initialize i to 1.

Step 9: Choose a *rhs* array and distribute it along dimension i. This array is now the base array.

Step 10: Choose an array which does not have a row in the reference matrix the same as that of the base array. For each row j in turn, set: $\vec{b}^p_j.[T^{-1}] = [0\ 0\ \cdots\ 0\ 1]$ for a reference to that array and $\vec{b}^p_{k \neq j} \cdot \vec{q}_n = 0$.
If a valid T^{-1} is found, check the determinant of T^{-1}. If non-zero block transfers are possible for that array, (break) go to Step 11.
If there are no valid T^{-1} or the determinant of T^{-1} is zero, block transfers are not possible for dimension j on that array with the given distribution of the base array; therefore, increment j and go to Step 10.

Step 11: Repeat Step 10 for all the reference matrices of a particular array to check the results for that particular value of j.

Step 12: Repeat Step 10 for all distinct arrays if necessary. (Increment p)

Step 13: If no block transfers are possible, then increment i and repeat Step 10.

Step 14: If block transfers are possible, then stop. Otherwise, initialize i to 1, repeat Step 10 for a new *rhs* base array and stop when a solution is found or there are no more *rhs* arrays to be chosen as base arrays.

Figure 5.2: Expanded Algorithm for Data Distribution and Loop Transformations

We identify the reference matrices as shown below

$$C_{\mathcal{R}} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix}, \; A^1_{\mathcal{R}} = B^2_{\mathcal{R}} = \begin{bmatrix} 0 & 0 & 1 \\ 1 & 0 & -1 \end{bmatrix}, \; A^2_{\mathcal{R}} = B^1_{\mathcal{R}} = \begin{bmatrix} 0 & 0 & 1 \\ 0 & 1 & -1 \end{bmatrix}.$$

The steps resulting from applying the algorithm to the problem above are presented in what follows.

1. C distributed along its first dimension. Set

$$\begin{aligned} \vec{C}_{\mathcal{R}}[1,:] \;&\cdot\; \vec{Q}[:,1] = 1 \\ \vec{C}_{\mathcal{R}}[1,:] \;&\cdot\; \vec{Q}[:,2] = 0 \\ \vec{C}_{\mathcal{R}}[1,:] \;&\cdot\; \vec{Q}[:,3] = 0 \end{aligned}$$

from which we obtain, $q_{11} = 1,\ q_{12} = 0,\ q_{13} = 0$. Note that there are no rows in any of the other reference matrices which are the same as any of the rows of $C_{\mathcal{R}}$. Otherwise, we could determine at this point which of the remaining arrays could be distributed using the same distribution that we have for C.

(a) Derive distribution for the first dimension of array A using the first reference matrix of A, i.e. $A^1_{\mathcal{R}}$ by checking if the first dimension of A can be not distributed. In other words, check if we can find a matrix of the form

$$A^1_{\mathcal{R}} Q = \begin{bmatrix} 0 & 0 & 1 \\ ? & ? & 0 \end{bmatrix}.$$

Set

$$\begin{aligned} \vec{A}^1_{\mathcal{R}}[1,:] \;&\cdot\; \vec{Q}[:,1] = 0 \\ \vec{A}^1_{\mathcal{R}}[1,:] \;&\cdot\; \vec{Q}[:,2] = 0 \\ \vec{A}^1_{\mathcal{R}}[1,:] \;&\cdot\; \vec{Q}[:,3] = 1 \end{aligned}$$

to obtain $q_{31} = 0$, $q_{32} = 0$, and $q_{33} = 1$. To satisfy the requirement that the innermost loop index variable must not appear in the second dimension, also set $\vec{A}^2_{\mathcal{R}}[1,:] \cdot \vec{Q}[:,3] = 0$ from which we obtain $q_{13} = q_{33} = 0$ which is a contradiction to the above finding which said that $q_{33} = 1$. Therefore, the first dimension of A must be distributed and thus we can not perform block transfers for A along its first dimension. Since $B^2_{\mathcal{R}} = A^1_{\mathcal{R}}$ this means that we can not perform block transfers for B along its first dimension either. This is all assuming that C is distributed along its first dimension.

(b) Derive distribution for the second dimension of array A using the first reference matrix of A, i.e. $A^1_{\mathcal{R}}$ by checking if the second dimension of A can be not distributed. In other words, check if we can find a matrix of the form

$$A^1_{\mathcal{R}}Q = \begin{bmatrix} ? & ? & 0 \\ 0 & 0 & 1 \end{bmatrix}.$$

Set

$$\begin{aligned} \vec{A}^1_{\mathcal{R}}[2,:] &\cdot \vec{Q}[:,1] = 0 \\ \vec{A}^1_{\mathcal{R}}[2,:] &\cdot \vec{Q}[:,2] = 0 \\ \vec{A}^1_{\mathcal{R}}[2,:] &\cdot \vec{Q}[:,3] = 1. \end{aligned}$$

Therefore $q_{11} = q_{31} = 0$, $q_{12} = q_{32} = 0$, and $q_{13} = q_{33} = 1$. Now set $\vec{A}^1_{\mathcal{R}}[1,:] \cdot \vec{Q}[:,3] = 0$ that is, $q_{33} = 0$ which is again a contradiction to the above finding which said that $q_{33} = 1$. Thus, the second dimension of A must be distributed and thus we can not perform block transfer for A along its second dimension. Since $B^2_{\mathcal{R}} = A^1_{\mathcal{R}}$ this means that we can not perform block transfers for B along its second dimension either. Remember that this analysis has been made assuming that the *lhs* array C is distributed along its first dimension.

2. C distributed along its second dimension. Set

$$\begin{aligned} \vec{C}_{\mathcal{R}}[2,:] &\cdot \vec{Q}[:,1] = 1 \\ \vec{C}_{\mathcal{R}}[2,:] &\cdot \vec{Q}[:,2] = 0 \\ \vec{C}_{\mathcal{R}}[2,:] &\cdot \vec{Q}[:,3] = 0 \end{aligned}$$

Therefore $q_{21} = 1$, $q_{22} = 0$, and $q_{23} = 0$. Again there are no rows in any of the other reference matrices that are the same to any of the rows of $C_{\mathcal{R}}$.

   (a) Derive distribution for the first dimension of array A using the first reference matrix of A, i.e. $A^1_{\mathcal{R}}$ by checking if the first dimension of A can be not distributed. In other words, check if we can find a matrix of the form

$$A^1_{\mathcal{R}}Q = \begin{bmatrix} 0 & 0 & 1 \\ ? & ? & 0 \end{bmatrix}.$$

   Set

$$\begin{aligned} \vec{A}^1_{\mathcal{R}}[1,:] &\cdot \vec{Q}[:,1] = 0 \\ \vec{A}^1_{\mathcal{R}}[1,:] &\cdot \vec{Q}[:,2] = 0 \\ \vec{A}^1_{\mathcal{R}}[1,:] &\cdot \vec{Q}[:,3] = 1 \end{aligned}$$

to obtain $q_{31} = 0$, $q_{32} = 0$, and $q_{33} = 1$. Also set $\vec{A}^2_{\mathcal{R}}[1,:] \cdot \vec{Q}[:,3] = 0$ which yields $q_{13} = q_{33} = 0$ which is a contradiction. Therefore the first dimension of A must be distributed. This means that we can not perform block transfers along the first dimension of A. Note that $B^2_{\mathcal{R}} = A^1_{\mathcal{R}}$ and thus we can not perform block transfers along the first dimension of B if C is distributed along its second dimension.

(b) Derive distribution for the second dimension of array A using the first reference matrix of A, i.e. $A^1_{\mathcal{R}}$ by checking if the second dimension of A can be not distributed. In other words, check if we can find a matrix of the form

$$A^1_{\mathcal{R}} Q = \begin{bmatrix} ? & ? & 0 \\ 0 & 0 & 1 \end{bmatrix}.$$

Set

$$\begin{aligned} \vec{A}^1_{\mathcal{R}}[2,:] &\cdot \vec{Q}[:,1] = 0 \\ \vec{A}^1_{\mathcal{R}}[2,:] &\cdot \vec{Q}[:,2] = 0 \\ \vec{A}^1_{\mathcal{R}}[2,:] &\cdot \vec{Q}[:,3] = 1. \end{aligned}$$

Therefore $q_{11} = q_{31} = 0$, $q_{12} = q_{32} = 0$, and $q_{13} = q_{33} + 1$. Now set $\vec{A}^1_{\mathcal{R}}[1,:] \cdot \vec{Q}[:,3] = 0$ which yields $q_{33} = 0$. Thus $q_{13} = 1$. This results in

$$Q = \begin{bmatrix} 0 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}$$

which is not unimodular. Therefore the second dimension of A must also be distributed. This means that we can not perform block transfers along the second dimension of A and, as before, since $B^2_{\mathcal{R}} = A^1_{\mathcal{R}}$ we can not perform block transfers along the second dimension of B either. Therefore, using the owner-computes rule does not allow block transfers for either array A or array B.

3. At this time we relax the owner-computes rule and allow the owner of a *rhs* array to be the one performing the computation. A distributed along its first dimension. Set

$$\begin{aligned} \vec{A}^1_{\mathcal{R}}[1,:] &\cdot \vec{Q}[:,1] = 1 \\ \vec{A}^1_{\mathcal{R}}[1,:] &\cdot \vec{Q}[:,2] = 0 \\ \vec{A}^1_{\mathcal{R}}[1,:] &\cdot \vec{Q}[:,3] = 0 \end{aligned}$$

from which we obtain, $q_{31} = 1$, $q_{32} = 0$, $q_{33} = 0$. Note that the first row of $A^2_{\mathcal{R}}$, $B^1_{\mathcal{R}}$, and $B^2_{\mathcal{R}}$ is identical to the first row of $A^1_{\mathcal{R}}$. Therefore, both arrays A and B can be distributed by rows.

(a) Derive distribution for the first dimension of array C to check if it can be not distributed. Check if we can find a matrix of the form

$$C_{\mathcal{R}}Q = \begin{bmatrix} 0 & 0 & 1 \\ ? & ? & 0 \end{bmatrix}.$$

Set

$$\begin{aligned} \vec{C}_{\mathcal{R}}[1,:] &\cdot \vec{Q}[:,1] = 0 \\ \vec{C}_{\mathcal{R}}[1,:] &\cdot \vec{Q}[:,2] = 0 \\ \vec{C}_{\mathcal{R}}[1,:] &\cdot \vec{Q}[:,3] = 1 \end{aligned}$$

which results in $q_{11} = 0$, $q_{12} = 0$, and $q_{13} = 1$. Now set

$$\vec{C}_{\mathcal{R}}[2,:] \cdot \vec{Q}[:,3] = 0$$

which yields $q_{23} = 0$. This means that

$$Q = \begin{bmatrix} 0 & 0 & 1 \\ q_{21} & q_{22} & 0 \\ 1 & 0 & 0 \end{bmatrix}$$

which is unimodular if we choose $q_{22} = \pm 1$, i.e.

$$Q = \begin{bmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \end{bmatrix}.$$

Therefore, by distributing A, B by rows and C by columns and using the transformation shown above we can transform the code to

DO $u = 1, N$
  DO $v = 1, N$
    DO $w = 1, N$
      $tmp[w] = tmp[w] + A[u, w-u+b] * B[u, v-u+b]$
        $+ A[u, v-u+b] * B[u, w-u+b]$
    ENDDO
    send $tmp[*]$
    receive $C[*, u]$
  ENDDO
ENDDO.

5.6 Using Tiling to Obtain Higher Granularity in the Communication Pattern

It would be advantageous to continue to increase the granularity of the communication between processors. One way to accomplish this is by tiling one or more dimensions of the iteration space. Tiling is a well known technique used to assign blocks of iterations, instead of just one at a time, to the available processors [5, 6, 69, 86]. In what follows we provide several ways in which the above code could be tiled. Note that the loops have been interchanged.

5.6.1 Tiling One Dimension Only

We can tile one or more dimensions of the iteration space. For every loop that we tile a new loop, the one that schedules the tiles, will be added. The old loop is modified to schedule the iterations along the tiled dimension. We will now show several examples of how to tile the loop nest that resulted from applying the algorithm in Figure 5.2 (relaxing the owner-computes rule) to the code shown in Example 5.4.

If the loops are interchanged so that the order of the loops is (w, u, v), by tiling loop u, which results in an outer loop u_T and an inner loop u followed by the innermost loop v, we could now either read a block of columns as shown in Figure 5.3(a) for the following code

DO $w \ = \ 1, N$
  DO $u_T \ = \ 1, N, S$
    DO $u \ = \ 1, min(u_T + S - 1, N)$
      DO $v \ = \ 1, N$

$$\begin{aligned} tmp[v, u] \ &= \ tmp[u, v] + A[w, u - w + b] * B[w, v - w + b] \\ &\quad + A[w, v - w + b] * B[w, u - w + b] \end{aligned}$$

      ENDDO
    ENDDO
    send $tmp[*, u_T], tmp[*, u_T + 1], \cdots, tmp[*, u_T + S - 1]$
    receive updated block of columns of C owned locally
  ENDDO
ENDDO

or a block of rows as shown in Figure 5.3(b) for the code shown below, where tmp is a temporary array with S columns. The storage allocated to the temporary array tmp is reused each time a new block of columns of C need to be computed by the processor. The u_T loop schedules the tiles and loops u, and v schedule iterations within the tile.

DO $w \ = \ 1, N$
  DO $u_T \ = \ 1, N, S$
    DO $u \ = \ 1, min(u_T + S - 1, N)$
      DO $v \ = \ 1, N$

```
                tmp[u, v] = tmp[u, v] + A[w, u − w + b] ∗ B[w, v − w + b]
                            + A[w, v − w + b] ∗ B[w, u − w + b]
            ENDDO
        ENDDO
        send tmp[u_T, ∗], tmp[u_T + 1, ∗], ···, tmp[u_T + S − 1, ∗]
        receive updated block of rows of C owned locally
    ENDDO
ENDDO
```

Now, if in the other hand the loops are interchanged so that the order of the loops is (w, v, u), by tiling loop v to obtain a tiled loop v_T and an inner loop v, which encloses loop u, we could either read a block of rows as shown in Figure 5.4(a) for the code below

```
DO w = 1, N
    DO v_T = 1, N, S
        DO v = 1, min(v_T + S − 1, N)
            DO u = 1, N
                tmp[v, u] = tmp[v, u] + A[w, u − w + b] ∗ B[w, v − w + b]
                            + A[w, v − w + b] ∗ B[w, u − w + b]
            ENDDO
        ENDDO
        send tmp[v_T, ∗], tmp[v_T + 1, ∗], ···, tmp[v_T + S − 1, ∗]
        receive updated block of rows of C owned locally
    ENDDO
ENDDO
```

or a block of columns in Figure 5.4(b) for the following code. In either case the v_T loop schedules the tiles and loops v, and u iterate within the tile.

```
DO w = 1, N
    DO v_T = 1, N, S
        DO v = 1, min(v_T + S − 1, N)
            DO u = 1, N
                tmp[u, v] = tmp[u, v] + A[w, u − w + b] ∗ B[w, v − w + b]
                            + A[w, v − w + b] ∗ B[w, u − w + b]
            ENDDO
        ENDDO
        send tmp[∗, v_T], tmp[∗, v_T + 1], ···, tmp[∗, v_T + S − 1]
        receive updated block of columns of C owned locally
    ENDDO
ENDDO
```

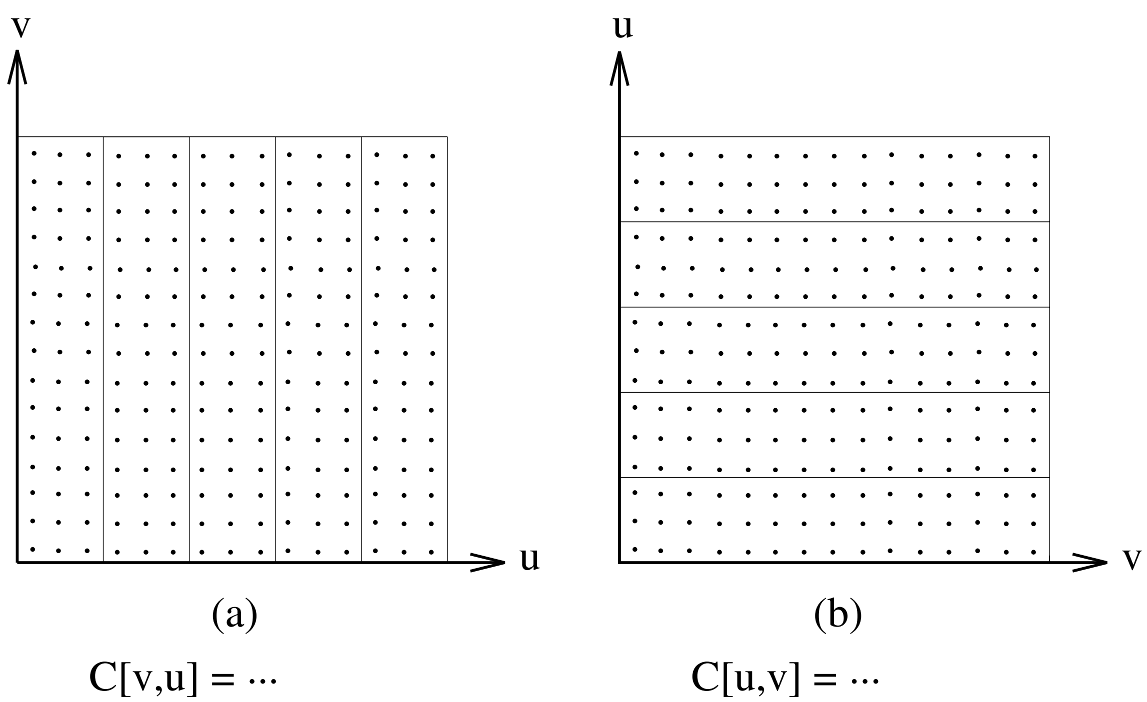


Figure 5.3: Tiling of Loop u to Result in Messages Consisting of (a) Columns or (b) Rows.

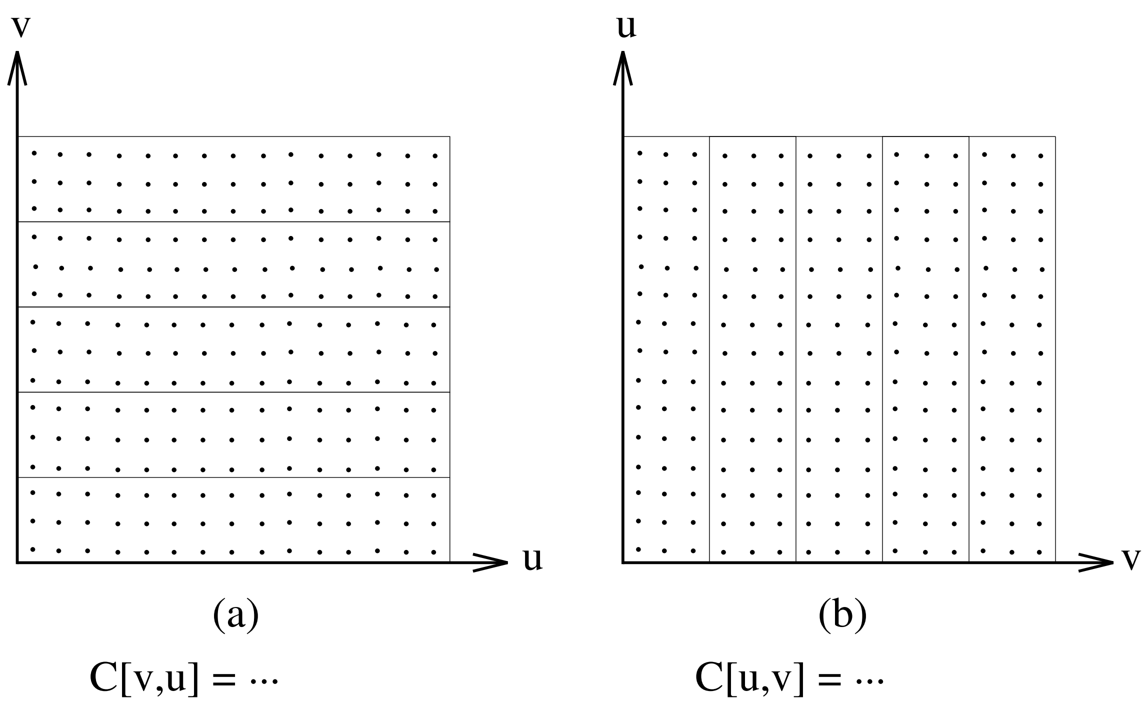


Figure 5.4: Tiling of Loop v to Result in Messages Consisting of (a) Rows or (b) Columns.

5.6.2 TILING TWO DIMENSIONS

In the previous section we showed how to tile the transformed code at the end of Section 5.5 along one dimension of the iteration space. In this section we show how to tile two dimensions of the iteration space of our running example. This could be done in several ways one of which is shown below. Notice that both loops u, and v are tiled resulting in two new loops v_T and u_T along with the modified v and u loops. The tile loops v_T and u_T will schedule the tiles and loops v and u will schedule iterations within the tiles.

We are only tiling a maximum of 2 dimensions of the iteration space. In general, if the original loop nest consists of n loops the tiled loop nest will consist of $2n$ loops n of which will schedule the $n-$ dimensional tiles and the other n will schedule iterations within the tiles. Though both tile block sizes are shown using the same symbol, i.e. S, this does not have to be the case in general and we could have a tile size S_1 for one of the loops and S_2 for the other.

DO $w \ = \ 1, N$
    DO $u_T \ = \ 1, N, S$
        DO $v_T \ = \ 1, N, S$
            DO $u \ = \ 1, min(u_T + S - 1, N)$
                DO $v \ = \ 1, min(v_T + S - 1, N)$

$$\begin{aligned} tmp[v,u] \ = \ & tmp[u,v] + A[w, u-w+b] * B[w, v-w+b] \\ & + A[w, v-w+b] * B[w, u-w+b] \end{aligned}$$

                ENDDO
            ENDDO
            send $tmp[v_T : v_T + S - 1, u_T : u_T + S - 1]$
            receive updated block of columns of C owned locally
        ENDDO
    ENDDO
ENDDO

5.7 COMPARISON WITH OTHER WORK

Li and Pingali [56] used user specified data distributions and developed a systematic loop transformation strategy identified by them as *access normalization* which restructures loop nests to exploit locality and block transfers whenever possible.

Li and Pingali [57] discuss the completion of partial transformations derived from the data access matrix of a loop nest; the rows of the data access matrix are subscript functions for various array accesses (excluding constant offsets). Their work assumes that all arrays are distributed by columns.

Kremer [52] proves the dynamic remapping problem NP-complete. Kremer et al. [53] and Kremer [51] consider the profitability of dynamic remapping and use an interactive tool for automatic data layout, respectively.

5.8 Chapter Summary

In this chapter a (unimodular) matrix-based approach for finding array distributions was presented. This approach finds a unimodular transformation which is derived from the array references. In this way the volume of the iteration space is not affected when going from the original iteration space to the transformed iteration space. In addition, the method also moves communication out of the innermost loop so that messages could be vectorized reducing the amount of communication by an order of magnitude. An algorithm was provided and several detail examples were used to show the effectiveness of this systematic approach. This algorithm begins by assuming the owner-computes rule and relaxes it if no block transfers solution is achieve.

This is the only method that we know of which makes use of matrices in order to determine the best distribution. In this way loop transformations and communication improvements are performed at the same time. The result is a transformed code which performs communication at an outer level so that the data is local to the memories before the execution of the innermost loop. Furthermore, we have presented how to use tiling to increase the granularity of the resulting communication.

CHAPTER 6

DISTRIBUTION: A GRAPH-BASED APPROACH

The distribution phase of the data mapping problem can be defined as the phase where the abstract template, and thus all the arrays aligned to it, are mapped onto the physical processors. This phase comes after the data structures have been aligned to the template. As with the alignment phase, the distribution phase can be subdivided into static distribution and dynamic distribution.

The rest of this chapter is organized as follows: Section 6.1 introduces the distribution preference graph (DPG), Section 6.2 presents how the DPG can be used to determine the distribution of arrays, and Section 6.3 shows the results of applying the DPG method of determining the distribution of arrays to the code for Jacobi, ADI, Disper, Livermore, and Shallow. Section 6.4 is a review of the related work and Section 6.5 the chapter summary.

6.1 THE DISTRIBUTION PREFERENCE GRAPH

Assume that we are using the owner-computes rule where the processor which owns the *lhs* array element of an assignment statement is the one that performs the computation. Then, for a parallel loop, what we need is for the dimension of the *lhs* array subscripted by the loop variable of the parallel loop to be the one to be distributed [30]. In this chapter we will be using the distribution preference graph (DPG). The DPG is a bipartite graph whose nodes represent loop index variables and array dimensions. Undirected edges are used to connect each loop index variable node to the array dimension nodes that use it in the subscripted expression and to connect array dimensions which use the same index variable. Labels are used for the loop node edges and these correspond to the constant coefficient which multiply the index variable in the subscripted expression. We will also use zero weight directed edges connecting a loop node with an array node that does not have the loop index variable as part of its label, though we will not be showing them most of the time. In other words, a zero weight directed edge will be used when the coefficient of the loop index variable for a particular array dimension is zero.

We are looking for disjoint cycles that will include one loop node and exactly one array node from each column. If this is the case we can then distribute an array along the dimension included in the cycle and have the loop corresponding to the loop node variable as the outermost loop by performing loop interchange if necessary.

6.2 HOW TO USE THE DPG TO DISTRIBUTE THE ARRAYS WITHIN A LOOP NEST

Example 6.1 shows a piece of code where an array A is assigned the transpose of array B. The distance vector for this loop nest is $d = (0, 0)^T$, i.e. either loop can be parallelized.

Example 6.1 Row and column

> **DO** $i = 1, N$
>   **DO** $j = 1, N$
> $A[i, j] = B[j, i]$
>   **ENDDO**
> **ENDDO**

The DPG for the example above is shown in Figure 6.1(a) and the available disjoint cycles (1 and 2) in Figure 6.1(b). In Figure 6.2(a) we show cycle 1. If the code above is distributed as suggested by cycle 1, then array A will be distributed by rows and array B by columns to result in no inter-processor communication. Loop i will be the outermost loop. If in the other hand the distribution is determined by cycle 2 in Figure 6.2(b) then array A will be distributed by columns and array B by rows also resulting in no inter-processor communication. In this case loop j would be the outermost loop.

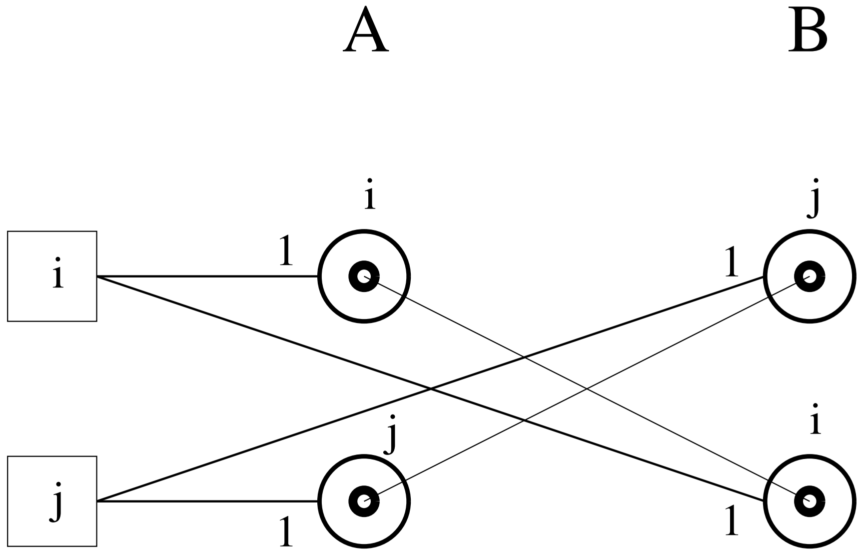
Our next example is the code for matrix multiplication shown in Example 6.2. The DPG is shown in Figure 6.3(a) and the existing cycles in Figure 6.3(b). Note that there are three disjoint cycles (1, 2, and 3) in the DPG shown in Figure 6.3(b) but none of the cycles visits exactly one node from each array. In this case we have that each of the cycles visits exactly one node from two of the three arrays. This is an indication that, unless the array which is not visited by a cycle is replicated, there is no communication-free solution.

Example 6.2 Matrix Multiplication

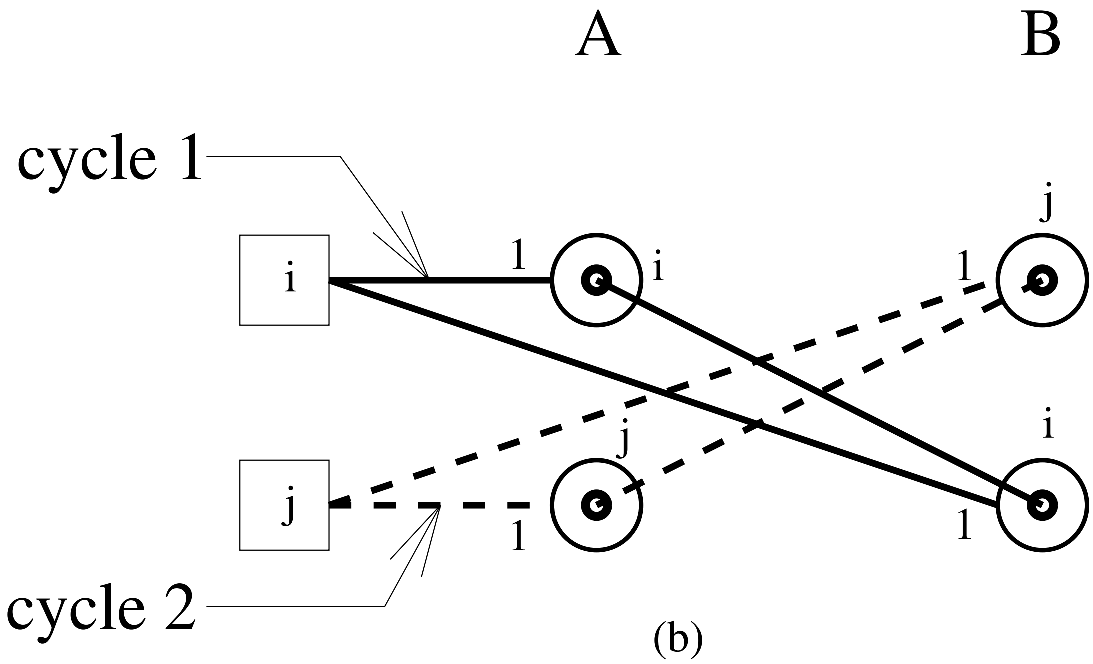
> **DO** $i = 1, N$
>   **DO** $j = 1, N$
>   **DO** $k = 1, N$
> $C[i, j] = C[i, j] + A[i, k] * B[k, j]$
>   **ENDDO**
>   **ENDDO**
> **ENDDO**

In this case our options are: distribute arrays C and A by rows and then choose between a row or a column distribution for B, or distribute C and B by columns and choose between a row or a column distribution for A. Consider cycles 1 and 2 in Figure 6.4(a). If C and A are distributed by rows as indicated by cycle 1 and B by columns as indicated by cycle 2, then the communication arising due to the B term can be moved outside of loop k and an entire block (column) of B ($B[*, j]$) can be read. We can make the i loop the outermost loop, followed by loop j, and then loop k. Since cycle 2 includes index variable j then the j loop will carry the communication which could be moved outside of loop k. Note that there will be no communication arising from the accesses to C and A. The only way to avoid the communication due to B is by replicating array B.

Now consider Figure 6.4(b). In this case we are considering only cycles 1 and 3. If C and A are distributed by rows (cycle 1) and B is distributed by rows (cycle 3) then

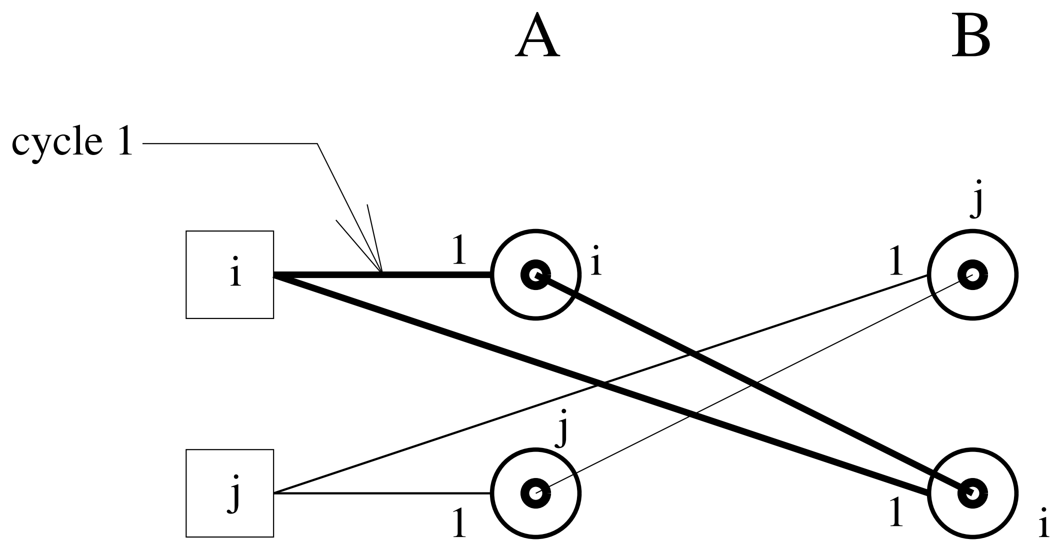


(a)

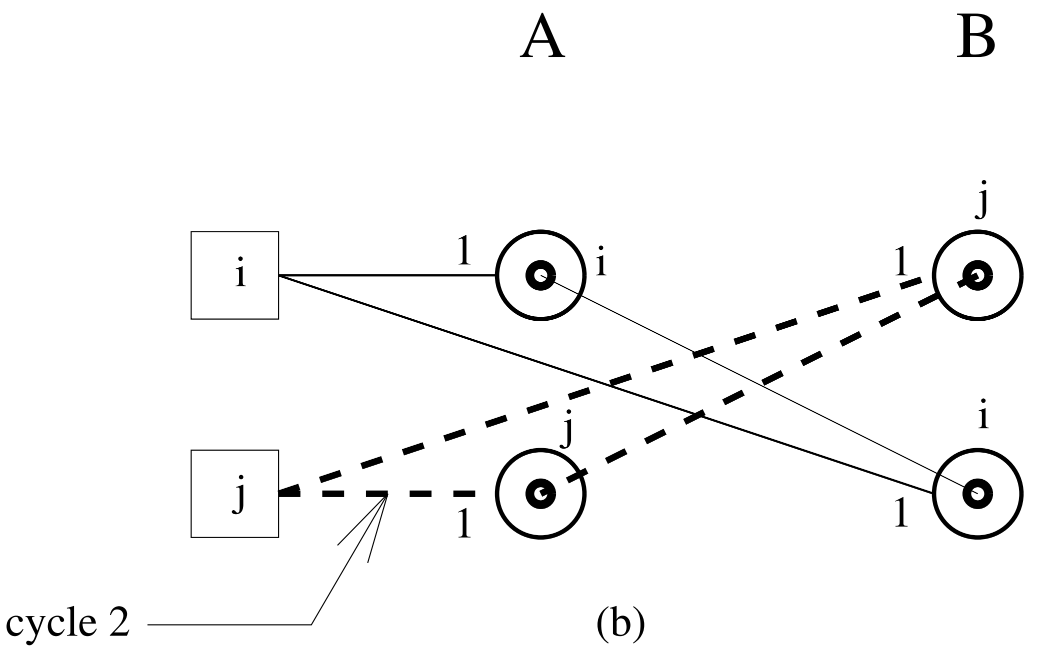


(b)

Figure 6.1: DPG for $A[i, j] = B[j, i]$ in Example 6.1.

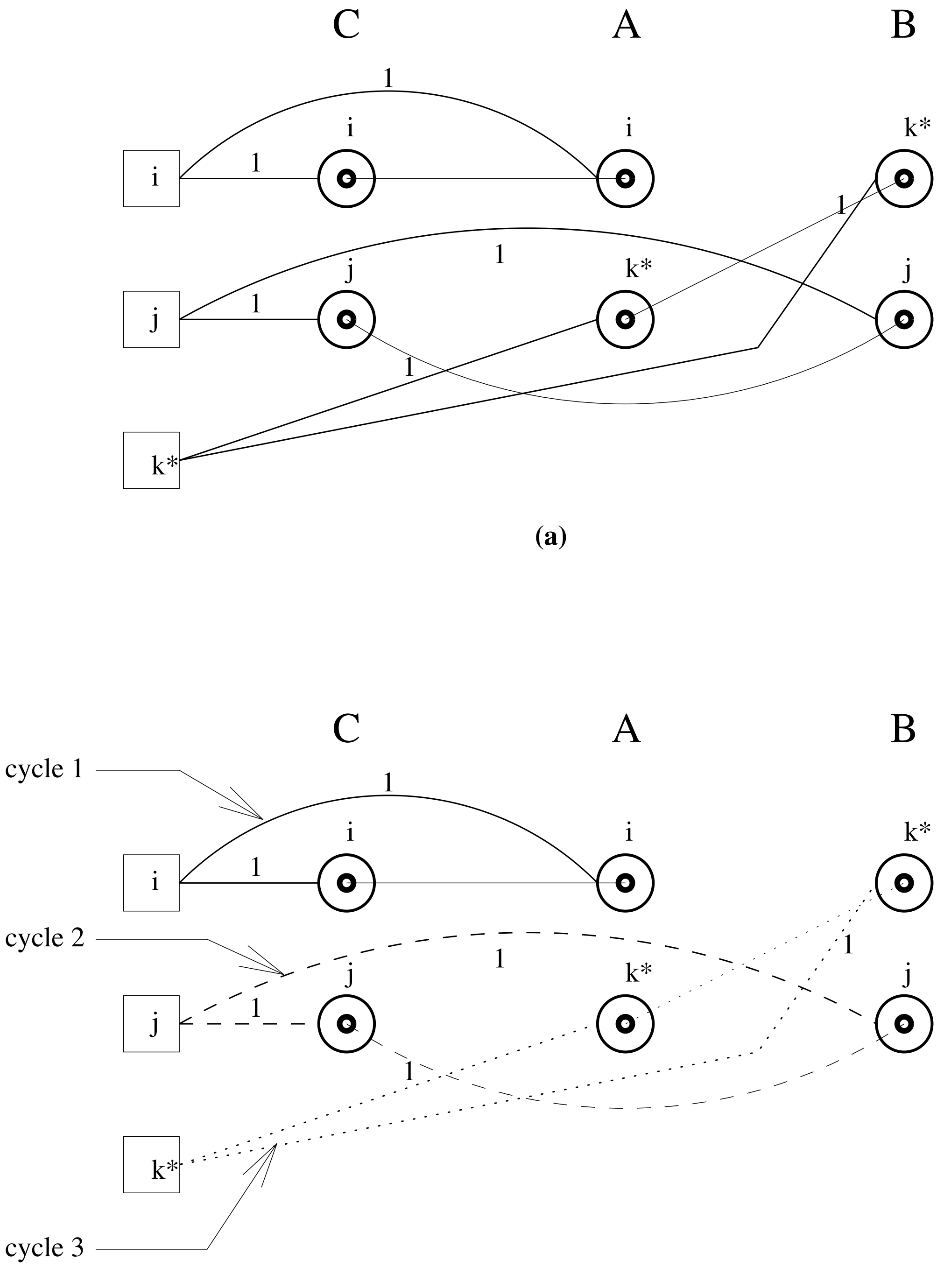


(a)



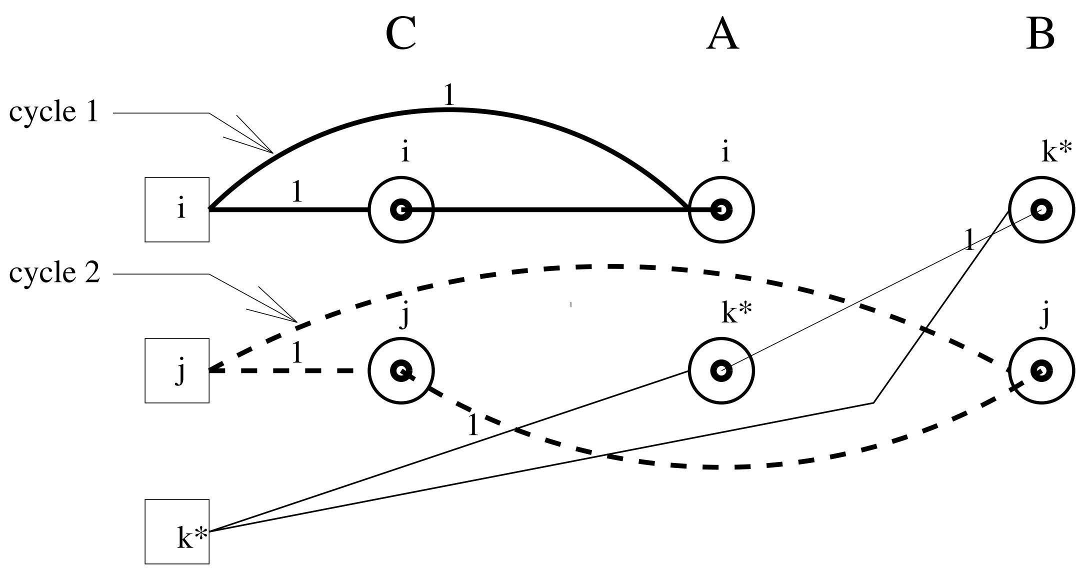
(b)

Figure 6.2: DPG for $A[i, j] = B[j, i]$ in Example 6.1 Showing the Individual Cycles.

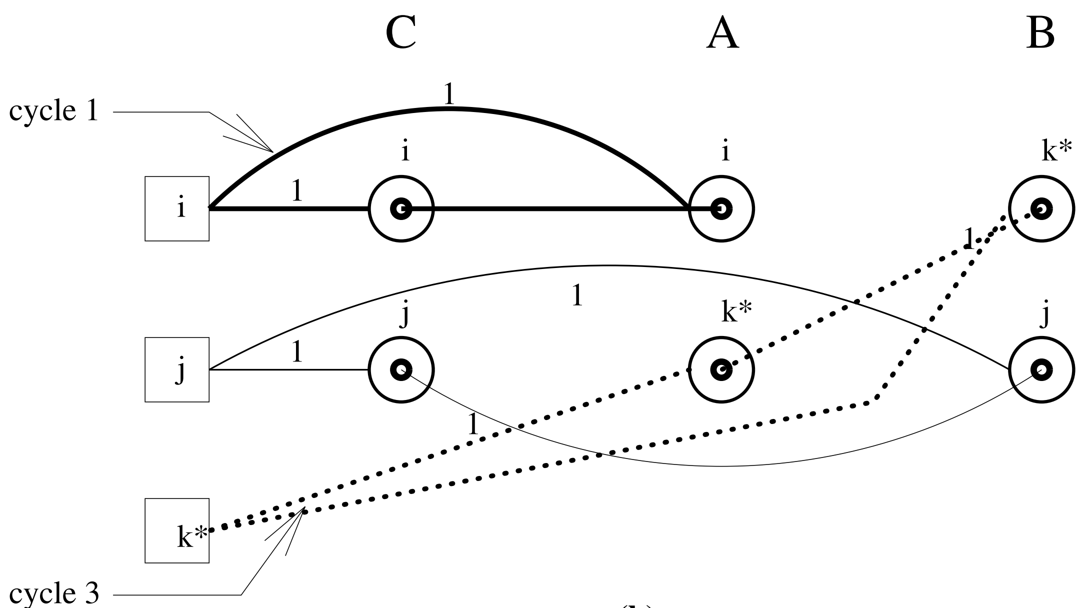


(b)

Figure 6.3: DPG for Matrix Multiplication Example 6.2 Showing (a) No Cycles and (b) Cycles 1, 2, and 3.



(a)



(b)

Figure 6.4: DPG for Matrix Multiplication Example 6.2 Showing (a) Cycles 1 and 2 and (b) Cycles 1 and 3.

there will be no communication from the accesses to arrays C and A, but there will be communication due to B. Since the communication will be carried by loop k, we can make i the outermost loop, followed by loop k, and then loop j and move the communication outside of loop j.

There are other options which can also be determined from the DPG. For example we could choose to distribute C by columns and B by rows (cycle 2) and A by either rows (cycle 1) or columns (cycle 3). If A is distributed by rows we could make j the outermost loop, followed by loop i since it carries the communication due to A, and then loop k as the innermost loop. In this way communication is outside of loop i. If A is distributed by columns then the communication due to A will be carried by loop k. We could then make j the outermost loop, followed by k, and then loop i. The last two options involve making k the outermost loop. We could distribute A by columns and B by rows (cycle 3) and have the choice of distributing C by either rows (cycle 1) or columns (cycle 2). Therefore, the total number of options for the matrix multiplication example is six.

6.3 The DPG Method Applied to Jacobi, ADI, Disper, Livermore, and Shallow

We now apply the Distribution Preference Graph (DPG) method to several programs we have been using throughout this work, e.g. Jacobi, ADI, Disper, Livermore 18, and a section of Shallow. The code segments for these programs were presented in Chapter 2. We will only be showing distinct nodes and in some cases we build the DPG for only part of the code. This is because otherwise the DPG would be too big to follow and also because the rest of the code does not provide any other useful information. Whenever necessary we will label the loop node corresponding to the loop that should be made the outermost loop, i.e. when the loop node is not the first loop in the DPG. In this way we avoid any unnecessary confusion.

Figure 6.5 shows the DPG for the Jacobi algorithm which was presented previously in Figure 2.6. The result of using the DPG method is that we have two cycles and can either distribute arrays A and B by rows or by columns. Note that we are representing the entire code and that we are showing only the array nodes which are distinct.

For statements $S1$ and $S2$ in the ADI code in Figure 2.7 the resulting DPG is shown in Figure 6.6. We are only showing the DPG for the above two statements because it is essentially the same as for the rest of the statements in Figure 2.7. Since the path corresponding to the second dimension in Figure 6.6 is a sequential path, our best choice is to distribute the first dimension of arrays x, a, and b. This choice corresponds to the cycle connecting the first node of all the arrays in Figure 6.6. Note that we have drawn only one of the two possible cycles. The other cycle corresponds to a sequential loop and this is why we do not consider it.

The code for Disper is shown in Figure 2.8 and its corresponding DPG in Figure 6.7. From Figure 6.7 the choice for the distribution of arrays $grady$, $pfmr$, and ddy is made

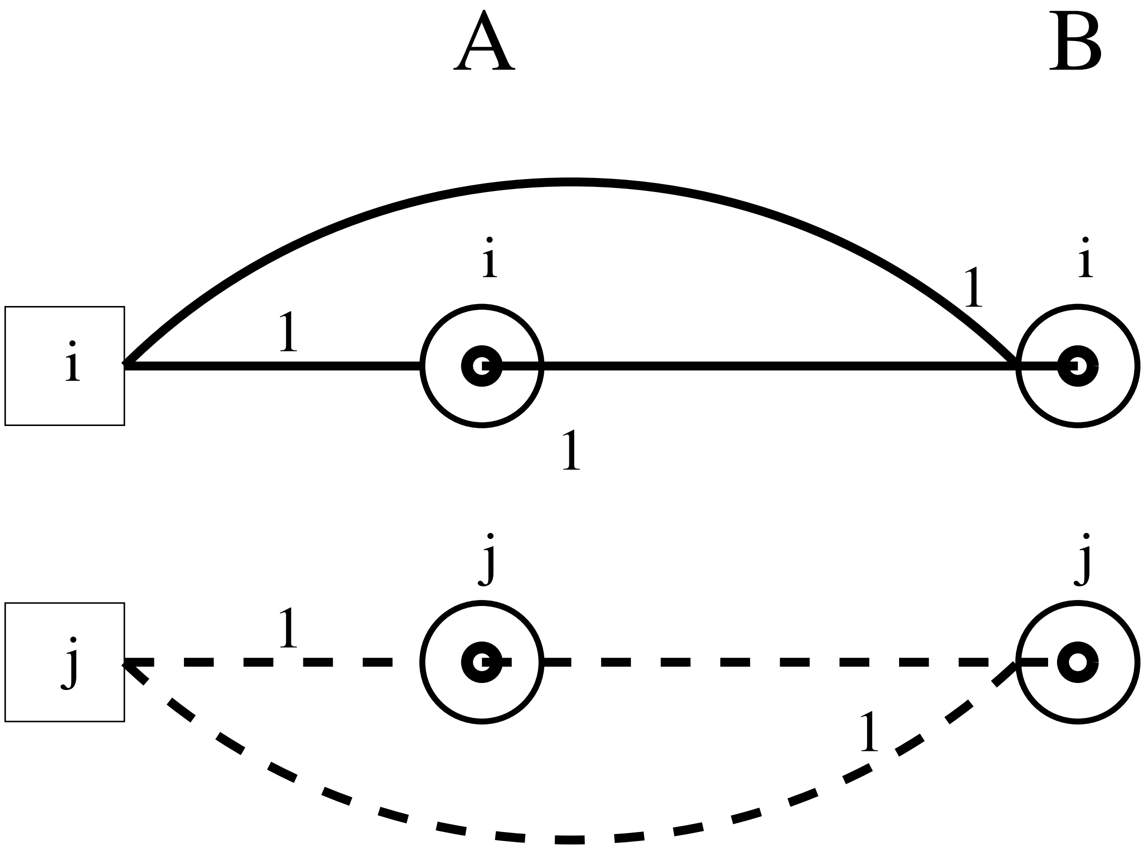


Figure 6.5: DPG for the Jacobi algorithm in Figure 2.6.

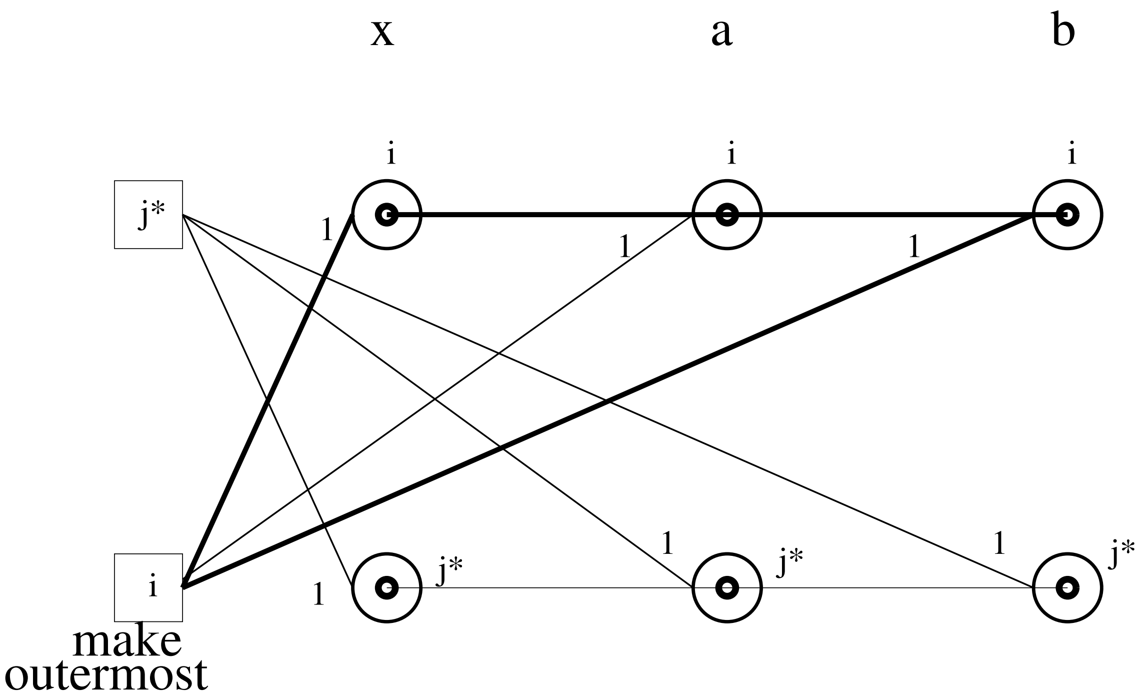


Figure 6.6: DPG for Statements S1 and S2 in the ADI Algorithm in Figure 2.7.

based on the only cycle including a node from each of the arrays, i.e. distribute the arrays along their first dimension. The other cycles were not drawn for the sake of clarity.

The choice of distributing the arrays in the code for Livermore 18 presented in Figure 2.9 along their second dimension is based on the DPG shown in Figure 6.8. This figure shows only cycle along the second dimension of all the arrays. The other cycle in Figure 6.8 corresponds to a sequential loop. This is because loop j carries a dependence. Again note that we have only shown one of the two cycles.

As opposed to the code for Livermore 18, Shallow, shown in Figure 2.12, does not have any dependences and thus either one of the two cycles (only one is shown) in Figure 6.9 is a good candidate for choosing the distribution of the arrays. The arrays can be distributed either along their first dimension or along their second dimension. We have drawn only the one corresponding to the outermost loop and the second dimension of the arrays.

6.4 Comparison With Other Work

Chatterjee et al. [20] and Sheffer et al. [76] deal with determining both static and dynamic distributions. They use the Alignment-Distribution Graph (ADG) whose nodes represent program operations, the ports in the nodes represent array object manipulated by the program, and the edges connect array definitions to their respective uses. The ADG is a directed edge-weighed graph although it is used as an undirected graph. Communication occurs when the alignment or distribution at the end points of an edge is different. The

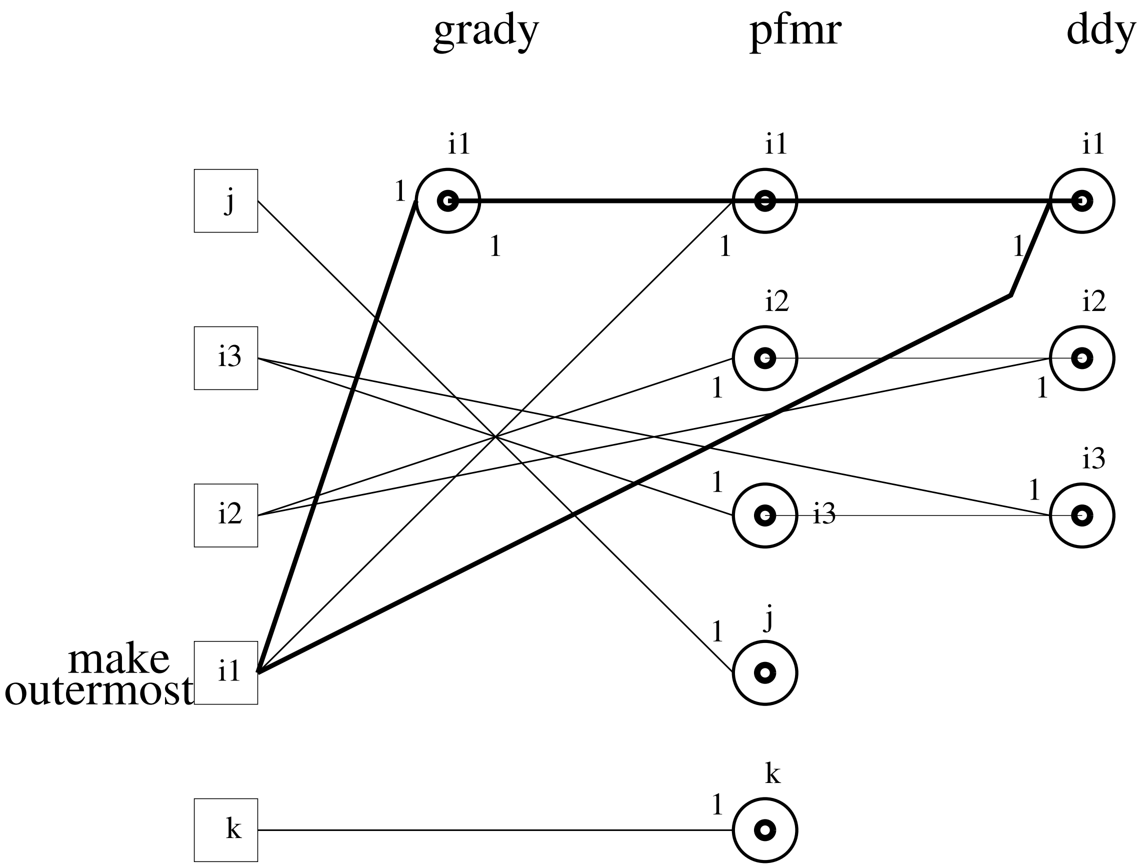


Figure 6.7: DPG for the Disper Algorithm in Figure 2.8.

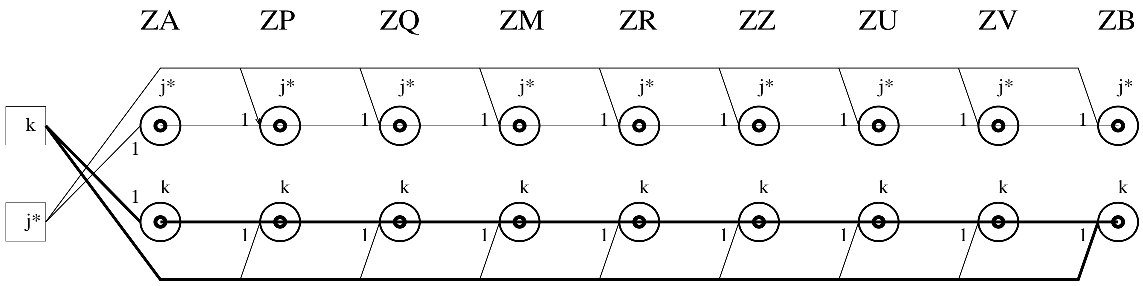


Figure 6.8: DPG for the Livermore 18 Algorithm in Figure 2.9.

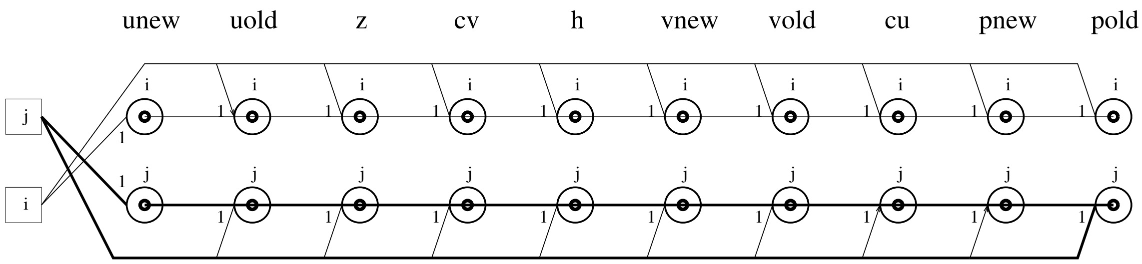


Figure 6.9: DPG for the Shallow Algorithm in Figure 2.12.

completion time of a program is modeled as the sum of the cost over all the nodes (which accounts for computation and realignment) plus the sum over all the edges of the redistribution time (which takes into account the cost per data item of all-to-all personalized communication, the total data volume, and the discrete distance between distributions).

Ayguadé et al.'s [8] main effort is directed toward intra-procedural data mappings. Candidate distributions are used to build a search space from which to determine, based on profitability analyses, the points at which to realign or redistribute the arrays in order to improve the performance by reducing the total data movement. The Component Affinity Graph (CAG) of Li and Chen [55] is used to determine the best local distribution for a particular phase of the code. All the arrays in a phase are distributed identically. Control flow information is used for phase sequencing identification. An intra-procedural remapping algorithm is provided.

Garcia et al. [30] present an approach to automatically perform static distribution using a constraint based model on the Communication-Parallelism Graph (CPG). The CPG contains edges representing both communication and parallelization constraints. The constraints are formulated and solved using a linear 0-1 integer programming model and solver. They obtain solutions for one-dimensional array distributions, i.e. only one dimension of the arrays is distributed, and use an iterative approach for the multi-dimensional problem.

Kremer [52] proves the dynamic remapping problem NP-complete. Kremer et al. [53] and Kremer [51] consider the profitability of dynamic remapping and use an interactive tool for automatic data layout, respectively.

Palermo and Banerjee [63] deal with dynamic partitioning by building the Communication Graph. In this graph the nodes correspond to statements in the program and the edges are flow dependences between the statements. The weight on these edges reflect communication. Maximal cuts are used to remove largest communication constraints and recursively divide the graph or subgraphs until chunks of code (phases) that should share the same partitioning schemes are grouped together. Thus remapping may be inserted between phases and not within a particular phase to reduce communication between phases.

6.5 CHAPTER SUMMARY

The subject of this chapter has been how to determine the distribution of arrays in a loop-nest by making use of a graph-based model. We have presented the distribution preference graph which encodes information about the arrays, the order of the loops in the loop-nest, the communication between nodes in the graph, the subscript variables, and whether the paths in the graph carry any dependences or not. The DPG is flexible enough to allow loop transformations such as loop interchange. An extension to this work is in progress to allow and record the effect of loop skewing. Results were presented for several problems including some benchmarks.

CHAPTER 7

CONCLUSION

In this work we have dealt with the data partitioning problem by dividing it into an alignment phase followed by a distribution phase, following Fortran D and HPF. We have presented several approaches to finding alignment and distribution for a particular piece of code. Our work focuses on loops with assignments to arrays, i.e. the *lhs* of the assignment statement within a loop is an array element. We have chosen to focus on loops and on assignment to arrays because most of the execution of scientific programs is spent in loops and arrays are the predominant data structure used in scientific programming, respectively. As opposed to most researchers we have relaxed the owner-computes rule during our work.

7.1 CONTRIBUTIONS

For the alignment phase we presented three constraint-based methods. One which represents the problem as a general linear programming problem and makes use of a linear programming solver to find the optimal solution. In this case the problem is expressed as a set of inequality constraints based on the first norm or l_1 distance between the virtual processors to which the elements of the template are assigned. At this point the assignment is considered to be one element per virtual processor and the number of virtual processors is assumed to be large enough. We used this method for the offset alignment problem assuming that the values for the parameters ($\alpha' s$) multiplying the index variables are always one (1) and that only the offset coefficients ($\beta' s$) needed to be found. We proved our method to compute both the computation and data alignment using several real life programs that are used in the scientific community. The method was also used to solve both axis and stride alignment sample problems. Linear Programming tools are readily available for a large variety of computer systems.

The second method uses the Lagrange Multiplier method and the software Mathematica [90]. For this method we used the Euclidean or l_2 metric and were able to model the offset alignment problem for the same programs we used during the Linear Programming validation phase. We were also able to model the sample problems we used for the axis and stride alignment cases. As in the case of the LP implementation, we were able to include additional constraints using this method. We have included in Appendix 7.2 an overview of the Lagrange Multiplier method for ease of reference.

The third method uses heuristics to decide which constraints to leave unsatisfied when the system of equations is over-constrained according to the penalty of increased communication incurred in doing so. Our decisions are based on the information stored in the Resource Information Table (RIT). We remind our readers that the component alignment problem has been proven NP-complete by Li and Chen [55].

For the distribution phase we have developed two methods to integrate the placement of computation with data mapping. Our scope have been one dimensional distribution of arrays. In the first method we find the best combination of data and computation mapping resulting in low communication overhead by allowing message vectorization. Our techniques are based on unimodular transformations. We have also made use of tiling to distribute iterations in chunks to further vectorize the messages. This is the first time that this approach to distribution is taken.

Our second approach to distribution is based on the distribution preference graph. This graph is a variation of the component affinity graph and it allows us to integrate loop restructuring transformations and data mappings. Both methods are very useful and could be improved further to allow multidimensional distributions. Note that communication will be unavoidable in the vast majority of the programs and this is the reason we did not restrict ourselves to communication-free problems.

7.2 Summary and Future Work

Our work fits very nicely with the most recent work of some of the researchers in this area where the main effort is directed towards dynamic remapping [8, 48, 49, 63]. Candidate alignments and/or distributions for the different loop nests are assumed and the best mapping (the one that minimizes execution time) is obtained using methods such as graph methods and 0-1 integer programming. In terms of the work performed by these researchers our work provides these candidates for alignment and distribution but it goes further in that we also deal with the problem of reducing communication when this is unavoidable by vectorizing the messages whenever possible.

In terms of future work the following would complement our research.

- Expanding our distribution algorithms to include multidimensional distributions.
- Heuristics for distribution.
- Extending the distribution algorithms for a sequence of loops and imperfectly nested loops, with or without partial replication.
- Including in our framework the use of dependence vectors to guide our algorithms for both alignment and distribution.
- Representing the computation and data alignment problems in such a way that an integer programming tool could be used to find the optimal solution including the optimal solution for axis and stride alignment.
- Accurate modeling of communication costs and communication primitives available on each machine, and matching derived communication to the primitives.
- Dealing with issues such as control and data flow along with interprocedural analysis.

BIBLIOGRAPHY

[1] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley Publishing Company, Reading, MA, 1986.

[2] J. Allen and K. Kennedy. Automatic loop interchange. In *Proc. 1984 SIGPLAN Symp. Compiler Construction*, pages 233–246, June 1984. vol. 19.

[3] J. R. Allen, D. Callahan, and K. Kennedy. Automatic decomposition of scientific programs for parallel execution. In *Proceedings of the Fourteenth Annual ACM Symposium on the Principles of Programming Languages*, Munich, Germany, January 1987.

[4] J. R. Allen and K. Kennedy. Automatic translation of Fortran programs to vector form. *ACM Transactions on Programming Languages and Systems*, 9(4):491–542, October 1987.

[5] S.P. Amarasinghe, J. M. Anderson, M.S. Lam, and A.W. Lim. An overview of a compiler for scalable parallel machines. In *Proceedings of the Sixth Annual Workshop on Languages and Compilers for Parallel Computing*, Portland, OR, August 1993.

[6] J. M. Anderson and M. S. Lam. Global optimizations for parallelism and locality on scalable parallel machines. In *Proceedings of the ACM SIGPLAN'93 Conference on Programming Language Design and Implementation*, pages 112–125, Albuquerque, NM, June 1993.

[7] M. Avriel. *Nonlinear Programming, Analyisis and Methods*. Prentice-Hall, Inc, Englewood Cliffs, N.J., 1976.

[8] E. Ayguadé, J. Garcia, M. Gironès, M.L. Grande, and J. Labarta. Data redistribution in an automatic data distribution tool. In *Proceedings of the Eighth Annual Workshop on Languages and Compilers for Parallel Computing, Lecture Notes in Computer Science*, Columbus, OH, aug 1995. Springer-Verlag.

[9] H.E. Bal, J.G. Steiner, and A.S. Tanenbaum. Programming languages for distributed computing systems. *ACM Computing Surveys*, 21(3), September 1989.

[10] U. Banerjee. An introduction to a formal theory of dependence analysis. *The Journal of Supercomputing*, 2(2):133–149, October 1988.

[11] U. Banerjee. A theory of loop permutations. In D. Gelernter, A. Nicolau, and D. Padua, editors, *Languages and Compilers for Parallel Processing*, Research Monographs in Parallel and Distributed Computing, pages 54–74. Pitman, London, 1990.

[12] U. Banerjee. Unimodular transformation of double loops. In A. Nicolau, D. Gelernter, T. Gross, and D. Padua, editors, *Advances in Languages and Compilers for Parallel Processing*, pages 192–219. Pitman, London, 1991.

[13] U. Banerjee, R. Eigenmann, A. Nicolau, and D. Padua. Automatic program parallelization. *Proceedings of the IEEE*, 81(2):211–243, February 1993.

[14] D. Bau, I. Kodukula, V. Kotlyar, K. Pingali, and P. Stodghill. Solving alignment using elementary linear algebra. In K. Pingali, U. Banerjee, D. Gelernter, A. Nicolau, and D. Padua, editors, *Languages and Compilers for Parallel Computing: Seventh International Workshop*. Springer-Verlag, 1994.

[15] M.S. Bazaraa, H.D. Sherali, and C.M. Shetty. *Nonlinear Programming: Theory and Algorithms*. John Wiley & Sons, Inc., New York, 1993.

[16] D.P. Bertsekas. *Constrained Optimization and Lagrange Multiplier Methods*. Academic Press, Inc., New York, 1982.

[17] 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*, pages 111–122, Montreal, Canada, August 1994.

[18] W. Blume and R. Eigenmann. Performance analysis of parallelizing compilers on the perfect benchmarks programs. *IEEE Transactions on Parallel and Distributed Systems*, 3(6):643–656, November 1992.

[19] D. Callahan, A. Carle, M.W. Hall, and K. Kennedy. Constructing the procedure call multigraph. *IEEE Transactions on Software Engineering*, 16(4):483–487, April 1990.

[20] S. Chatterjee, J. R. Gilbert, R. Schreiber, and T.J. Sheffler. Array distribution in data-parallel programs. In *Proceedings of the Seventh Annual Workshop on Languages and Compilers for Parallel Computing*, August 1994.

[21] S. Chatterjee, John R. Gilbert, Fred J. E. Long, Robert Schreiber, and Shang-Hua Teng. Generating local addresses and communication sets for data-parallel programs. In *Proceedings of the Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 149–158, San Diego, CA, May 1993. Also available as RIACS Technical Report 93.03.

[22] S. Chatterjee, John R. Gilbert, and Robert Schreiber. Mobile and replicated alignment of arrays in data-parallel programs. In *Proceedings of Supercomputing'93*, Portland, OR, November 1993. To appear.

[23] S. Chatterjee, John R. Gilbert, Robert Schreiber, and Shang-Hua Teng. Optimal evaluation of array expressions on massively parallel machines. Technical Report TR

92.17, Research Institute for Advanced Computer Science, NASA Ames Research Center, Moffett Field, CA, September 1992. Also available as Xerox PARC Technical Report CSL-92-11. Submitted to *ACM Transactions on Programming Languages and Systems*.

[24] S. Chatterjee, John R. Gilbert, Robert Schreiber, and Shang-Hua Teng. Automatic array alignment in data-parallel programs. In *Proceedings of the Twentieth Annual ACM SIGACT/SIGPLAN Symposium on Principles of Programming Languages*, pages 16–28, Charleston, SC, January 1993. Also available as RIACS Technical Report 92.18 and Xerox PARC Technical Report CSL-92-13.

[25] K.D. Cooper, M.W. Hall, and K. Kennedy. A methodology for procedure cloning. *Computer Languages*, 9(2), apr 1993.

[26] R. Cytron. Doacross: Beyond vectorization for multiprocessors. In *Proceedings of the 1986 International Conference on Parallel Processing*, pages 836–844, August 1986.

[27] J.Z. Fang and M. Lu. An iteration partition approach for cache or local memory thrashing on parallel processing. *IEEE Transactions on Computers*, 42(5), May 1993.

[28] K. Gallivan, W. Jalby, and D. Gannon. On the problem of optimizing data transfers for complex memory systems. In *Proc. 1988 ACM International Conference on Supercomputing*, pages 238–253, St. Malo, France, June 1988.

[29] D. Gannon, W. Jalby, and K. Gallivan. Strategies for cache and local memory management by global program transformations. *Journal of Parallel and Distributed Computing*, 5(5):587–616, October 1988.

[30] J. Garcia, E. Ayguadé, and J. Labarta. A novel approach towards automatic data distribution. In *Proceedings of Supercomuting '95*, San Diego, CA, December 1995.

[31] J. R. Gilbert and R. Schreiber. Optimal expression evaluation for data parallel architectures. *Journal of Parallel and Distributed Computing*, 13(1):58–64, September 1991.

[32] G. Goff, K. Kennedy, and C. Tseng. Practical dependence testing. In *Proceedings of the SIGPLAN '91 Conference on Program Language Design and Implementation*, Toronto, Canada, June 1991. 1991.

[33] M. Gupta and P. Banerjee. Demonstration of automatic data partitioning techniques for parallelizing compilers on multicomputers. *IEEE Transactions on Parallel and Distributed Systems*, 3(2):179–193, March 1992.

[34] S. K. S. Gupta, S. D. Kaushik, S. Mufti, S. Sharma, C.-H. Huang, and P. Sadayappan. On compiling array exprressions for efficient execution on distributed-memory machines. In Alok N. Choudhary and P. Bruce Berra, editors, *Proceedings of the 1993 International Conference on Parallel Processing*, volume II, pages 301–305. CRC Press, Inc., August 1993.

[35] Mary W. Hall, John M. Mellor-Crummey, Alan Carle, and René G. Rodrı̌guez. FIAT: A framework for interprocedural analysis and transformation. In *1993 Workshop on Languages and Compilers for Parallel Computing*, number 768 in Lecture Notes in Computer Science, pages 522–545, Portland, Ore., August 1993. Berlin: Springer Verlag.

[36] M.W. Hall. *Managing Interprocedural Optimization*. PhD thesis, Rice University, April 1991.

[37] M.W. Hall, S. Hiranandani, K. Kennedy, and C. Tseng. Interprocedural compilation of Fortran D for MIMD distributed-memory machines. In *Proceedings of Supercomputing '92*, Minneapolis, MN, November 1992.

[38] M.W. Hall and K. Kennedy. Efficient call graph analysis. *ACM Letters on Programming Languages and Systems*, 1(3), September 1992.

[39] M.W. Hall, K. Kennedy, and K.S. McKinley. Interprocedural transformations for parallel code generation. In *Proceedings of Supercomputing '91*, Albuquerque, NM, November 1991.

[40] P. Havlak and K. Kennedy. An implementation of interprocedural bounded regular section analysis. *IEEE Transactions on Parallel and Distributed Systems*, 2(3):350–360, July 1991.

[41] High Performance Fortran Forum. High Performance Fortran language specification version 1.0. Draft, December 1993. Center for Research on Parallel Computation, Rice University.

[42] S. Hiranandani, K. Kennedy, J. Mellor-Crummey, and A. Sethi. Advanced compilation techniques for fortran d. Technical Report Rice CRPC-TR-93-338, Center for Research on Parallel Computation, Rice University, Houston, TX, October 1993.

[43] S. Hiranandani, K. Kennedy, and C. Tseng. Compiler optimizations for Fortran D on MIMD distributed-memory machines. In *Proceedings of Supercomputing '91*, Albuquerque, NM, November 1991.

[44] S. Hiranandani, K. Kennedy, and C. Tseng. Compiler support for machine-independent parallel programming in Fortran D. In J. Saltz and P. Mehrotra, editors, *Languages, Compilers, and Run-Time Environments for Distributed Memory Machines*. Noth-Holland, Amsterdam, The Netherlands, 1992.

[45] C.-H. Huang and P. Sadayappan. Communication-free hyperplane partitioning of nested loops. *Journal of Parallel and Distributed Computing*, 19(2):90–102, October 1993.

[46] F. Irigoin and R. Triolet. Supernode partitioning. In *Proc. 15th Annual ACM Symp. Principles of Programming Languages*, pages 319–329, San Diego, CA, January 1988.

[47] Joseph JáJá. *An Introduction to Parallel Algorithms*. Addison Wesley, New York, 1992.

[48] K. Kennedy and U. Kremer. Initial framework for automatic data layout in fortran d: Ashort update on a case study. Technical Report Rice CRPC-TR93324-S, Center for Research on Parallel Computation, Rice University, Houston, TX, July 1993.

[49] K. Kennedy and U. Kremer. Automatic data layout for high performance fortran. Technical Report Rice CRPC-TR94498-S, Center for Research on Parallel Computation, Rice University, Houston, TX, December 1994.

[50] I. Kim and M. Wolfe. Communication analysis for multicomputer compilers. In *Proceedings of Parallel Architectures and Compilation Techniques (PACT 94)*, aug 1994.

[51] U. Kremer. Automatic data layout for distributed-memory machines. Technical Report Rice CRPC-TR93299-S, Center for Research on Parallel Computation, Rice University, Houston, TX, February 1993.

[52] U. Kremer. Np-completeness of dynamic remapping. Technical Report Rice CRPC-TR93330-S, Center for Research on Parallel Computation, Rice University, Houston, TX, August 1993.

[53] U. Kremer, J. Mellor-Crummey, K. Kennedy, and A. Carle. Automatic data layout for distributed-memory machines in the d programming environment. Technical Report Rice CRPC-TR93298-S, Center for Research on Parallel Computation, Rice University, Houston, TX, February 1993.

[54] H.W. Kuhn and A.W. Tucker. Nonlinear programming. In *Proceedings of the Second Berkeley Symposium on Mathematical Statistics and Probability*, pages 481–492, 1951.

[55] J. Li and M. Chen. Index domain alignment: Minimizing cost of cross-referencing between distributed arrays. In *Frontiers '90: The 3rd Symposium on the Frontiers of Massively Parallel Computation*, College Park, MD, October 1990.

[56] W. Li and K. Pingali. Access normalization: Loop restructuring for numa compilers. In *Proceedings Fifth International Conf. on Architectural Support for Programming Languages and Operating Systems*, pages 285–295, Boston, MA, October 1992.

[57] W. Li and K. Pingali. A singular loop transformation framework based on non-singular matrices. In *Proc. 5th Workshop on Languages and Compilers for Parallel Computing*. Springer-Verlag, August 1992.

[58] D. E. Maydan, S. P. Amarasinghe, and M. S. Lam. Array data flow analysis and its use in array privatization. In *Proc. 20th Annual ACM Symposium on Principles of Programming Languages*, January 1993.

[59] D. E. Maydan, J.L. Hennessy, and M.S. Lam. Efficient and exact data dependence analysis. In *Proceedings of the ACM SIGPLAN'91 Conference on Programming Language Design and Implementation*, pages 1–14, June 1991.

[60] D.I. Moldovan and J.A.B. Fortes. Partitioning and mapping algorithms into fixed size systolic arrays. *IEEE Transactions on Computers*, C-35(1):1–12, January 1986.

[61] M. O'Boyle. A data algorithm for distributed memory compilation. In *Proceedings of the Sixth International PARLE Conference*, Athens, Greece, July 1994. Springer-Verlag.

[62] D. A. Padua and M. Wolfe. Advanced compiler optimizations for supercomputers. *Communications of the ACM*, 29(12):1184–1201, December 1986.

[63] D.J. Palermo and P. Banerjee. Automatic selection of dynamic data partitioning schemes for distributed-memory multicomputers. In *Proceedings of the Eighth Workshop on Languages and Compilers for Parallel Computing, Lecture Notes in Computer Science*, Columbus, OH, aug 1995. Springer-Verlag.

[64] T.W. Parsons. *Introduction to Compiler Construction*. W.H. Freeman and Company, New York, 1992.

[65] R.W. Pike. *Optimization for Engineering Systems*. Van Nostrand Reinhold Company, New York, 1986.

[66] M.J. Quinn. *Designing Efficient Algorithms for Parallel Computers*. McGraw Hill, New York, 1987.

[67] J. Ramanujam and A. Narayan. Automatic distribution for HPF-like languages. Technical Report TR-94-07, Department of Electrical and Computer Engineering Louisiana State University, Baton Rouge, LA, January 1994.

[68] J. Ramanujam and P. Sadayappan. Compile-time techniques for data distribution in distributed memory machines. *IEEE Transactions on Parallel and Distributed Systems*, 2(4):472–482, October 1991.

[69] J. Ramanujam and P. Sadayappan. Tiling multidimensional iteration spaces for nonshared memory machines. In *Proceedings of Supercomputing '91*, Albuquerque, NM, November 1991.

[70] G.V. Reklaitis, A. Ravindran, and K.M. Ragsdell. *Engineering Optimization: Methods and Applications*. John Wiley & Sons, Inc., New York, 1983.

[71] S. Richardson and M. Ganapathi. Interprocedural analysis vs. procedure integration. *Information Processing Letters*, 21, August 1989.

[72] S. Richardson and M. Ganapathi. Interprocedural optimization: Experimental results. *Software Practice and Experience*, 19(2), February 1989.

[73] K.H. Rosen. *Elementary Number Theory and Its Applications*. Addison-Wesley, New York, 1985.

[74] R.W. Sebesta. *Concepts of Programming Languages*. Benjamin Cummings, 2nd edition, 1992.

[75] C. Shah. Interprocedural analysis and optimization. Master's thesis, Louisiana State University, Baton Rouge, Louisiana, May 1994.

[76] T.J. Sheffer, R. Schreiber, J.R. Gilbert, and B. Pugh. Efficient distribution analysis via graph contraction. In *Proceedings of the Eighth Workshop on Languages and Compilers for Parallel Computing, Lecture Notes in Computer Science*, Columbus, OH, August 1995. Springer-Verlag.

[77] Z. Shen, Z. Li, and P.-C. Yew. An empirical study of fortran programs for parallelizing compilers. *IEEE Trans. Parallel Distributed Syst.*, 1(3):356–364, July 1990.

[78] J. M. Stichnoth. Efficient compilation of array statements for private memory multicomputers. Technical Report CMU-CS-93-109, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, February 1993.

[79] H.S. Stone. *High Performance Computer Architecture*. Addison Wesley, New York, 2 nd edition, 1990.

[80] C.-W. Tseng. *An Optimizing Fortran D Compiler for MIMD Distributed-Memory Machines*. PhD thesis, Department of Computer Science, Rice University, Houston, TX, January 1993. Available as technical report CRPC-TR93291.

[81] A. Wakatani and M. Wolfe. A new approach to array redistribution: Strip mining redistribution. In *Proceedings of the Sixth International PARLE Conference*, Athens, Greece, July 1994. Springer-Verlag.

[82] M.E. Wolf and M.S. Lam. An algorithmic approach to compound loop transformations. In A. Nicolau, D. Gelernter, T. Gross, and D. Padua, editors, *Proceedings of the Third Workshop on Languages and Compilers for Parallel Computing*, pages 243–259. MIT Press, Irvine, California, 1990. available as Advances in Languages and Compilers for Parallel Computing.

[83] M. Wolfe. Advanced loop interchange. In *Proc. 1986 International Conference on Parallel Processing*, pages 536–543, St. Charles, IL, August 1986.

[84] M. Wolfe. Loop skewing: The wavefront method revisited. *International Journal of Parallel Programming*, 15(4):279–294, 1986.

[85] M. Wolfe. Multiprocessor synchronization for concurrent loops. *IEEE Software*, pages 34–42, January 1988.

[86] M. Wolfe. More iteration space tiling. In *Proc. Supercomputing 89*, pages 655–664, Reno, NV, November 1989.

[87] M. Wolfe. *High Performance Compilers for Parallel Computing*. Addison-Wesley Publishing Company, Redwood City, CA, 1996.

[88] M. Wolfe and U. Banerjee. Data dependence and its application to parallel processing. *International Journal of Parallel Programming*, 16(2):137–178, 1987.

[89] M. J. Wolfe and C. Tseng. The power test for data dependence. *IEEE Transactions on Parallel and Distributed Systems*, 3(5):591–601, September 1992.

[90] S. Wolfram. *Mathematica: A System for Doing Mathematics by Computer*. Addison-Wesley Publishing Company, 1988.

[91] H. Zima and B. Chapman. *Supercompilers for Parallel and Vector Computers*. Frontier Series. Addison-Wesley, 1990.

[92] H. Zima and B. Chapman. Compiling for distributed-memory systems. *Proceedings of the IEEE*, 81(2):264–287, February 1993.

Appendix: Lagrange Multipliers Method

The Lagrange Multipliers method is frequently used for constraint optimization problems. The following derivation is taken from [65]. Assume that we want to optimize a function of two variables $y(x_1, x_2)$ which is subject to constraint $f(x_1, x_2) = 0$. Both functions $y(x_1, x_2)$ and $f(x_1, x_2) = 0$ are expanded into a Taylor series. Using only the first order terms yields:

$$\begin{aligned} dy &= \frac{\partial y}{\partial x_1} dx_1 + \frac{\partial y}{\partial x_2} dx_2 \\ 0 &= \frac{\partial f}{\partial x_1} dx_1 + \frac{\partial f}{\partial x_2} dx_2. \end{aligned}$$

From the second equation above we obtain the following equation by solving for dx_2

$$dx_2 = -\frac{\frac{\partial f}{\partial x_1}}{\frac{\partial f}{\partial x_2}} dx_1.$$

This last equation is then substituted into the equation for dy above to obtain the equation below

$$dy = \frac{\partial y}{\partial x_1} dx_1 - \frac{\partial y}{\partial x_2} \left[\frac{\frac{\partial f}{\partial x_1}}{\frac{\partial f}{\partial x_2}} \right] dx_1$$

from which we obtain the following equation after rearranging the terms

$$dy = \left[\frac{\partial y}{\partial x_1} + \frac{-\frac{\partial y}{\partial x_2}}{\frac{\partial f}{\partial x_2}} \frac{\partial f}{\partial x_1} \right] dx_1.$$

Let λ be defined at the stationary point of the constrained function in the following manner

$$\lambda = -\frac{\frac{\partial y}{\partial x_2}}{\frac{\partial f}{\partial x_2}}.$$

Thus λ is a constant at the stationary point. We then have

$$dy = \left[\frac{\partial y}{\partial x_1} + \lambda \frac{\partial f}{\partial x_1} \right] dx_1 = \frac{\partial (y + \lambda f)}{\partial x_1} dx_1.$$

But $dy = 0$ at the stationary point, thus

$$\frac{\partial (y + \lambda f)}{\partial x_1} = 0.$$

Define the Lagrangian function $L = y + \lambda f$, then

$$\frac{\partial L}{\partial x_1} = 0$$

which is a necessary condition to locate the stationary points of an unconstrained function L constructed from the function $y(x_1, x_2)$ and the constraint equation $f(x_1, x_2) = 0$. Similarly, the other necessary condition is found to be

$$\frac{\partial L}{\partial x_2} = 0.$$

A third equation is given by the constraint equation f. Note that $f = \frac{\partial L}{\partial \lambda} = 0$.

Let x be a vector of n components, i.e. $x = (x_1, x_2, \cdots, x_n)$. Now let y be a function of n variables, i.e. $y(x)$, subject to $f_i(x) = 0$, for $1 \leq i \leq m$, where $n > m$. The Lagrangian function is then defined as follows

$$L(x, \lambda) = y(x) + \sum_{i=1}^{m} \lambda_i f_i(x).$$

The stationary points of this problem are located by setting the first partial derivatives of the Lagrangian function with respect to the x_j's and λ_i's to zero.

So far we have only dealt with the case when we have equality constraints. If we need to deal with inequality constraints, then we add a slack variable to each inequality to convert it into an equality. We find from Avriel [7], Bazaraa et al. [15], Bertsekas [16], Kuhn and Tucker [54], Pike [65], and Reklaitis et al. [70] that to minimize a function

$$L(x, \lambda) = y(x) + \sum_{i=1}^{h} \lambda_i \left[f_i(x) + x_{n+i}^2 \right] + \sum_{i=h+1}^{m} \lambda_i f_i(x)$$

subject to the conditions $f_i(x) \leq 0, i = 1, 2, \cdots, h$, and $f_i(x) = 0, i = h+1, h+2, \cdots, m$, where $n > m$, and the x_{n+i}'s are the slack variables used to convert the inequality constraints to equalities, the necessary conditions for the existence of a relative minimum at a point x^* are:

1. $\frac{\partial L}{\partial x_j}(x^*) + \sum\limits_{i=1}^{h} \lambda_i \frac{\partial L}{\partial x_j}(x^*) + \sum\limits_{i=h+1}^{m} \lambda_i \frac{\partial L}{\partial x_j}(x^*) = 0$ for $j = 1, 2, \cdots, n$
2. $f_i(x^*) \leq 0$ for $i = 1, 2, \cdots, h$
3. $f_i(x^*) = 0$ for $i = h+1, h+2, \cdots, m$
4. $\lambda_i f_i(x^*) = 0$ for $i = 1, 2, \cdots, h$
5. $\lambda_i \geq 0$ for $i = 1, 2, \cdots, h$
6. λ_i is unrestricted in sign for $i = h+1, h+2, \cdots, m$

where n is the number of unknowns, h is the number of inequality constraints, and m ($n \geq m$) is the total number of constraints including equality constraints [65]. The first condition sets the first partial derivatives of the Lagrangian function with respect to $x_i,\ i = 1, 2, \cdots, n$, equal to zero to locate the Kuhn-Tucker point x^*. Conditions 2 and 3 are the inequality and equality constraints, respectively, that must be met at the minimum point found by solving the system of equations obtained from condition 1. The fourth condition comes from setting the partial derivatives of the Lagrangian with respect to the surplus variables equal to zero. Condition 5 arises from the fact that the rate of change of the distance function with respect to the parameters on the *rhs* of the constraints is equal to the negative of the corresponding Lagrange multiplier. By increasing the *rhs* of a constraint the constraint region would be enlarged, which could not result in a larger value for the distance function evaluated at x^* but could result in a lower value. Thus the Lagrange multiplier must be positive to satisfy the rate of change mentioned above [65, 70]. Condition 6 is due to a proof that the Lagrange multipliers associated with the equality constraints are not restricted in sign [65].