Loop and Data Transformations: A Tutorial

Dattatraya Kulkarni and Michael Stumm

Technical Report CSRI-337
June 1993

Computer Systems Research Institute
University of Toronto
Toronto, Canada
M5S 1A1

The Computer Systems Research Institute (CSRI) is an interdisciplinary group formed to conduct research and development relevant to computer systems and their application. It is an Institute within the Faculty of Applied Science and Engineering, and the Faculty of Arts and Science, at the University of Toronto, and is supported in part by the Natural Sciences and Engineering Research Council of Canada.

Loop and Data Transformations: A Tutorial

Dattatraya Kulkarni and Michael Stumm
Department of Computer Science and
Department of Electrical and Computer Engineering
University of Toronto, Toronto, Canada, M5S 1A4
{kulki,stumm}@eecg.toronto.edu

June 1993

Abstract

In this tutorial, we address the problem of restructuring a (possibly sequential) program to improve execution efficiency on parallel machines. This restructuring involves the transformation and partitioning of loop structures and data so as to improve parallelism, static and dynamic locality, and load balance. We present previous and ongoing work on loop and data transformations and motivate a unified framework.

Key Words: Dependence Analysis, Iteration and Data Spaces, Hierarchical Memory, Parallelism, Locality, Load Balance, Conventional and Unified Loop transformations, Data Alignment, Data Distributions.

1 Introduction

In order to be able to execute a program in parallel, it is necessary to partition and map the computation and data onto the processors and memories of a parallel machine. The complexity of parallel architectures has increased in recent years, and an efficient execution of programs on these machines requires that the individual characteristics of the machine be taken into account. In the absence of an automatic tool, the computation and data must be partitioned by the programmer herself. Her strategies usually rely on experience and intuition. However, considering the massive amount of sequential code with high computational demands that already exists, the need for automatic tools is urgent.

Automatic tools have a greater role than just salvaging proverbial *dusty decks*. As the architectural characteristics of parallel hardware become more complex, trade offs in parallelizing a program become more involved, making it more difficult for the programmer to do so in an optimal fashion. This increases the need for tools that can partition computation and data automatically, taking the hardware characteristics into account. Even in the uniprocessor case, the sequential computation model and the architecture is well understood, yet code optimizers still improve the execution of a program significantly. We believe that automatic tools have great promise in improving the performance of parallel programs in a similar fashion. Thus we view such automatic tools as optimizers for parallel machines rather than automatic parallelization tools.

Nested loops are of interest to us because they are the core of scientific and engineering applications which access large arrays of data. This document deals with the restructuring of nested loops and data arrays so that a partitioning of the loops and data gives the best execution time on a target parallel machine. These restructurings are called *transformations*. While some of the problems in loop and data partitioning are computationally hard, efficient approximate solutions often exist. In the following sections we describe the state of the art loop and data transformation techniques and their advantages.

Hierarchically structured machines appear to be becoming the dominant parallel computing structure. These systems have non-uniform memory access times. A restructuring tool has to take into account the hierarchical nature of interaction among processes. Although we do not make explicit assumptions of hierarchy in this document, the essence of non-uniform access is captured in the distinction between the time required by a processor to access data in its local memory and that to access (perhaps by messages) remote data.

In order to illustrate some of the goals of a transformation consider a system comprised of collection of processor-memory pairs connected by an interconnection network. Each processor has a private cache memory. Suppose the two dimensional array A is of size n by n and the linear array B is of size n. Given a two dimensional loop with $n \geq m$:

```
for i = 0, m
  for j = i, n
    A(i, j) = A(i - 1, j) + B(j)
  end for
end for
```

suppose that we map each instance of the outer loop onto a processor so that each processor executes the inner loop iterations sequentially and that there are enough processors to map each instance of outer loop onto a different processor. Hence, processor k executes all iterations with $i = k$. Suppose we map the data so that processor k stores in its local memory A(k, ∗), where ∗ denotes all elements in that dimension of the array, and element B(j) is stored in $(j \bmod (m+1))-$th processor's memory.

In this scenario, depicted in Figure 1, processor k executes $n - k + 1$ iterations. At least $(n - \lceil n/(m+1) \rceil)$ elements of array B() must be obtained remotely from other processors and processor k must fetch A(k − 1, ∗) from processor $(k-1)^{th}$.

Now, consider the following transformed loop nest that is semantically identical to the above loop.

```
for j = 0, n
  for i = 0, min(j, m)
    A(i, j) = A(i - 1, j) + B(j)
  end for
```

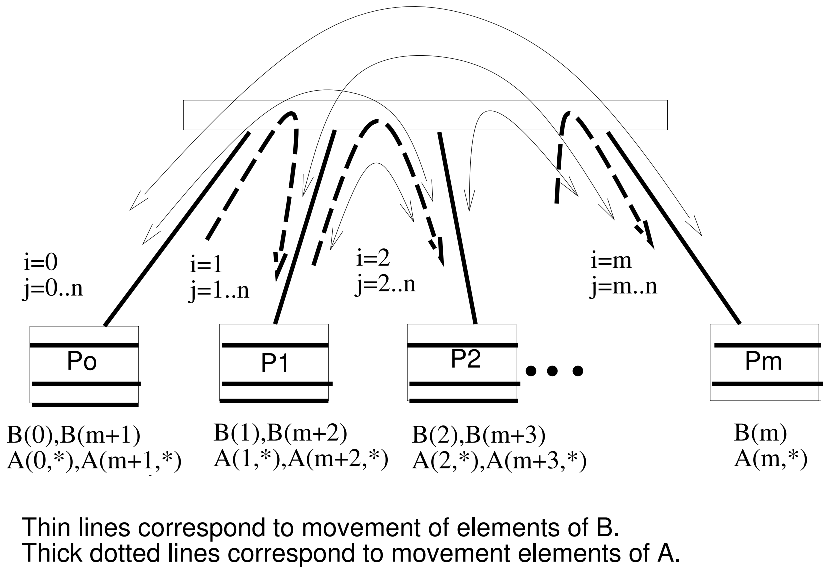

Figure 1: An example mapping

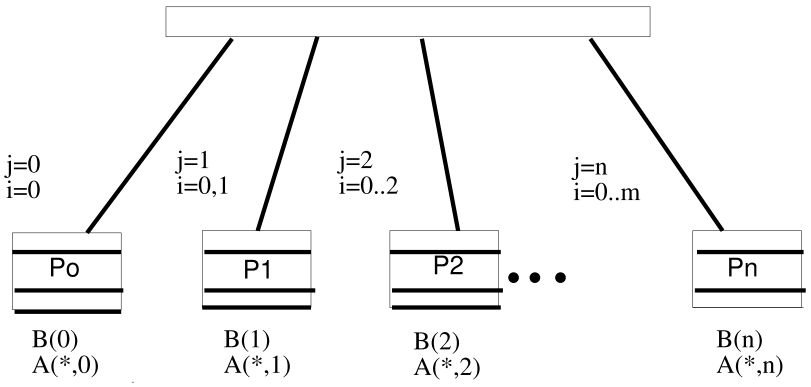

Figure 2: Another mapping

```
end for
```

If processor k executes all iterations with $j = k$, and has stored in its local memory $\mathsf{B(k)}$ and $\mathsf{A}(*, \mathsf{k})$ then processor k will execute $\min(k, m) + 1$ iterations. Because $\mathsf{B(k)}$ is never modified, it can reside in a register (or possibly the cache). Moreover, the elements of A accessed by a processor are all in its *local* memory (Figure 2).

The above mappings use m and n processors, respectively. Since n is larger than m, the second mapping exploits more *parallelism* out of the loop than the first. In the second mapping, all of the computations are *independent* and all data is local, so remote accesses are not necessary. In contrast the first mapping involves considerable inter-processor data movement. The second version of the loop and mapping has better (in this case perfect) *static locality* and has no overhead associated with accesses to remote data. More over, the elements of B() can be kept in a register or at worst in the cache for *reuse* in each iteration, resulting in a better *dynamic locality*. In the first mapping, references to the same element of B() are distributed across different processors.

In both of the mappings, since each processor executes a different number of iterations, the computational *load* on each processor varies. High variance in computational load has a detrimental effect on the performance. Between the above two semantically equivalent loops, the second one has a lower variance, and thus the *load balance* is better.

From the above examples we see that semantically equivalent nested loops can have different execution times based on parallelism, static and dynamic locality, and load balance. There are also other aspects we have not yet considered, such as *replicating* array B() on all processors in the above example. The objective of the restructuring process is to obtain a program that is semantically equivalent, yet performs better due to improvements in parallelism, locality, and load balance for the target parallel machine. *Transforming* the nested loops into semantically equivalent loops is the first concern of this document. Instead of presenting a collection of apparently unrelated existing loop transformations, we present linear transformations that subsume any sequence of existing transformations. We discuss in detail the formalism and techniques for linear transformations. A second concern is the relative mapping of the arrays of data and their distribution across processors. We discuss systematic transformations on data and mapping, and their implications on loop structure are also discussed.

Dependence analysis, the fundamental basis for the transformation methodology is presented in the next section. Section 3 presents the state of the art techniques in loop transformation. Data transformations are discussed in Section 4. We conclude with directions for future research, that includes our current work on a new loop transformation called *Computation Alignment*.

2 Dependence Analysis

The analysis of precedence constraints on the execution of the statements of a program is a fundamental step in parallelizing the program. For a example, the `then` clause of an `if` statement is *control dependent* on the branching condition. A statement that uses the value of a variable assigned by an earlier statement is *data dependent* on the earlier statement[Ban88]. The dependence relation between two statements constrains the order in which the statements may be executed. In this document, we concern ourselves only with data dependence.[1] In this section, we briefly discuss the basic concepts in data dependence and the computational complexity of deciding the existence of a dependence. [Ban88] serves as a very good reference

[1]Control dependence is important to identify functional level parallelism, and to choose between various candidates for data distribution, among other things. Since, our discussion is limited to analysis of dependence between loop iterations, control dependence does not concern us much.

of early development in the area. Recent developments can be found in [LYZ90, WT92, Pug92, MHL91].

There are four types of data dependences: *flow*, *anti*, *output*, and *input* dependence. These dependences are defined as follows.

Definition 1 (Flow dependence) *A statement* S_2 *is flow dependent on statement* S_1 *if a variable assigned in* S_1 *is later referenced in* S_2 *and there is no statement* S_3 *that assigns to the same variable after* S_1 *and before* S_2*. This is denoted by* $\mathsf{S}_1\ \delta^f\ \mathsf{S}_2$*.*

Definition 2 (Anti dependence) *A statement* S_2 *is anti dependent on statement* S_1 *if a variable referenced in* S_1 *is later reassigned in* S_2 *and there is no statement* S_3 *that assigns to the same variable after* S_1 *and before* S_2*. This is denoted by* $\mathsf{S}_1\ \delta^a\ \mathsf{S}_2$*.*

Definition 3 (Output dependence) *A statement* S_2 *is output dependent on statement* S_1 *if a variable assigned in* S_1 *is later reassigned in* S_2 *and there is no statement* S_3 *that assigns to the same variable after* S_1 *and before* S_2*. This is denoted by* $\mathsf{S}_1\ \delta^o\ \mathsf{S}_2$*.*

Definition 4 (Input dependence) *A statement* S_2 *is input dependent on statement* S_1 *if a variable referenced in* S_1 *is later referenced again by* S_2 *and there is no statement* S_3 *that assigns to the same variable after* S_1 *and before* S_2*. This is denoted by* $\mathsf{S}_1\ \delta^i\ \mathsf{S}_2$*.*

Consider the following program segment.

$$\begin{aligned}
&\mathsf{S}_1 : \mathsf{A} = \mathsf{B} + \mathsf{C}\\
&\mathsf{S}_2 : \mathsf{B} = \mathsf{C} + \mathsf{D}\\
&\mathsf{S}_3 : \mathsf{E} = 2 * \mathsf{A} + 1\\
&\mathsf{S}_4 : \mathsf{E} = 5
\end{aligned}$$

Statement S_3 uses A that is defined in S_1. Any valid execution order of the above statements should have S_3 execute after S_1. S_2 reassigns to B that is used in S_1. S_2 can be executed only after S_1. S_4 has to be executed after S_3 because E is reassigned in S_4. Notice however, that S_2 and S_3 can be executed in any order. In this example $\mathsf{S}_1\ \delta^f\ \mathsf{S}_3$, $\mathsf{S}_1\ \delta^a\ \mathsf{S}_2$, $\mathsf{S}_3\ \delta^o\ \mathsf{S}_4$, and $\mathsf{S}_1\ \delta^i\ \mathsf{S}_2$ hold.

The only *true dependence* is flow dependence. The other dependences are the result of reusing the same location of memory and are hence called *pseudo dependences*. They can be eliminated by renaming some of the variables [CF87]. For this reason we write $S1\ \delta\ S2$ to mean flow dependence from S_1 to S_2 from now on.

Consider the following loop:

```
for  i = 0, n
   S1 :  A(i) = A(i - 1)
   S2 :  B(i) = A(i)
end for
```

It is easy to show that $S_1 \ \delta \ S_2$. Let us denote the instance of S_1 (S_2) in iteration i as $S_1[i]$ ($S_2[i]$). If we *unroll* the loop, and write down the statement instances $S_1[0]$,$S_2[0]$,$S_1[1]$, $S_2[1]$,...,$S_1[n]$,$S_2[n]$ then $S_1[1]$ references array element $A(0)$ assigned to by $S_1[0]$, and hence $S_1[0] \ \delta \ S_1[1]$. In general, $S_1[i] \ \delta \ S_1[i+1]$ for $0 \leq i \leq n-1$. Since these dependences exist between statements in different iterations they are called *loop carried* dependences as opposed to *loop independent* dependences between statements in the same iteration. In this document we will focus entirely on loop carried flow dependences of array variables.

A compact representation of a loop carried dependence can be given by identifying the iterations in a nested loop with tuples. Consider the following generic nested loop.

```
for I1 = L1, U1
  for I2 = L2(I1), U2(I1)
    ...
      for In = Ln(I1, .., In-1), Un(I1, .., In-1)
        H(I1, ..., In)
      end for
    ...
  end for
end for
```

$I_1, ..., I_n$ are the iteration indices; L_i and U_i, the lower and upper loop limits, are linear functions of iteration indices $I_1, .., I_{i-1}$; and implicitly a stride of one is assumed. $\mathsf{I} = (I_1, ..., I_n)^T$ is called the iteration vector. H is the body of the nested loop. Typically, an access to an m-dimensional array A in the body of the loop has the form $A(f_1(I_1, ..., I_n), ..., f_m(I_1, ..., I_n))$, where f_i 's are functions of the iteration indices, and are called subscript function. A loop of the above form with linear subscript functions is called an *affine loop.*

Definition 5 (Iteration space) *[Ban88]*[2] $\mathcal{I} \subseteq \mathbf{R}^n$ *such that*

$$\mathcal{I} = \{(i_1, ..., i_n) \mid L_1 \leq i_1 \leq U_1, ..., L_n(i_1, ..., i_{n-1}) \leq i_n \leq U_n(i_1, ..., i_{n-1})\},$$

is an iteration space, where i_1, ..., i_n *are the iteration indices, and* $(L_1, U_1), ..., (L_n, U_n)$ *are the respective loop limits.*

As is evident from the definition, we characterize the iteration space by a set of inequalities corresponding to loop limits. A *bound matrix* is a succinct way of specifying the bounds of the loop iteration space. Since each of the lower and upper bounds are affine functions of the iteration vector, the set of bounds can be written in a matrix vector notation. The set of lower bounds can be represented by

[2] In the definition, the iteration space was deliberately chosen to be in real space to enable us to use results from real space for dependence analysis. In practice, however, the iteration space will always be in integer space.

$$S_L\,\mathbf{I} \geq \mathbf{l}$$

where S_L is an n by n integer lower triangular matrix, $\mathbf{I}$ is the n by 1 iteration vector, and $\mathbf{l}$ is a n by 1 integer vector. Similarly, upper bounds can be represented by

$$S_U\,\mathbf{I} \leq \mathbf{u} \quad \text{or} \quad -S_U\,\mathbf{I} \geq -\mathbf{u}$$

.

We denote the inequalities corresponding to both the upper and lower bounds by a combined set of inequalities $(S, \mathbf{c})$.

$$(S, \mathbf{c}) = \left(\begin{bmatrix} \mathsf{S_L} \\ \hline -\mathsf{S_U} \end{bmatrix}, \begin{bmatrix} \mathsf{l} \\ -\mathsf{u} \end{bmatrix} \right)$$

$(S, \mathbf{c})$ is a complete description of the polyhedral shape of the iteration space.[3]

$$S\,\mathbf{I} \geq \mathbf{c}$$

Example 1 Consider the following doubly nested loop which has an iteration space as shown in Figure 3.

```
for I1 = 0, 10
   for I2 = 0, 10
      A(I1, I2) = A(I1 − 1, I2) + A(I1, I2 − 1) + A(I1 − 2, I2 + 1)
   end for
end for
```

From the upper and lower loop limits we can identify the following:

$$S_L = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \qquad \mathbf{l} = \begin{bmatrix} 0 \\ 0 \end{bmatrix} \qquad S_U = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \qquad \mathbf{u} = \begin{bmatrix} 10 \\ 10 \end{bmatrix}$$

Thus the bounds can be represented by

$$\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} I_1 \\ I_2 \end{bmatrix} \geq \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

$$\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} I_1 \\ I_2 \end{bmatrix} \leq \begin{bmatrix} 10 \\ 10 \end{bmatrix}$$

[3] If maximum and minimum functions exist in the expressions for the loop bounds then the number of inequalities will increase. For example, if a lower bound for index $\mathsf{I_2}$ is $\max(\mathsf{I_1}, 10 - \mathsf{I_1})$, then $\mathsf{I_2} - \mathsf{I_1} \geq 0$ and $\mathsf{I_2} + \mathsf{I_1} \geq 10$ both belong to the set of inequalities S_L.

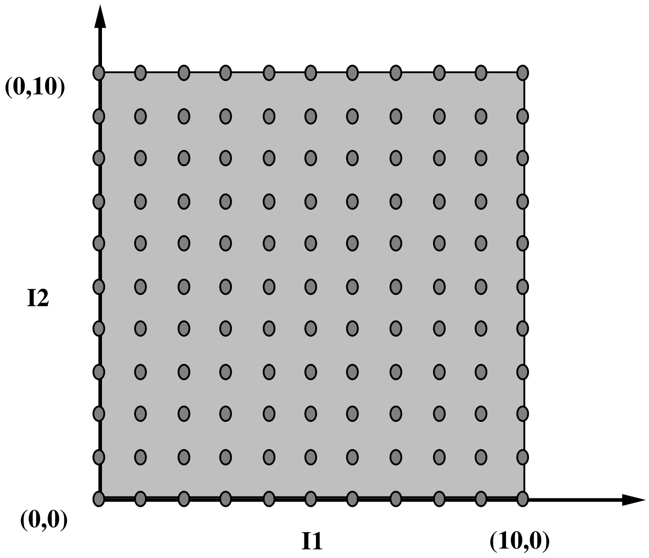

Figure 3: Iteration space

Both upper and lower bounds can represented by

$$\left[\begin{array}{rr} 1 & 0 \\ 0 & 1 \\ \hline -1 & 0 \\ 0 & -1 \end{array}\right] \left[\begin{array}{c} I_1 \\ I_2 \end{array}\right] \geq \left[\begin{array}{r} 0 \\ 0 \\ \hline -10 \\ -10 \end{array}\right]$$

□

Individual iterations are denoted by tuples of iteration indices. Thus, there is a lexicographical order defined on the iterations, and this order corresponds to the sequential execution order.

Definition 6 (Lexicographic order $<$) [4] *Tuples* $\mathbf{i} = (i_1, \ldots, i_n)$ *and* $\mathbf{j} = (j_1, \ldots, j_n)$ *satisfy* $\mathbf{i} < \mathbf{j}$ *iff there exists an integer* k, $1 \leq k \leq n$ *such that* $i_1 = j_1, \ldots, i_{k-1} = j_{k-1}$ *and* $i_k < j_k$.

Definition 7 (Dependence) *An iteration* $\mathbf{j} \in \mathcal{I}$ *is dependent on iteration* $\mathbf{i} \in \mathcal{I}$, *denoted* $\mathbf{i}\ \delta\ \mathbf{j}$, *iff there exists an element of an array assigned to in* $\mathbf{i}$ *and referenced in* $\mathbf{j}$, *such that* $\mathbf{i} < \mathbf{j}$ *and there is no* $\mathbf{k}$, $\mathbf{i} < \mathbf{k} < \mathbf{j}$ *with* $\mathbf{k}\ \delta\ \mathbf{j}$ *for the same array element.*

Given iterations $\mathbf{i}$ and $\mathbf{j}$, two references to an m-dimensional array A, $\mathsf{A}(f_1(\mathbf{i}), ..., f_m(\mathbf{i}))$ and $\mathsf{A}(g_1(\mathbf{j}), ..., g_m(\mathbf{j}))$ are to the same element iff $f_1(\mathbf{i}) = g_1(\mathbf{j}), ..., f_m(\mathbf{i}) = g_m(\mathbf{j})$. For

[4] Note that the symbol $<$ is used to compare numbers as well as to compare tuples. Although, this may be some what confusing at first, it simplifies the notation greatly, and the intended meaning should be clear from the context.

conciseness we denote this set of equalities by $F(\mathbf{i}) = G(\mathbf{j})$. The dependence problem can then be stated as follows: Do there exist iterations $\mathbf{i}$ and $\mathbf{j}$ in the iteration space such that $F(\mathbf{i}) = G(\mathbf{j})$? In other words, we need to know whether the following integer programming problem has solutions:

$$S\,\mathbf{i} \geq \mathbf{c}$$

$$S\,\mathbf{j} \geq \mathbf{c}$$

$$F(\mathbf{i}) = G(\mathbf{j})$$

Clearly, the above problem is NP-complete. Moreover, for restructuring purposes we need more information than the mere existence of a dependence. For example, we need to know all the pairs of iterations that are dependent and the precise relationship between them.

Definition 8 (Dependence Distance Vector) *For a pair of iterations* $\mathbf{i} = (i_1, \ldots, i_n)$ *and* $\mathbf{j} = (j_1, \ldots, j_n)$ *such that* $\mathbf{i}\ \delta\ \mathbf{j}$, *the vector* $\mathbf{j} - \mathbf{i} = (j_1 - i_1, ..., j_n - i_n)$ *is called the dependence distance vector.*

In this document "dependence" refers to a dependence distance vector unless mentioned otherwise. The dependence distance vector is called a *constant* or *uniform dependence* if it contains only constants. For example, $(1, 2)$ is a constant dependence. In contrast, $(i, 0)$, is a *linear dependence.*

Definition 9 (Dependence Direction) *For a pair of iterations* $\mathbf{i} = (i_1, \ldots, i_n)$ *and* $\mathbf{j} = (j_1, \ldots, j_n)$, $sign(\mathbf{j} - \mathbf{i}) = (sign(j_1 - i_1), ..., sign(j_n - i_n))$ *is called the dependence direction vector, where* $sign(a) \in \{1, 0, -1\}$ *denotes whether* a *is positive, zero or negative.*

A dependence $(d_1, .., d_n)$ is positive if the first nonzero element d_k is positive, and we say that the dependence is carried by the k^{th} loop. The sequential semantics of a loop suggests that a flow dependence is always positive.[5] The *Dependence matrix*, D, is an n by m integer matrix, where n is the dimension of the nested loop and m is the number of dependences. Each column of D contains a dependence distance vector Di.

Example 2

```
for I1 = 0, 5
  for I2 = 0, 5
    A(I1, I2) = A(I1 − 1, I2) + A(I1, I2 − 1) + A(I1 − 2, I2 + 1)
  end for
end for
```

There are 3 dependences corresponding to 3 pairs of reads and the writes to array A : $[A(I_1, I_2), A(I_1 - 1, I_2)]$, $[A(I_1, I_2), A(I1, I_2 - 1)]$ and $[A(I_1, I_2), A(I_1 - 2, I_2 + 1)]$. Consider the third pair of references.

[5] This property determines the *validity* of a loop transformation as discussed later in the document.

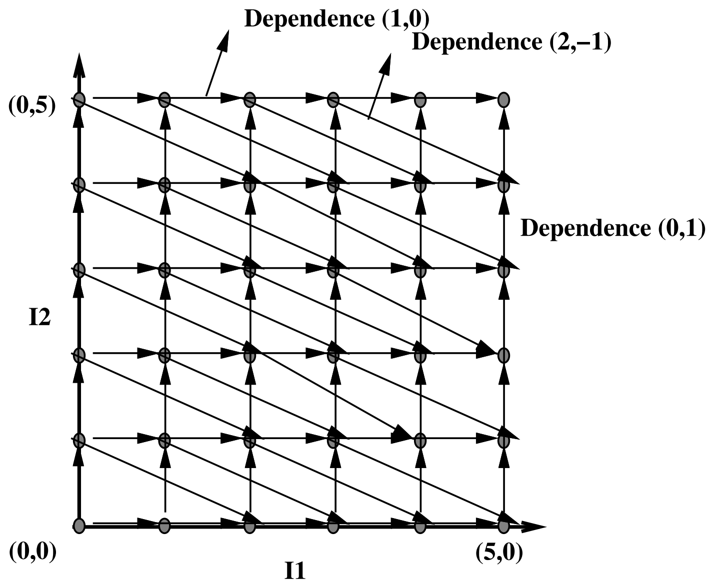

Figure 4: Dependences in iteration space

With iterations $\mathbf{i} = (i_1, i_2)$ and $\mathbf{j} = (j_1, j_2)$ such that $\mathbf{i} < \mathbf{j}$, if $i_1 = j_1 - 2$ and $i_2 = j_2 + 1$ then $\mathbf{i}\ \delta\ \mathbf{j}$. Therefore, the dependence distance and direction vectors are, $(j_1 - i_1, j_2 - i_2) = (2, -1)$ and $(+1, -1)$ respectively. The other dependence distances are $(1, 0)$ and $(0, 1)$ (with directions $(+1, 0)$, $(0, +1)$ respectively). The dependence matrix therefore is

$$D = \begin{bmatrix} 1 & 0 & 2 \\ 0 & 1 & -1 \end{bmatrix}$$

Figure 4 shows the dependences in the iteration space.

Because the problem of determining the existence of a dependence is NP-complete, one often employs efficient algorithms that solve restricted cases or provide approximate solutions in practice. GCD test [Ban88] finds the existence of an integer solution to the dependence problem with only a single subscript in an unbounded iteration space. Some algorithms find real valued solutions in bounded iteration spaces and dependence direction information [LYZ90, WT92]. Recently, Pugh noted that integer programming solutions, with exponential worst case complexity has much lower average complexity [Pug92]. A modified *Fourier-Motzkin* variable elimination [Sch86] produces exact solutions in a reasonable amount of time for many problems.

In the next section we discuss the loop transformation techniques assume that dependence distance information is available.

3 Linear Loop Transformations

The objective of a loop transformation is to change the loop to a new one that exploits the architectural features of the hardware, while still computing the same result. Some of the objectives in transforming a loop are: increasing parallelism at the desired loop level, minimizing the size of sequential loop level, improving load balance and static locality, and grouping reuse of a variable for dynamic locality. Many loop transformations exist, including *loop interchange*, *loop permutation*, *skew*, *reversal*, *wavefront*, and *tiling*[6]

Loop interchange, as the name suggests, exchanges two loop levels. In a doubly nested loop, a loop interchange can expose parallelism at the inner loop level enabling vectorization or it can expose parallelism at the outer loop. Loop permutation is a generalization of loop interchange and corresponds to a sequence of interchanges of loop levels.

Wavefront identifies sets of independent iterations and restructures the loop to execute these sets sequentially. All the available parallelism is thus at inner loop level which achieves finer grain of parallelism. Loop skewing, parameterized by a skew factor, is an instance of wavefront transformation.

Tiling *strips* different loop levels so that outer loops step through blocks of iterations, and inner loops step through iterations in these blocks. Tiling can reduce the size of a loop to a given vector register size. Since the block size is smaller than the original loop size, the chances are that reused variable still exists in the cache. Typically, another transformation prior to tiling is performed so that all the reuse occurs in inner loop levels. Thus tiling can improve the cache hit rate.[7]

Computing a loop transformation involves a derivation of the references and loop bounds for the transformed loop, and test of *legality*. A legal transformation guarantees that the transformed loop produces the same results as the original one. Accordingly, techniques have been developed to compute the transformed loop for each individual transformation, and to test their legality.

To fully exploit the features of complex architectures it is generally necessary to apply compound transformations, a combination of two or more transformations. Viewing compound transformations as sequences of simpler transformations has its problems, and identifying a legal sequence of transformations that constitutes the compound transformation involves search in a space that can be very large. For example, finding a sequence of interchanges for a desired permutation of a loop requires a search in a space of all permutations of the loop; a loop skew has infinite instances based on the skew factor. It is difficult to characterize the optimality of a sequence of transformations with respect to a particular goal. Thus, pruning the search space by evaluating the contribution of a subsequence of transformations to over all "goodness" is not easy. Even when we find the right sequence of transformations, applying them one by one can produce complex loop bounds.

A loop transformation can be viewed as a reorganization of the loop iterations. It can

[6] The accompanying *catalog of loop transformations* provides details on these transformations, and their implications.

[7] The number of times an element of an array is reused remains the same after any valid transformation. The enhancement we refer to here is the effective reuse taking cache and array sizes into consideration.

thus be characterized by a transformation of the iteration space. Unimodular transformations [Ban90, WL90, Dow90, KKBP91, KKB91] are linear transformations on iteration spaces that give a unified view of all possible sequences of most of the existing transformations. A unimodular matrix[8] determines the transformed loop and dependences; encapsulates the objective function for "goodness" of the transformation (in some direct or indirect way); and in conjunction with the dependences provides legality criterion. The unimodularity of the transformation matrix ensures that the mapping is one to one and onto with unit stride. Although, the approach itself does not solve the problem of deriving the optimal transformation for a given loop, there exist techniques to achieve some of the objectives. Further elaboration on this aspect and the advantages of the unified approach are discussed in a following subsection.

3.1 General Technique

A unimodular transformation is defined by an n by n unimodular integer matrix U. U maps the iteration vector $\mathbf{I}$ into a new iteration vector $\mathbf{K} = (K_1, ...K_n)^T$,

$$U\,\mathbf{I} = \mathbf{K}$$

and maps each dependence vector D_i into a new vector D_i':

$$U D_i = D_i'$$

The transformation is legal iff D_i' for all D_i are lexicographically positive. The positivity of transformed dependences is a strong condition, and ensures a valid sequential execution of the transformed loop. In theory, it is possible to schedule a transformed loop with negative dependences, unless the dependences lead to a deadlock (that is, there is a cycle of dependences in the transformed iteration space).

To perform the loop transformation, the references in the original loop must be substituted with new ones, and new loop bounds must be derived. Each reference in the body of the loop can be replaced by substituting the original iteration indices with the equivalent expressions in terms of the new iteration indices using the following equation:

$$U^{-1}\,\mathbf{K} = \mathbf{I}$$

Determining the new loop bounds is nontrivial. We know from the bound matrix that

$$S\,\mathbf{I} \geq \mathbf{c}$$

So,

$$S\,U^{-1}\,U\,\mathbf{I} \geq \mathbf{c}$$

and

$$S\,U^{-1}\,\mathbf{K} \geq \mathbf{c}$$

[8] A matrix with determinant ± 1.

This set of inequalities describes a convex polyhedron, and the sought after loop bounds are the set of affine functions specify the polyhedron. If $S' = S\ U^{-1}$ is lower triangular then it is clear that the new loop bounds can be directly obtained from the rows of S'. In general, variable elimination techniques such as *Fourier-Motzkin*[Sch86] has to be applied on S' to obtain the new loop bounds. Suppose α's and β's are linear expressions in $K_1, ..., K_{n-1}$, and a's and b's are constants. The two sets of inequalities, $\beta \le bK_n$ and $aK_n \ge \alpha$ provide the lower bound for K_n $max(\lceil \alpha/a \rceil)$ and upper bound $min(\lfloor \beta/b \rfloor)$. Elimination of K_n gives us the sets of inequalities $a\beta \le b\alpha$ that do not have K_n in them. By rearranging this new set of inequalities, we can get bounds for K_{n-1} as functions of $K1, ..., K_{n-2}$ in a similar way. Continuing in a similar way we get bounds for K_1 as constants. The elimination performs only a real projection, that is, after elimination of say K_n, there can be integer solutions to the resulting inequalities in $n-1$ variables that do not correspond to solutions in the original polyhedron.

When the transformation is unimodular the above approach results in the transformed space that corresponds exactly to the original space. The unimodularity of the transformation matrix ensures that the stride in the transformed loop is unit as well. Banerjee[Ban90] computes the bounds for a two dimensional loop directly. Kumar and Kulkarni [KKB91] compute the bounds for a loop of any dimension by transforming the bounding hyper planes in the original loop and substituting for the extremes points of the polyhedron. The approach is elegant when the bounds are constants, and becomes complex when the original loop bounds are linear expressions. All of the above methods fail to produce exact bounds when the transformation is not unimodular. Ramanujam[Ram92a] and Kumar[Kum93] compute the new bounds for any non-singular transformation by *stepping aside* the non-existent iterations. For illustrative purposes, we compute the new bounds for a two dimensional loop directly in the following example. Consider the following loop.

```
for  I1 = n1, N1
  for  I2 = n2, N2
    H(I1, I2)
  end for
end for
```

$(I_1, I_2)^T$ is the iteration vector. Suppose the unimodular matrix U

$$U = \begin{bmatrix} u_{11} & u_{12} \\ u_{21} & u_{22} \end{bmatrix}$$

transforms the loop into the following new loop.

```
for  K1 = m1, M1
  for  K2 = m2(K1), M2(K1)
    H(f(K1, K2), g(K1, K2))
```

```
    end for
  end for
```

where $(K_1, K_2)^T$ is the new iteration vector,

$$(K_1, K_2)^T \;= U\;(I_1, I_2)^T \;= (u_{11}I_1 + u_{12}I_2, u_{21}I_1 + u_{22}I_2)$$

and $m1$ and $M1$ are constants, $m2()$ and $M2()$ are functions of K_1, and $f()$ and $g()$ are functions of K_1 and K_2.

With $\Delta = det(U) = \pm 1$, and with Δu_{ij} denoting $det(U) * u_{ij}$, we know that U^{-1} is

$$U^{-1} = \begin{bmatrix} \Delta u_{22} & -\Delta u_{12} \\ -\Delta u_{21} & \Delta u_{11} \end{bmatrix}$$

Hence,

$$(I_1, I_2)^T \;= U^{-1}\;(K_1, K_2)^T \;= (\Delta u_{22}K_1 - \Delta u_{12}K_2, -\Delta u_{21}K_1 + \Delta u_{11}K_2)$$

This enables us to replace the references to I_1 and I_2 in the body of the loop with $\Delta u_{22}K_1 - \Delta u_{12}K_2$ and I_2 by $-\Delta u_{21}K_1 + \Delta u_{11}K_2$ respectively.

The bounds for K_1 and K_2 can be computed directly as follows. The bounds of K_1 are constants, and can be derived from the expression :

$$K_1 = u_{11}I_1 + u_{12}I_2$$

and the upper and lower bounds of I_1 and I_2.

$$m1 = min(u_{11}I_1 + u_{12}I_2, n_1 \leq I_1 \leq N_1, n_2 \leq I_2 \leq N_2)$$

$$M1 = max(u_{11}I_1 + u_{12}I_2, n_1 \leq I_1 \leq N_1, n_2 \leq I_2 \leq N_2)$$

Introducing the notation $a^+ = max(a, 0)$ and $a^- = max(-a, 0)$, we can write the above as

$$m1 = u_{11}^+ n_1 - u_{11}^- N_1 + u_{12}^+ n_2 - u_{12}^- N_2$$

$$M1 = u_{11}^+ N_1 - u_{11}^- n_1 + u_{12}^+ N_2 - u_{12}^- n_2$$

(since either u_{ij}^+ or u_{ij}^- will always be equal to zero)

It is possible to compute the lower and upper bounds of K_2 as follows. Because $n_1 \leq I_1 \leq N_1$ and $n_2 \leq I_2 \leq N_2$,

$$n_1 \leq \Delta u_{22}K_1 - \Delta u_{12}K_2 \leq N_1$$

$$n_2 \leq -\Delta u_{21}K_1 + \Delta u_{11}K_2 \leq N_2$$

by substituting equations for I_1 and I_2. This is is equivalent to

$$\frac{n_1 - \Delta u_{22} K_1}{-\Delta u_{12}} \leq K_2 \leq \frac{N_1 - \Delta u_{22} K_1}{-\Delta u_{12}}$$

$$\frac{n_2 + \Delta u_{21} K_1}{\Delta u_{11}} \leq K_2 \leq \frac{N_2 + \Delta u_{21} K_1}{\Delta u_{11}}$$

which gives us two lower bounds, $lb1$ and $lb2$, and two upper bounds, $ub1$ and $ub2$, for K_2 (depending on the signs of $-\Delta u_{12}$ and Δu_{11}). Thus the lower and upper bounds for K_2 are

$$m2(K_1) = \lceil\ max(lb1, lb2)\ \rceil \text{ and } M2(K_1) = \lfloor\ min(ub1, ub2)\ \rfloor$$

Example 3

```
for  I1 = 0, 10
   for  I2 = 0, 10
      A(I1, I2) = A(I1 − 1, I2) + A(I1, I2 − 1) + A(I1 − 2, I2 + 1)
   end for
end for
```

The dependence matrix for this nested loop is $D = \{(1,0),(0,1),(2,-1)\}$
If U defines our transformation,

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

then

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

$$I_1 = K_1 - K_2, I_2 = K_2$$

and

$$D' = \{(1,0),(1,1),(1,-1)\}$$

(All the transformed dependences are lexicographically positive, so U is a legal transformation.) The bounds for K_1 are then:

$$\begin{aligned} m1 &= u_{11}^{+} n_1 - u_{11}^{-} N_1 + u_{12}^{+} n_2 - u_{12}^{-} N_2 \\ &= 1 * 0 + 1 * 0 = 0 \end{aligned}$$

and

$$\begin{aligned} M1 &= u_{11}^{+} N_1 - u_{11}^{-} n_1 + u_{12}^{+} N_2 - u_{12}^{-} n_2 \\ &= 1 * 10 + 1 * 10 = 20 \end{aligned}$$

To compute the bounds of K_2, note that $-\Delta u_{12} = -1 * 1 = -1 < 0$ and $\Delta u_{11} = 1 * 1 > 0$, so

$$ub1 = \frac{(n_1 - \Delta u_{22} K_1)}{-\Delta u_{12}} = (0 - K_1)/-1 = K_1$$

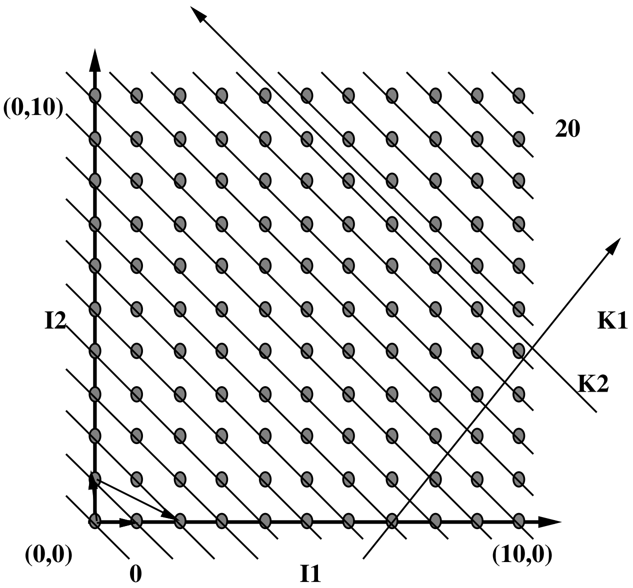

Figure 5: Example Transformation

$$
\begin{aligned}
lb1 &= \frac{(N_1 - \Delta u_{22} K_1)}{-\Delta u_{12}} = (10 - K_1)/-1 = K_1 - 10 \\
lb2 &= \frac{(n_2 + \Delta u_{21} K_1)}{\Delta u_{11}} = (0+0)/1 = 0 \\
ub2 &= \frac{(N_2 + \Delta u_{21} K_1)}{\Delta u_{11}} = 10/1 = 10
\end{aligned}
$$

The transformed loop is therefore

```
for K1 = 0, 20
   for K2 = max(0, K1 − 10), min(10, K1)
      A(K1 − K2, K2) = A(K1 − K2 − 1, K2) + A(K1 − K2, K2 − 1)
            +A(K1 − K2 − 2, K2 + 1)
   end for
end for
```

Figure 5 depicts the transformation. Notice that the iterations along the K_2 axis are all independent and can be executed in parallel.

□

To illustrate the computation of the new loop bounds with the Fourier-Motzkin variable elimination method, consider the description of the original iteration space in terms of the bound matrix S.

$$S\,\mathbf{I} \geq \mathbf{c}$$

$$\left[\begin{array}{rr} 1 & 0 \\ 0 & 1 \\ \hline -1 & 0 \\ 0 & -1 \end{array}\right] \left[\begin{array}{c} I_1 \\ I_2 \end{array}\right] \geq \left[\begin{array}{r} 0 \\ 0 \\ \hline -10 \\ -10 \end{array}\right]$$

Because,

$$\mathbf{S}\ U^{-1}\ \mathbf{K}\ \geq\ \mathbf{c}$$

which gives the new bound matrix $\mathbf{S}\ U^{-1}$.

$$\left[\begin{array}{rr} 1 & -1 \\ 0 & 1 \\ \hline -1 & 1 \\ 0 & -1 \end{array}\right] \left[\begin{array}{c} K_1 \\ K_2 \end{array}\right] \geq \left[\begin{array}{r} 0 \\ 0 \\ \hline -10 \\ -10 \end{array}\right]$$

Since the new bound matrix is not in lower triangular form, we apply the variable elimination method in the following way. From the new set of inequalities, it is clear that,

$$0 \leq K_2 \leq 10 \text{ and } K_1 - 10 \leq K_2 \leq K_1$$

Thus the bounds for K_2

$$K_2 \geq max(K_1 - 10, 0) \text{ and } K_2 \leq min(K_1, 10)$$

Once K_2 is eliminated the above inequalities provide the following projections on K_1. $K_1 - 10 \leq 10$, $0 \leq K_1$, $0 \leq 10$, and $K_1 \leq K_1 + 10$. Ignoring the redundant constraints we have constant bounds for K_1.

$$K_1 \geq 0 \text{ and } K_1 \leq 20$$

□

3.2 Advantages

A unimodular transformation can be a primary transformation such as an interchange or it can be a combination of two or more primary transformations. Figure 6 shows some of the possible primary transformations for the two dimensional case. The transformation matrix for a given permutation is just a permuted identity matrix. A transformation matrix that reverses the k^{th} loop level is an identity matrix with k^{th} row multiplied by -1.

A compound transformation is represented by a matrix which results from the multiplication of the component transformations in the opposite order in which the transformations are applied. The legality of the compound transformation, the new loop bounds, and the references are determined in one step at the end rather than doing so for each of the constituent transformations. The computation of the transformed loop structure is done in the same systematic way irrespective of the transformation employed.

The overall goal of a transformation is to exploit the architectural features of the hardware. The "goodness" of a transformation can be specified in terms of certain aspects of the transformed loop, such as, parallelism at outer (inner) loop level, volume of communica-

$$\underset{(a)}{\begin{bmatrix} -1 & 0 \\ 0 & 1 \end{bmatrix}} \quad \underset{(b)}{\begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix}} \quad \underset{(c)}{\begin{bmatrix} -1 & 0 \\ 0 & -1 \end{bmatrix}}$$

$$\underset{(d)}{\begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}} \quad \underset{(e)}{\begin{bmatrix} 1 & p \\ 0 & 1 \end{bmatrix}} \quad \underset{(f)}{\begin{bmatrix} 1 & 0 \\ p & 1 \end{bmatrix}} \quad \underset{(g)}{\begin{bmatrix} 1 & 1 \\ 1 & 2 \end{bmatrix}}$$

The transformation matrices for (a) reversal of outer loop, (b) reversal of inner loop, (c) reversal of both loops, (d) interchange, (e,f) skew by p in second and first dimensions, and (g) wavefront respectively.

Figure 6: Example 2d transformations

tion, the average load, and load balances. In the conventional approach, where some primary transformation like interchange, or an *a priori* sequence of such transformations is applied, there is no way of evaluating how good a transformation is. On the other hand, a unimodular matrix completely characterizes the transformed loop, and hence the goodness criterion can be specified in quantitative terms. The parallelism at outer (inner) loop level, volume of communication, the average load, and load balances for the transformed loop can be specified in terms of the elements of the transformation matrix, dependences, and original loop bounds [KKBP91].

For example, we may want to find a transformation that minimizes the size of the outer loop level, because it is sequential. The first row of the transformation matrix in conjunction with the original loop bounds gives a measure of the size of the outer loop in the transformed loop. This function provides us with the goodness of a candidate transformation.

As another example, we may desire that most of the dependences be independent of the inner loop levels.[9] The dependences in the transformed loop can be expressed in terms of the original dependences and the elements of the transformation matrix.

Finally, suppose the outer most level of a transformed loop is to be executed in parallel. Using the transformation matrix elements and the original loop bounds we have a way of establishing the load imbalance – the variance of the number of iterations in each instance of the outer loop.

Unfortunately, the derivation of a unimodular matrix that satisfies the desired requirements is hard in general. The problem is NP-complete for unrestricted loops, and even affine loops with non-constant dependence distances [Dow90]. A unimodular matrix can however be found in polynomial time for affine loops with only constant dependences [Dow90, KKB91].

A dependence matrix can provide a good indication as to the desired transformations. In fact, it is common to start with a dependence matrix augmented with the identity ma-

[9] In a loop with only constant dependences, it is possible to make all dependences independent of the inner loop.

trix. The transformations sought are then those that result in dependences with a particular form – for example, no dependences within a particular loop level. Proofs on the existence of a transformation that achieves certain goals tend to be constructive, and by themselves provide algorithms to derive the transformation. In the following subsections we discuss a few instances of identifying the structure of a transformation given specific goals for the transformed loop.

As discussed in a later section, it is an open question as to whether it is possible to derive a transformation that provides the best trade-off parallelism, locality, and load balance.

Although a unimodular transformation has its advantages, a loop transformation can be represented by any invertible integer matrix[LP92, Ram92a, Kum93]. A non-unimodular linear transformation is more general does not result in unit strides, and computing the transformed loop bounds is more involved.[10]

3.3 Internalization

Consider the execution of a nested loop on a hierarchical memory architecture, where non-uniform memory access times exist. Suppose we partition the iterations in the loop into different groups where each group is executed in parallel. The degree of parallelism is the number of such groups. Suppose that the data to be computed resides on the processor computing it, and that the read-only data is replicated.[11] The dependences that exist between iterations that belong to different groups result in non-local accesses. The dependences that exist between iterations in the same group result in local accesses.

Let us now consider the restricted case where we intend to execute every instance of the outermost loop in parallel (assuming we have sufficient number of processors to do so), and that each group is executed purely sequential. The amount of parallelism is the size of the outer loop. Any dependences carried by the outer loop result in non-local accesses.

Internalization [KKBP91, KKB91] transforms a loop so that as many dependences as possible are independent of the outer loop, and so that the outer loop is as large as possible. For example, consider the two dimensional loop:

```
for i = 0, n
  for j = 0, n
    A(i, j) = A(i - 1, j - 1)
  end for
end for
```

This nested loop has a dependence of $(1, 1)$, and if we transform the loop so that the dependence is not carried by the outer level, that is, transform the dependence into $(0, 1)$, then we obtain outer loop iterations that are fully independent. One unimodular matrix U

[10] Due to the non-unit strides there is a need for techniques to step aside from non-existent iterations, and apply offsets to the bounds computed from the variable elimination algorithms.

[11] In other words, we follow the *ownership rule* to distribute the data.

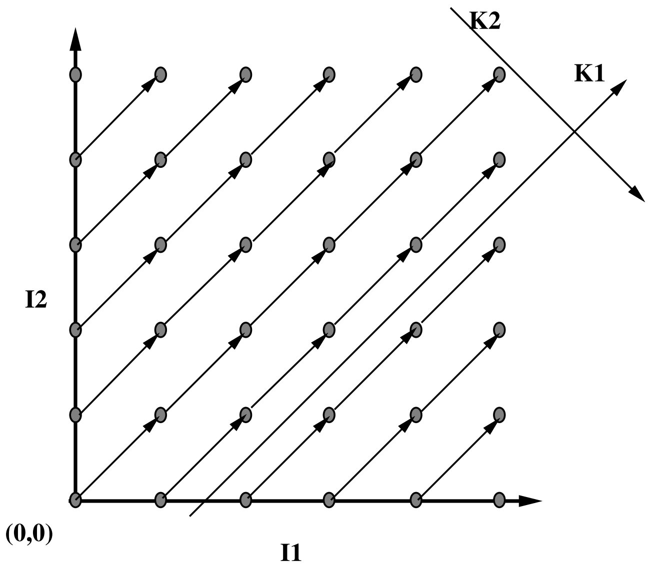

Figure 7: Internalization in 2d iteration space

(of many) that internalizes $(1, 1)$ is:

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

The transformed loop then is:

```
for K1 = −n, n
  for K2 = max(0, −K1), min(n, n − K1)
    A(K1 + K2, K2) = A(K1 + K2 − 1, K2 − 1)
  end for
end for
```

Figure 7 shows the transformed iteration space.

In the general case, let U be a unimodular matrix that internalizes a dependence (a, b):

$$U = \begin{bmatrix} u_{11} & u_{12} \\ u_{21} & u_{22} \end{bmatrix}$$

Suppose the desired transformed dependence is (a', b') with $a' = 0$ and $b' > 0$. Then we have

$$\begin{aligned} a' &= au_{11} + bu_{12} = 0 \\ b' &= au_{21} + bu_{22} > 0 \end{aligned}$$

$$det(U) \;=\; \Delta = u_{11}u_{22} - u_{12}u_{21} = \pm 1$$

$u_{11} = kb$ and $u_{12} = -ka$ is a solution for some integer k. Substituting in $det(U)$ we see that

$$\Delta = bu_{22} + au_{21} = \pm 1/k$$

Since the left hand side is an integer, $k = \pm 1$, so u_{21} and u_{22} can be chosen accordingly.

Guaranteeing the positivity of the transformed dependences is conceptually the same as choosing dependences that form the face of a convex hull constructed from the origin and the points corresponding to different dependence distances. The first row of U is the normal to this face, and hence perpendicular to $n-1$ linearly independent dependences; the second row is chosen to be normal to $n-2$ dependences so on. Further details can be found in [KKB91].

One can only internalize $n-1$ linearly independent dependences in an n-dimensional loop. The choice of dependences to internalize has an impact on such factors as the validity of the transformation, the size of the outer level in the transformed loop, the load balance, locality etc. Kulkarni and Kumar [KKBP91] introduced the notion of *weight* to a dependence to characterize the net volume of non-local accesses. They also provided metrics for parallelism and load imbalance for the two dimensional case. Internalization can be further generalized to mapping with multiple parallel levels [KKB92]. Locality can be improved by internalizing a dependence or a reference with reuse. In other words, internalization is a transformation that enhances parallelism and locality[KSa].

3.4 Access Normalization

Ideally, we want a processor to own all the data it needs in the course of its computation. In that case we wish to transform a loop so that it matches the existing layout of the data in the memory of the parallel system[12]. For example, suppose processors own complete columns of matrices A and B, and suppose each iteration of the following outer loop is executed in parallel:

```
for  i = 0, N1 − 1
   for  j = i, i + b − 1
      for  k = 0, N2 − 1
         B(i, j − i) = B(i, j − i) + A(i, j + k)
      end for
   end for
end for
```

Since a processor needs to access the rows of each matrix, a large number of non-local accesses. If it is possible to transform the loop so as to reflect the data owned by each processor, then the number of remote accesses will be reduced significantly. For this we need the references

[12] This is referred to as data distribution an will be considered in detail in a later section

to the second dimension to be the outer loop index in the transformed loop.

Three different access patterns that appear in the above nested loop can be represented by a matrix-vector pair as below.

$$\begin{bmatrix} -1 & 1 & 0 \\ 0 & 1 & 1 \\ 1 & 0 & 0 \end{bmatrix} \begin{bmatrix} i \\ j \\ k \end{bmatrix} = \begin{bmatrix} j-i \\ j+k \\ i \end{bmatrix}$$

This is called an access matrix. It is interesting to note that the access matrix itself can be a transformation matrix. The transformed loop would be

```
for u = p, b − 1, P
  for v = u, u + N1 + N2 − 2
    for w = 0, N1 − 1
      B(w, u) = B(w, u) + A(w, v)
    end for
  end for
end for
```

All accesses to B now become local, although A still has some non-local accesses.

To be a valid transformation the access matrix has to be invertible. The techniques to make the access matrix invertible do so at the cost of reduced normalization [LP92].

3.5 Other relevant work

The *hyperplane* or *wavefront* method suggested by Lamport [Lam74] to parallelize **do** loops is the most influential early work. This method attempts to find the optimum family of hyperplanes of parallelism in the iteration space, maximizing the number of iteration points on each hyperplane and minimizing the inter plane dependences. The resulting transformation essentially achieves inner loop parallelism with the hyperplanes corresponding to each of the outer loop instances executing strictly sequentially. The method is however not directly suited to the far more general objective we are pursuing.

Until recently, the various loop transformations such as reversal, skewing, permutations, and inner/outer loop parallelization were handled independently, and determining which loop transformations to apply to a given nested loop and in what order remained an open issue. The concept of unimodular transformations for loop partitioning were introduced independently by Banerjee [Ban90], and by Wolf and Lam [WL90].

Certain significant existence results concerning candidate and optimal unimodular transformations were presented by Dowling [Dow90]. The scope of his work is limited to transformations for inner loop parallelism in affine nested loops. The results are essentially existential and not constructive. Further, the work adopts a dynamic notion of dependence vectors which brings the size of the loop into the picture while addressing every complexity result. Such a dynamic view is however known to be unnecessary while dealing with uniform and linear dependence vectors.

A number of researchers [D'H89, IT88, PC87, SF88] look for completely independent partitions in the iteration space of a nested loop. However, the number of completely independent partitions inherent in most situations are rather low and hence this method usually results in very little parallelism.

4 Tiling

A related method in loop transformation, which has received considerable attention lately, is the grouping together iterations to obtain smaller iteration spaces. These techniques, which have been termed *tiling* [RS90, WL91, IT88] or *grouping* [KN89], serve the purpose of combining the communication associated with each group in order to save on the overhead for each communications. Tiling also results in smaller and less complex iteration spaces for further partitioning, and can improve the cache performance by improving the amount of variable reuse.

The following example illustrates the tiling of a two dimensional loop.

Example 4 A two dimensional loop:

```
for i = 0, n
  for j = 0, n
    A(i,j) = A(i − 1,j) + B(i,j)
  end for
end for
```

tiled along the directions of the i and j axis with sizes S_i and S_j is:

```
for ii = 0, n, S_i
  for jj = 0, n, S_j
    for i = ii, min(n, ii + S_i − 1)
      for j = jj, min(n, jj + S_j − 1)
        A(i,j) = A(i − 1,j) + B(i,j)
      end for
    end for
  end for
end for
```

The transformed loop iteration space is depicted in Figure 8.

It is useful to think of the tiles as being delineated by a family of hyperplanes that define the tiles. Consider a two dimensional loop which we intend to tile by tiles of two sets of parallel edges. We can specify the edges of the tiles by two families of lines that are perpendiculars to the edges. These lines define the orientation of the tiles, and the spacing between the lines define the size of each tile side (except for the boundaries).

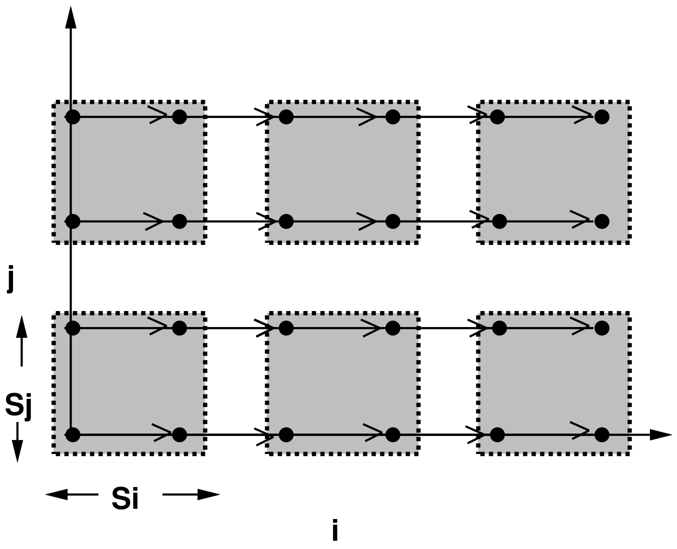

Figure 8: Tiling in two dimensions

Another way of defining tiles in a, say two dimensional iteration space, is to specify the tile origins by two lines and define the boundaries for each tile by inequalities.

The tiles should be atomic; that is, synchronizations should not be necessary when computing a tile. This enables us to view the tiles as abstractions of iterations. Inter-tile dependences result in remote accesses. The choice of tile orientation has an influence on the atomicity and number of remote accesses. The choice of tile size is often determined by the cache size. The tile size can also be tuned to optimize the size of remote data transfers or the the size of the vector registers. Determining the loop bounds of a tiled loop is non-trivial and are usually realized either by conditionals inside the loop body [IT88] or by shifts [Ram92b, Kum93].

5 Data Transformations

Loop performance is dependent as much on data organization as it is on code organization. So far we have confined ourselves to reorganizing loop iteration spaces. In this section, we concentrate on reorganizing array structures. In hierarchical memory architectures there is a natural non-uniformity in the cost of accessing data, and the non-uniformity is a function of the levels of the hierarchy. The location of data is crucial to performance. Ideally, we wish to have the data local to the processor that needs it, so that the overhead due to access latency is minimal. The array reference patterns constrain (in often incompatible ways) the relative placement or *alignment* of the array elements and their distribution on available processors. In this section, we consider transformations on arrays to achieve the alignment and to distribute the data for reduced communication overhead.

5.1 Data Alignment

5.1.1 Basic Idea

The reference patterns in a loop body provide information on the amount of data movement required for the computation. The objective in aligning data is to improve the lower bound on the locality irrespective of the data distribution that could be employed.

Definition 10 (Rectangular Data space) *Given an n-dimensional array* A*, $\mathcal{A} \subseteq \mathbf{Z}^n$ such that*

$$\mathcal{A} = \{(i_1, ..., i_n) \mid l_1 \leq i_1 \leq L_1, ..., l_n \leq i_n \leq L_n\},$$

is a rectangular data space of A*, where i_j is the index, and l_j and L_j are the lower and upper bounds of the j^{th} dimension of* A*.*

This above definition characterizes an array as a rectangular integer space. For example, an n by n array A corresponds to an integer space (assuming array indexing starts at 0)

$$\mathcal{A} = \{(i_1, i_2) \mid 0 \leq i_1 \leq n, 0 \leq i_2 \leq n\}$$

Definition 11 (Alignment) *The alignment of an n dimensional array* B*, to an $m \geq n$ dimensional array* A *is a function $f_a : \mathcal{B} \rightarrow \mathcal{A}$, where $\mathcal{A}$ and $\mathcal{B}$ are the data spaces of arrays* A *and* B*.*

An alignment from array B to array A, maps the elements of array B to the elements of array A. As we see in the following paragraphs an identical distribution of two arrays can result in reduced communications. If we consider each element of array A to be on a different processor, then the alignment determines the elements of A and B that go on the same processor. If we assume the familiar *ownership-rule*[13] then, alignment implicitly determines the placement of computation in an iteration onto a processor.

We want to consider all alignment constraints in the program and select an alignment for each pair of arrays. For example, in Figure 9 the reference patterns for arrays A and B suggest a data movement of B in the direction of $(1, 1)$. Assuming that each element of an array is placed on a different processor, and that both the arrays are placed identically, there are $O(n^2)$ non-local references in the loop. The reference pattern constrains the alignment of B to be a shift in the direction $(1, 1)$. This alignment makes all the references to B local, for any identical distribution for the arrays.

The basic idea in finding a pairwise alignment transformation is to find the hyperplane in the iteration space along which elements of the arrays to be aligned are similarly indexed, and construct a transformation on the data space of one of the arrays such that the size of that hyperplane is maximized.

5.1.2 General Method

We consider references occurring in statements of the body of an m-dimensional loop with **J** as the iteration vector. We confine ourselves to affine references and provide a representation,

[13] The processor that owns the lhs in a statement computes it.

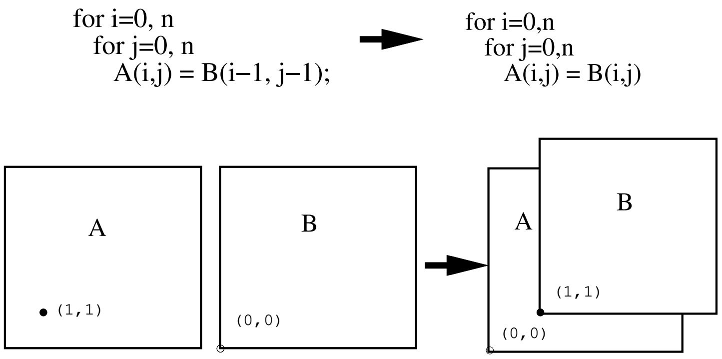

Figure 9: Basic Idea behind Data Alignment

pairwise alignment criteria, and general technique for alignment. The representation closely follows [OH92] being a good algebraic representation.

For notational ease we consider a reference as though it were a column vector instead of a row vector. Let the dimension of an array A be N_A. For the moment let N_A be same as m, the dimension of the loop. An affine reference to array A can be represented by

$$F_A(\ \mathbf{J}\ ) = R\,\mathbf{J}\ +\ \mathbf{c}$$

where, R is an m by m integer matrix, $\mathbf{J}$ is the m by 1 iteration vector, and $\mathbf{c}$ is a m by 1 integer vector. R is called an *occurrence matrix.*

Example 5 Consider, the following loop.

```
for  i = 1, n
   for  j = 1, n
      A(i, j) = B(i − 1, j − 2)
   end for
end for
```

The references for A and B are

$$F_A(\ \mathbf{J}\ ) = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} i \\ j \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

$$F_B(\ \mathbf{J}\ ) = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} i \\ j \end{bmatrix} + \begin{bmatrix} -1 \\ -2 \end{bmatrix}$$

respectively.

It is not uncommon to have the dimension of the loop and the dimension of an array different. Moreover, the dimensions of the arrays referenced inside the loop can differ. To have an uniform representation for all references inside a loop, and to enable alignment between

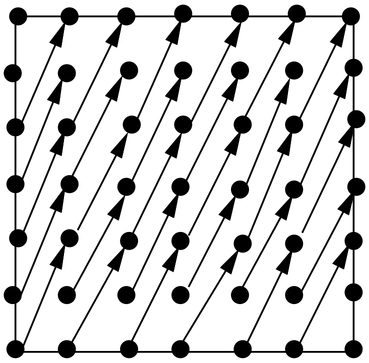

Figure 10: Communication patterns

arrays of different dimensions, the occurrence matrix R and vector $\mathbf{c}$ are padded by $M - N_A$ rows of zero, where M is the maximum of dimensions of all arrays under consideration, and m.

As with the iteration space transformations, we represent the alignment functions by transformation matrices. We characterize the alignment of array B to array A by $T = (\alpha, \mathbf{a})$, where α is an M by m matrix, and $\mathbf{a}$ is an M-vector. The element (i, j) of B is mapped to $\alpha(i, j)^T + \mathbf{a}$. Informally, α provides a linear transformation similar to a skew or permutation of the data space $\mathcal{B}$ and $\mathbf{a}$ provides the shift.[14]

Example 6 Consider the loop in the previous example. Assume that arrays A and B are of the same size and B(i,j) is aligned to A(i,j), and each pair of elements (i,j) belongs to a processor. Figure 10 depicts the resulting communication patterns. There are $O(n^2)$ communications. The alignment transformation on B results in communication free computation (Figure 11). The transformed loop corresponds to:

```
for i = 1, n
   for j = 1, n
      A(i,j) = B(i,j)
   end for
end for
```

□

Consider arrays A and B occurring in the lhs and rhs of a statement.[15] Let the array

[14] We confine ourselves to integer, invertible transformations as alignment functions.

[15] The idea is to align the rhs array B to lhs array A. However there is nothing sacred about doing it this way. The alignment of rhs arrays to lhs works to advantage when there are multiple arrays referenced which can all be aligned to single lhs array.

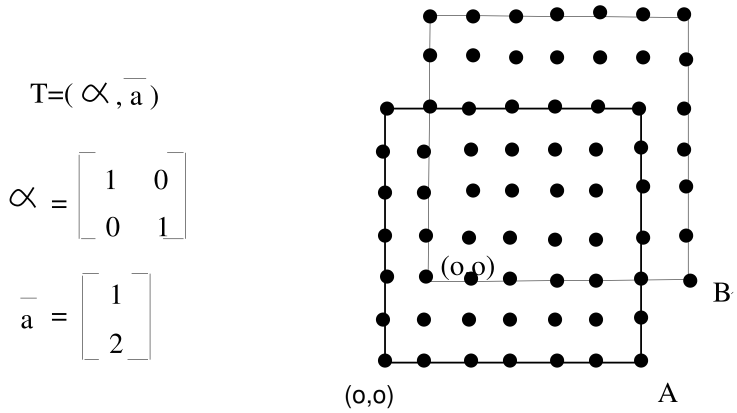

Figure 11: An alignment of B to A

reference for arrays A and B be

$$F_A(\mathbf{J}) = C\mathbf{J} + \mathbf{c}$$

$$F_B(\mathbf{J}) = U\mathbf{J} + \mathbf{u}$$

The hyperplane S is the iteration space V along which the references are the same is

$$S = \{\mathbf{J} \in V | F_A(\mathbf{J}) = F_B(\mathbf{J})\}$$

Thus, S is the hyperplane in which references to A and B are aligned. The objective of an alignment is to find a transformation $T = (\alpha, \mathbf{a})$ such that

$$T : U\mathbf{J} + \mathbf{u} \rightarrow \alpha U\mathbf{J} + \mathbf{a} + \mathbf{u}$$

$$\text{and } |S'| \text{ where } S' = \{\mathbf{J} \in V | F_A(\mathbf{J}) = TF_B(\mathbf{J})\} \text{ is maximum.}$$

A heuristic to increase the alignment is to maximize the dimension of S'[OH92]. Assuming the availability of a cost model, one has to enumerate all the possible alignments and choose the best. Moreover, arriving at a cost model (parameterized by the architectural features) at the right level of detail itself is a hard problem.[16]

Algorithm 1 *[OH92] Construct F_A and F_B, and thus U, u, U^T, C, c, C^T. Reduce the following to row echelon form.*

$$\left[U^T | C^T | I\right]$$

Place column i of C^T as the row i of α if it is independent of the others, otherwise choose an independent column of I in the row echelon form.

Example 7 *[OH92]* Consider the alignment of array B to array A in the program segment below.

[16] We will see that there exist heuristics for a restricted class of alignments, called permutations.

```
for i = 1, n + 3
   for j = 1, 2n
      B(i, j) = i + j
   end for
end for

for i = 1, n
   for j = 1, n
      for k = 1, n
         A(i, j) = B(j + 3, i + j) * C(k, n - i + 1)
      end for
   end for
end for
```

The number of non-local references to B is $O(n^2)$, and that for C is $O(n^3)$, assuming an identical element by element distribution of the arrays.

$$F_A(\ \mathbf{J}\ ) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} i \\ j \\ k \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}$$

$$F_B(\ \mathbf{J}\ ) = \begin{bmatrix} 0 & 1 & 0 \\ 1 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} i \\ j \\ k \end{bmatrix} + \begin{bmatrix} 3 \\ 0 \\ 0 \end{bmatrix}$$

Now, applying row operations on

$$\left[\begin{array}{ccc|ccc|ccc} 0 & 1 & 0 & 1 & 0 & 0 & 1 & 0 & 0 \\ 1 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{array}\right]$$

we get

$$\left[\begin{array}{ccc|ccc|ccc} 1 & 0 & 0 & -1 & 1 & 0 & -1 & 1 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{array}\right]$$

Thus, alignment transformation on B is

$$T = \begin{bmatrix} -1 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix}, \begin{bmatrix} -3 \\ 0 \\ 0 \end{bmatrix}$$

The code becomes[17]

```
for i = 1, n + 3
   for j = 1, 2n
      B(j - i + 3, i - 3) = i + j
```

[17]The alignment of array C to A produces another similar transformation of the code.

```
      end for
   end for

   for  i = 1, n
      for  j = 1, n
         for  k = 1, n
            A(i,j) = B(i,j) * C(k, n - i + 1)
         end for
      end for
   end for
```

All the references to B now are local.[18]

□

5.1.3 Restricted Alignments

Given a program we have to choose the best alignment between pairs of arrays among the constraints occurring in different parts of the program, and then compute the alignment function for that pair. The transformation $T : (\alpha,\ \mathbf{a}\ )$ could be either a general linear transformation, a permutation, an embedding, a shift or reflection. In this section we present heuristics for determining the permutation alignment of arrays in a program (due to [LC91]).

Definition 12 (Permutation Alignment) *The alignment function $T : (\alpha,\ \mathbf{a}\ )$ from an n-dimensional data space to another is said to be a permutation if α is a column permutation of $\mathcal{I}$ the identity matrix, and* $\mathbf{a}$ *is zero.*

For instance, a mapping of the element $(i_1, \ldots, i_n)$ to $(i_{q_1}, ..., i_{q_n})$ where $(q_1, ..., q_n)$ is a permutation of $(1, ..., n)$ is a permutation alignment.

An embedding alignment occurs if the dimension of the data space of the array to be aligned is less than that of the target data space.

The computation for permutation alignment reduces to representing the constraints on the alignment and finding the groups of dimensions of different arrays that go together. Li and Chen [LC91] formulate the problem as one of partitioning the node set of *component affinity graph* defined below. The alignment constraints are restricted to be *affinity* relations between dimensions of arrays.

Definition 13 (Affinity) *Given a pair of references* $\mathsf{A}(i_1, \ldots, i_m)$*,* $\mathsf{B}(j_1, \ldots, j_n)$ *occurring in lhs and rhs of a statement in a loop, dimension p of* A *has affinity to dimension of q of* B *iff $j_q = i_p + c$ where c is a constant.*

Definition 14 (Component Affinity Graph) *The component affinity graph (CAG) of a program is a graph CAG(V,E), where V is the set of dimensions of all the arrays, and $(v_i, v_j) \in E$ iff v_i and v_j have affinity and v_i and v_j do not belong to the same data space.*

[18] Note that the first loop needs a different loop scheduling to have all lhs references to B local.

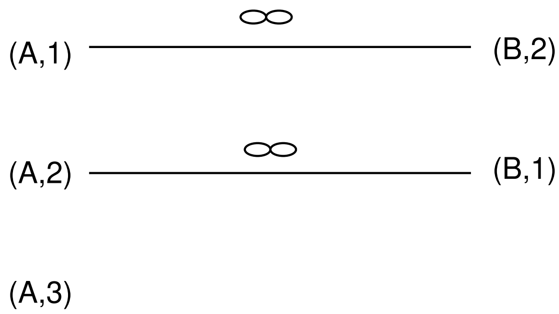

Figure 12: Component Alignment Graph

The edges of the CAG are given weights corresponding to the penalty for not aligning. Li and Chen [LC91] assign ∞ for dimensions appearing in `for` loops, ϵ for multiple affinities to the same dimension, or 1 otherwise.[19] Gupta and Banerjee [GB92] use the cost of communication to arrive at a better cost measure of non-alignment. However, their method to estimate the cost is *ad hoc*: The cost of not aligning dimensions p and q of two different arrays is considered to be the cost of choosing some other arbitrary alignment. The difficulty lies in the large number of permutations without the p having aligned to q.

A permutation alignment is a partitioning of the node set of CAG such that the total weight of edges between nodes in different partitions is minimum. The problem is clearly intractable.[20] Li and Chen [LC91] provide a heuristic algorithm.

Example 8 Consider program segment

```
for i = 1, n
  for j = 1, n
    for k = 1, n
      A(i, j, k) = B(j − 1, i − 2)
    endfor
  endfor
endfor
```

In a CAG (A,p) denotes p^{th} dimension of array A. The partitions would be $\{(\mathsf{A},1),(\mathsf{B},2)\}$ and $\{(\mathsf{A},2)(\mathsf{B},1)\}$. If we do an (1,2) shift on B and embed B(i,j) onto B(j+2,i+1,1), we get,

[19] They consider the alignment for data parallel programs. The weights for the sequential programs we have been considering happen to be ∞ always and hence does not provide a good cost measure.

[20] It is easy to see that MAXCUT [GJ79] can be reduced to the problem of partitioning the node set of CAG. The problem is tractable when the CAG is planar [GJ79]. It is interesting to find the characteristics of the programs that result in planar CAGs, and to investigate the fraction such programs form of the benchmarks.

```
for i = 1, n
   for j = 1, n
      for k = 1, n
         A(i, j, k) = B(i, j, 1)
      end for
   end for
end for
```

With this alignment, the number of non-local references to B reduce from $O(n^2)$ to zero.

□

5.1.4 Comments

Alignment improves the lower bound of the locality exploited and thus it is only a first step in solving the the data partitioning problem. A good partitioning can certainly improve the locality further. We will consider automatic data partitioning in the next section. A loop organization equivalent to alignment is access normalization. However, as we have seen in the previous section, access normalization is useful only to a limited extent. Since data alignment can be applied to many pairs of arrays it is a better technique (leaving the memory management difficulties out). The aligned arrays have to be partitioned in similar ways along the aligned dimensions. Across different segments of a program we could possibly have different classes of aligned arrays that are partitioned differently. Occurrence of an array in different loops gives raise to multiple ways of alignment and multiple classes of alignment and partitioning. The choice again relies on the cost model. When there exist complex alignment functions, memory management has to either rely on complex addressing mechanisms to implement non-rectangular arrays or resort to a subsuming rectangular array at the cost of wasted space and consequent holes in caches. Since the holes will never be accessed its transparent at the top level. The trade-off requires some cost benefit analysis.

5.2 Data Distribution

Proper distribution of data is essential to achieve good performance when non-local data access costs are significantly higher than local accesses. In this section we address the issue of how to distribute arrays to multidimensional array of processors $\mathcal{P}$, called a processor space. The assumption of a rectangular processor space simplifies the distribution functions, and other topologies can be mapped on to a rectangular array. As discussed in the previous section, aligning a set of arrays improves the lower bound on the locality. Once the arrays have been aligned, it is important to distribute them together (so as not to undo alignment).

5.2.1 Basic Idea

Definition 15 (Data Distribution) *The distribution of a data space $\mathcal{D}$ to a processor space $\mathcal{P}$ is a function $f_d : \mathcal{D} \to \mathcal{P}$.*

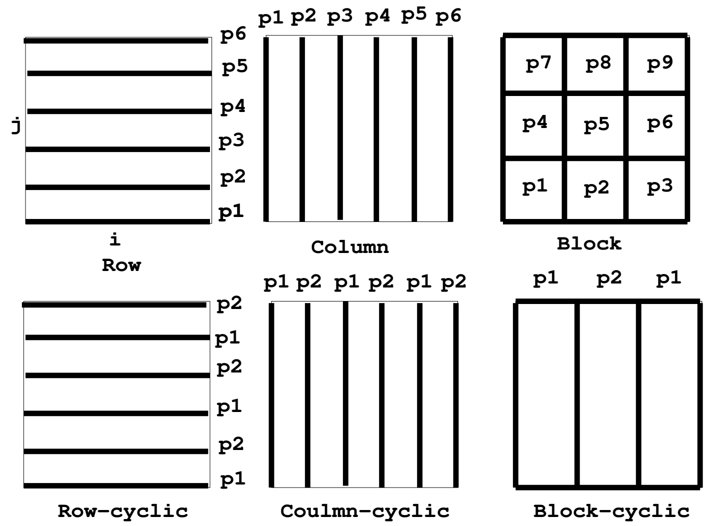

Figure 13: Regular Distributions onto a linear array of processors

It is interesting to see that when the dimension and size of the processor space $\mathcal{P}$ is greater than or equal to that of the array to be distributed, the distribution function can be linear. Unfortunately, this is typically not the case and the distribution function is therefore non-linear involving *floor* and *mod* operators on functions of the indices. *Block*, *column*, *cyclic*, and *block-cyclic* are some of the common distributions that have a regular structure. Block (column, row) distribution maps blocks (columns, rows) of array elements onto processors. The assignment of individual chunks of data to the available processors could be cyclic (Figure 13). Cyclic distributions can aid in balancing the load.

We could think of most of the regular distributions as a set of parameters: the block sizes (in each dimension), their orientation, the way they are assigned to processors etc.[21] A parameterization of this sort helps in finding a distribution by exhaustive search, as we see below.

The general problem is to find the optimal data distribution for all arrays in a program. For a program that has flow constraints, the problem of finding an optimal distribution is NP-hard even when the number of allowed distributions is constant, and a fixed number of operations are to be performed [Mac87]. The problem is tractable when there are no flow or output dependences involved, but then such programs are uninteresting and uncommon.

The intractability of the problem has motivated research in finding good heuristic data distributions, and design tools to assist in the process. However, some believe that the distribution is best done by the user. For this reason, extensions have been added to some languages like Fortran to enable one to express the distribution for an array [HKK+91, KKBP90, Ger89].

[21] In particular, in a two dimensional case, we could think of distribution, as two lines, and difference in consecutive intercepts, just as we specified tiling.

It is a matter of ongoing debate as to whether data distribution is best done by the user, or by an automated tool. The factors that affect the best choice of a data distribution are numerous, and interact in non-intuitive ways. Thus, a user defined distribution does not necessarily result in better performance. Moreover, it is better to hide the architectural details for reasons of portability and programming ease. In simple cases, a user can successfully arrive at simple distributions, but so can an automated tool, since the heuristics for these distributions tend to be simple.[22] Very complex distributions are hard to arrive at, and are often the result of extensive theoretical study. The distributions that fall in between these extremes are the ones that are hard for both the user and the automated tool. There is ample scope for heuristics. While user advice would certainly help, we as yet do not know what insights the programmer might be able to provide, and how he can express his intuitions. We believe that an automatic or a semi-automatic tool for data distribution has promise.

5.2.2 Computation of Distribution

A key to automatic data distribution is a cost model that lets one compare two competing distributions. This cost model can be quite detailed, using estimates for the cost of accesses used in the program [BFKK, Who92] or can be quite abstract. It is non-trivial to decide on the appropriate level of detail desired of the cost model. In the following paragraphs we discuss some of the approaches to derivation of array distributions.

Exhaustive Search. Automatic data partitioning can be considered to be a search in a space of array distributions. (That is, each state is a candidate data distribution.) A transition from a state to another is a decision to change the distribution in a certain way, say decreasing the size of a data partition in a dimension. A cost measure is associated with each state, that estimates the execution time of the program if that distribution is used. The reference patterns can provide heuristics to prune the search. Statements in the program also provide the desired relation between distributions for different arrays. The search space is usually very large. It is hard to see how constraints between distributions for arrays can be accommodated in the transitions. [Who92] is an example of this approach.

Constraint Resolution. The observation that an array distribution can be specified in terms of several parameters, and that reference patterns in the program provide *constraints* on these parameters led Gupta and Banerjee [GB92] to a constraint-based approach.[23] The first step in this approach is to find a permutation alignment of arrays and thus identify the sets of dimensions of the arrays that are similarly distributed. The distribution is characterized as one of finding whether a dimension of an array (in an aligned group of dimensions) is distributed or not; whether to distribute in blocked fashion, or cyclically; the block size;

[22] For example, a non-rectangular loop suggests a cyclic distribution; uninternalized dependences suggest block distribution.

[23] It should be noted that the term *constraint* is a misnomer. So called constraints are reference patterns in a statement which merely *suggests* the distribution that leads to faster execution. The idea is to choose a distribution that meets as many of the constraints as possible.

and number of processors on which to distribute. These decisions constitute distribution parameters. The idea is to associate a *quality measure* with each constraint advocating a particular parameter; sum the quality measures for each parameter in the program; and then choose the best.

For example, consider the following program segment:

```
for i = 0, n
   for j = 0, n
      A(i, j) = A(i − 1, j) + B(i, 2 ∗ j)
   end for
end for
```

The dependence suggests that block distribution is better than cyclic distribution (irrespective of whether it is block-cyclic, or just cyclic). The reference to array B suggests that the block size in the second dimension for B be twice that of A. On the other hand, following loop suggests a cyclic distribution due to non-rectangular iteration space.

```
for i = 0, n
   for j = i, n
      A(i, j) = ...
   end for
end for
```

The quality measure of a constraint for a suggested parameter is the difference in execution time when the suggested parameter is chosen and when it is not. For example, the quality measure of a constraint that suggests block distribution is the difference in execution time between the case where block distribution is chosen and the case when it is not. When the distribution is not block then it could be cyclic with a block size of 1 or any other size. The idea is to sum such constraints over the program and choose the best parameter.

Unfortunately, the parameters are not independent. For example, to compute the quality measure of a constraining advocating cyclic distribution we need to know the rest of the parameters. Resorting to some default parameters weakens the detail of the architecture model. The default parameters often tend to be ad hoc. Gupta and Banerjee [GB92] have an implementation that shows the feasibility of the approach, albeit with limitations. The constraints could have a scoping rule that they are effective only in certain parts of the program. The detection and the type of the constraint is limited, and has room for improvement.

Analytical Method. As we have seen in earlier sections, there exist good algebraic solutions to loop partitioning. There are two analytical approaches for data partitioning. We could construct an algebraic representation of the data partitioning problem, possibly in terms of the reference patterns and input dependences. Techniques similar to those of loop partitioning can be applied to data partitioning. This approach has the limitation that it can at best yield a good solution at the loop level, and as yet we do not know how to solve it for a collection of loops. A second approach is to perform process directed data partitioning. A loop partitioning implicitly provides information on the distribution of some of the arrays. Again, the approach would be limited to loop level.

Ramanujam and Sadayappan [RS91] formulate the task of data partitioning for arrays appearing in *dependence free loops* as a linear programming problem. Consider the following dependence free loop.

```
for  i = 0, n
  for  j = 0, n
    A(i, j) = B(i − 1, j) + B(i, j − 1)
  end for
end for
```

Given a family of hyperplanes h_A (lines here) in data space $\mathcal{A}$, find a corresponding family of hyperplanes h_B in data space $\mathcal{B}$ such that an element of A accessed in an iteration lie on a particular hyperplane, say h_{A1}, then the elements of B accessed in the same iteration lie in a corresponding hyperplane h_{B1}. In the above example, a family of lines $i + j$ satisfy. (The constants for corresponding particular lines for the arrays will have a difference of 1).

The above approach is very limited because communication free loops are rare. They do not provide a systematic way of arriving at the loop structure that corresponds to the data partitioning derived. Techniques used in access normalization[LP92] that we discussed previously can be applied to do this.

A loop partitioning implicitly provides the data partitioning for the lhs array. Consider for the following loop.

```
for  i = 0, n
  for  j = 0, n
    A(i, j) = A(i − 1, j − 1) + B(i, j − 1)
  end for
end for
```

A loop partitioning in the $(1, 1)$ direction provides a partitioning of array A in the same direction, i.e, as defined by the family of hyperplanes $j - i$. A naive partitioning of B results in communication, where as an alignment to A and partitioning along $j - i$ provides a communication free loop. Although this approach is also limited to loops, it provides the new loop structure from the loop partitioning techniques that we discussed before.

6 Conclusions

In the preceding sections we described state of the art techniques for loop and data transformations. Instead of presenting a collection of apparently unrelated loop transformations, we primarily discussed linear transformations that subsume any compound linear transformation. Tiling is another important loop transformation that we briefly discussed. We presented in detail the techniques for deriving the transformed loop nests and provided illustrations for these techniques. Tiling is another important loop transformation that we briefly discussed. We then presented data alignment and distribution techniques and how they affect loop nests (like improvement in locality). From our discussion, we draw a number of conclusions:

- The architectural features of a target machine influence the choice of loop and data transformations.
- It is possible to achieve certain objectives with loop and data transformations, but there may not exist polynomial time solutions to do it.
- Currently, the choice of a transformation is based primarily on qualitative reasoning, and at best a limited amount of quantitative reasoning.
- There exist distinct transformations that increase outer (inner) loop parallelism, improve static and dynamic locality, improve load balance, but no one single transformation may be able to achieve all of the desired goals to the extent needed. In particular, there can be conflicting goals on a transformation. For example, a transformation to increase parallelism can also increase load imbalance.
- Loop transformation techniques consider only one loop at a time and hence the data partitionings if imposed need not be the best for adjacent nested loops.
- Loop and data transformation techniques have been developed more or less independently, and there is no uniform methodology of combining loop and data transformations for a given program.

7 Directions for Future Work

Hierarchically structured parallel machines are attractive due to their scalability and better cost-performance ratio. We believe that future general purpose parallel computing will be dominated by such architectures. Such machines are scaled in size by increasing the number of levels. Access locality on such machines is structured and is more general than the two level distinction between local and remote accesses. Program restructuring tools take the role of optimizers of sequential (and programs with some parallel constructs) to improve the run time on such target architectures. We intend to work on a *unified* framework to restructure programs as whole to run efficiently on such machines. As we see it now, there are four related sub-problems that must be solved:

- A model of the hierarchical machines, nested loops and data mapping to predict the run time of a transformed loop (and thus a complete program) is necessary for a quantitative evaluation of different choices in restructuring.
- New loop and data transformations are required to consider a wider range of loops which routinely arise and are not handled by the current techniques. For instance, imperfect loop nest are very common and have to be brought under the purview. Number of dependences in the loop set limits on what can be achieved and a transformation to reduce the number of dependences across iterations would greatly help.
- A single loop transformation to give the best parallelism, locality, and load balance in a loop.
- A representation of constraints occurring in several loops, global transformations, and global criteria to evaluate the different choices of such transformations. Global transformations are the backbone of a successful unified methodology for compiling for parallel machines.

7.1 Performance model

As noted earlier, the choice of a particular loop transformation from among many is often based only to a limited extent on quantitative measures. To be able to effectively choose an appropriate transformation, it is necessary to be able to predict the run time of a transformed loop on the target architecture. Since we usually have complete information about the dependence structure, loop bounds, and possible data mappings for a nested loop, it should be possible to estimate the run times quite accurately.

IN order to predict the run time of a nested loop, it is necessary to have a good model of target architectures and the nested loops themselves. In particular, the existing architectural models for non-uniform memory access machines [SZ91, HS92, KKB92], (with possible modifications,) might be able to serve this purpose. Dependences and data mapping, together with the mapping of iterations on the processor space can give us an estimate of the number of remote accesses. An analysis of the *self input/output dependences* can give us a measure of variable reuse. Loop bounds can provide us information on the amount of parallelism, and together with the mapping of iterations onto the processors space, they can provide an estimate of load imbalance.

7.2 New loop transformations

Dependences constrain the possible valid loop transformations. The number of dependences limit the improvements that a transformation can achieve. For example, one can internalize only (n-1) linearly independent dependences; each dependence constrains the orientation of a tile; a column distribution of an array does not suffice when there exist dependences (0,1) as well as (1,0).

Computation Alignment is a new loop transformation that we are developing to reduce the dependence constraints, and to increase the *uniformity* across the loop nests. In the existing

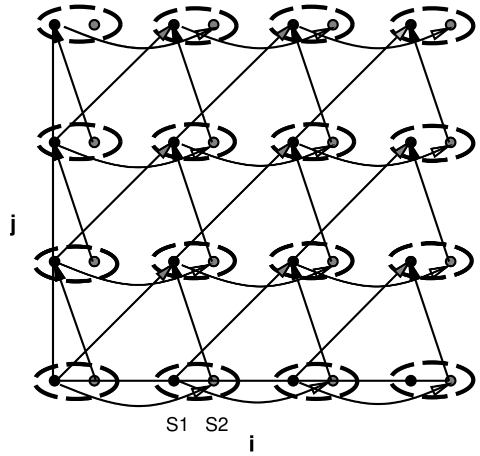

Figure 14: Before Computation Alignment

transformations, an iteration is treated as the basic computation unit to be rearranged. In computation alignment on the other hand, we treat individual instances of statements[24] as basic units of computation, and a loop transformation is a rearrangement of the statements. With this approach, we add an extra dimension to the iteration space to represent the statements in the body of the loop. [25] Loop carried dependences are explicitly represented across statements in corresponding iterations. The following example illustrates the representation.

Example 9 Consider the following doubly nested loop which has $(1,1)$ and $(1,0)$ dependence in accessing array A, and $(0,1)$ dependence in accessing array B. An outer loop parallelization leaves two dependences resulting in remote accesses.

```
for  i = 1, n
   for  j = 1, n
      S1 :  A(i,j) = A(i − 1, j − 1) + B(i, j − 1)
      S2 :  B(i,j) = A(i − 1, j)
   end for
end for
```

Computation alignment can reorganize the loop to the one below, with only two loop carried dependences. The conditional statements are needed due to finite boundaries of the iteration space; they can be removed by a further rearrangement step.

```
for  i = 1, n
   for  j = 0, n
      S1 :  if(j ≥ 1)  B(i,j) = A(i − 1, j)
      S2 :  if(j ≤ n − 1)  A(i, j + 1) = A(i − 1, j) + B(i, j)
```

[24] We consider only assignment statements right now.

[25] Clearly, a loop with a single statement does not have room for computation alignment.

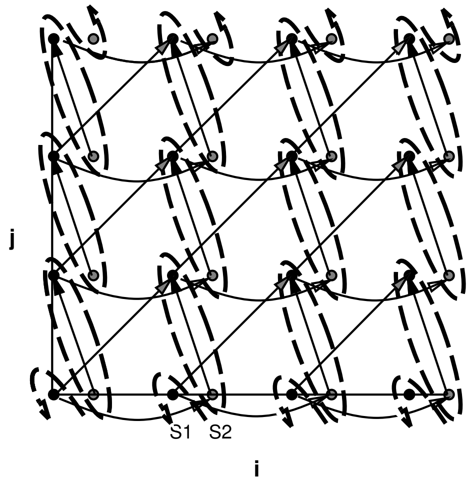

Figure 15: After Computation Alignment

```
    end for
end for
```

□

Computation alignment can reduce the number of dependences and thus increase the for transformations. It can increase *uniformity* across different loops and thus achieve inter loop alignments. It can also improve the dynamic locality among other things.

We defer the representation, techniques, legality criterion, different types of computation alignments, and details of applications of computation alignment to the following paper[KSb].

7.3 Cure-all

When a sequence of transformations is applied, then one of the transformations can achieve a desired objective such as improved parallelism, but it can undo the effects of an earlier transformation (which was applied to achieve a different objective). It is an open problem how to derive a single loop transformation that results in a transformed loop with best estimated execution time. Such a transformation has to (1) work with the target architecture model and (2) arbitrate between different parameters. The transformation has to exploit the *structured locality* a hierarchical architecture provides.

Internalization makes as many dependences as possible independent of outer loops. As discussed in earlier sections, this improves locality and parallelism. Dependences that are internalized determine the characteristics of the transformed loop with respect to parallelism, locality and load balance. One way of deriving a near optimal transformation is to assign weights to the flow, input and output dependences, and to arbitrate among the different subsets of these to be internalized – the higher the weight, the higher the desire to internalize.

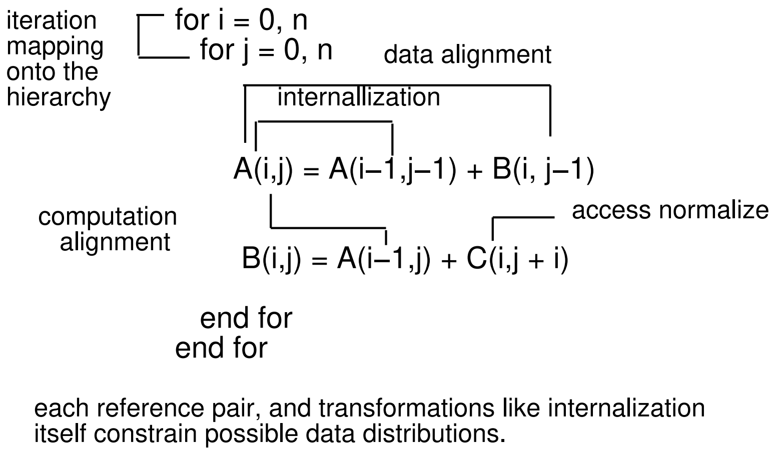

Figure 16: What Different transformations do

These weights may have to be arrived at for a target architecture by collecting traces for some *pilot loops*. For instance, a dependence that has a reuse [26] might be given more weight than one without reuse. Among those dependences that have reuse a higher reuse implies higher weight. The input and output dependences with reuse are given lower weight than the flow dependences with reuse, because an internalized flow dependence improves parallelism. Dependences that lead to row, column or block distributions are given higher weight than those that lead to say, diagonal distributions, because the former distributions are more natural, have simpler index computations, and may lead to better load balance. The dependences due to an array of real numbers are given higher weight than those due to an array of integers. A linear dependence may have to be give higher weight so as to reduce non-regular accesses on the network. The weights may have to be revised based on the transformations already applied to adjacent loops.

7.4 Unified methodology

Figure 16 shows different transformations that can be applied to a loop. The lines roughly indicate the kind of patterns of references that a particular transformation is interested in. For example, array B is aligned to array A by an analysis of the references $\mathsf{A}(\mathsf{i},\mathsf{j})$ and $\mathsf{B}(\mathsf{i},\mathsf{j}-1)$.

A loop transformation has implications on the best data distribution for arrays accessed in the loop. For example, an internalization of a $(1,1)$ dependence due to array A being referenced in a two dimensional loop suggests that it is best to distribute A (with at least one reference to A(i,j)) along families of lines parallel to $(1,1)$. A data transformation also influences the structure of a loop. For instance, access normalization attempts to transform a loop so that the existing distribution of an array is suitable. A data alignment does not

[26] A dependence between two references to an array in a loop with dimension higher than that of the array has a hyperplane in the iteration space along which the dependence relation exists due to the same element of the array. In that case, we say there is a reuse along this hyperplane.

affect the dependence structure of a loop, but constrains the data distribution to be the same for the aligned arrays. The data access directions in an array can also be changed by a data alignment, for example, by transposing the array, so that reference $\mathsf{A(i-1,j)}$ becomes $\mathsf{A(j,i-1)}$, and if the loop has an iteration vector (i,j), the reference potentially uses all data in the cache.

Restructurings based on single loops can have adverse effects on the adjacent loops. That is, optimal transformations of a given loop with the data it accesses, can preclude optimal loop transformations for other loops in the program. As yet, there is no integrated approach to derive optimal loop and data transformations for a given set of nested loops and data. Success of automatic restructuring technology will depend heavily on global program transformations, parameterized by the target architecture. We plan to study an *unified* framework to represent constraints in and across loops, and to systematically perform the loop and data transformations that result in optimal run time on a target hierarchical architecture.

A Catalog of Loop Transformations[27]
Version 0.0

Dattatraya Kulkarni and Michael Stumm
Computer Science Research Institute
University of Toronto, Toronto M5S 1A4
kulki@{eecg,csri}.toronto.edu

This is a catalog of common program transformations. For a complete description of the transformations, their validity, the computation of new loop bounds, and computation of the new array references refer to relevant papers or the accompanying document.

We classify the transformations into *preliminary*, *primary*, and *secondary* transformations. Preliminary transformations are the first to apply, and the purpose is to improve the data and control dependence analysis. Primary transformations achieve the intended restructuring of the program. Some of the objectives could be to enhance parallelism, load balance and/or locality. Secondary transformations are improve the performance of the programs further by fine tuning. For example, one could identify block communication primitives.

The loop transformations can change the dependence structure. It is necessary to ensure that a transformed loop preserves the sequential semantics, that is, produces the same results as the original. The sequential execution of the transformed loop is possible if all the new flow dependences are positive. A transformation is said to be legal if all the dependences in the transformed loop are positive.

A Preliminary Transformations

These transformations are intended to improve further analysis.[28]

Induction variable elimination

Simplifies the subscript analysis in dependence tests.

```
j = n
for i = 0, n                          for i =0, n
  A(i) = B(j-1)                        A(i) = B(n-i-1)
  B(j) = C(i)      ===>                B(n-i) = C(i)
  j = j-1                             end for
end for
```

But it can be difficult; for instance, in the following example, We have to detect that C(1) and C(2) are induction variables, and that their initial relation and update relation to find the dependence.

[27] The objective is to make this catalog a one point source eventually. Considering the volume of research done in the area incompleteness is natural, and we welcome suggestions for transformations to be included.

[28] Although most of the preliminary transformations were introduced in the context of vectorization, we leave out vectorization techniques here.

```
C(1) = 1
C(2) = 0
for i = 0, n
  A(C(1)) = A(C(2))
  C(1) = C(1) +1
  C(2) = C(2) +1
end for
```

Normalization

Many transformations assume that the lower bound of a loop index starts with zero and stride in one. [29]

```
for i = 2, n, 2                     for i = 0, n/2-1
  for j = 0, n                        for j = 0, n
    A(i,j) = A(i-2,j)   ====>           A(2i+2,j) = A(2i,j)
  end for                             end for
end for                             end for
```

Forward substitution

Simplifies dependence analysis by substituting constants for the expressions in array references.

```
x = n +1
for i = 0, n                         for i =0, n
  A(i) = B(i)                          A(i) = B(i)
  C(i) = A(x)          ===>            C(i) = A(n+1)
end for                              end for
```

False dependence elimination

Anti and output dependences are false as they arise just because of sharing the same storage location. Idea is to eliminate these and work only with the flow (true) dependence.

```
A = B + C                      A1 = B + C
A = D + E            ===>      A2 = D + E
... = A                        ... = A2
```

Loop distribution

To enable vectorization, or provide simpler loops for analysis distribute statements in a loop into multiple similar loops.

[29] As we see later, changing the stride itself is treated as another transformation called scaling.

```
for i = 0, n                              for i = 0, n
    A(i) = x                                A(i) = x
    C(i) = A(i-1)+D(i-1)    ===>          end for
    D(i) = C(i)                           for i = 0, n
end for                                     C(i) = A(i-1)+D(i-1)
                                            D(i) = C(i)
                                          end for
```

Node splitting

Dependence cycles in a loop, prevent loop distribution and vectorization. It is possible break the cycles by introduction of temporary variables to keep copies of data. Node splitting is essentially an introduction of temporary variable followed by loop distribution.[30]

```
for i=1,n                               for i = 1,n
  A(i) = B(i) + C(i)                     A(i) = B(i) + C(i)
  D(i) = A(i-1) + A(i+1)  ===>           Temp(i) = A(i+1)
end for                                  D(i) = A(i-1) + Temp(i)
                                        end for
```

The original loop contains a flow dependence (1) and an antidependence (-1) leading to a cycle, and preventing the loop distribution. Introduction of temporary variable Temp, to keep a copy of A enables us to distribute the loop (and subsequently vectorize) as below.

```
for i = 0,n
  A(i) = B(i) + C(i)
  Temp(i) = A(i+1)
end for
for i = 0,n
  D(i) = A(i-1) + Temp(i)
end for
```

Loop fusion

The opposite of distribution (or fission). Fusing adjacent loops increases the grain size and decrease the overhead of a doall loop. This can also improve cache performance by increasing reuse of elements accessed in fused loops.

B Primary Transformations

One may have to apply a sequence of these transformations.

[30] Some consider only the introduction of temporaries as node splitting.

Strip mining

Strip mining transforms a 1-d loop into a 2-d loop.[31] A dependence (c) becomes $(0, c)$ and $(1, c - S - 1)$, where S is the strip size (and assuming $S \geq c$). It is always legal to do strip mining. (That is, strip mining does not result in negative dependences in the transformed loop. The bounds are rectangular and easy to compute. Strip mining is done mostly to exploit the vector register size.

```
for i = 0, n                               for ii = 0, n, 64
  A(i) = A(i-1)       ===>                   for i = ii, min(ii+63, n)
end for                                        A(i) = A(i-1)
                                             end for
                                           end for
```

Loop interchange

Interchanges nested loops.[32] A dependence (a, b) becomes (b, a), and thus interchange is not valid if any dependence (a, b) has a negative b. Loop interchange can enhance inner loop parallelization, vectorization, outer loop parallelization, adjustment of array access stride, and matching loop parallelism to data distributions. Loop bound computation can be non-trivial if the iteration space is non-rectangular.

```
for i = 0, n                            for j = 0, n
  for j = 0, n                            for i = 0, n
    A(i,j) = A(i,j-1)   ====>               A(i,j) = A(i,j-1)
  end for                                 end for
end for                                 end for

for i = m1, n1                          for j = m2, n1
  for j = m2, i                            for i = max(m1,j), n1
    A(i,j) = A(i,j-1)   ====>               A(i,j) = A(i,j-1)
  end for                                 end for
end for                                 end for
```

Loop permutation

Generalization of a loop interchange. Gives a loop that has indices that are a permutation of the original loop indices. A dependence (a, b, c) with permutation $\begin{pmatrix} 1 & 2 & 3 \\ 3 & 1 & 2 \end{pmatrix}$ becomes (c, a, b). A permutation is valid if all the permuted dependences are positive. The above permutation is valid if the dependences are $\{(1, 1, 0), (1, -1, 1)\}$, not if a dependence $(1, 0, -1)$ also exists. Loop bounds are again non-trivial to compute for non-rectangular iteration spaces.

[31] We see later in the document that strip mining is a trivial case of tiling.

[32] In a doubly nested loop this corresponds to transposing the iteration space.

A permutation can be realized as a sequence of interchanges, but finding legal sequences of interchanges is a difficult task.

```
for i = 0, n                                    for k = 0, n
  for j = 0, n                                    for i = 0, n
    for k = 0, n                                    for j = 0, n
      A(i,j,k) = A(i-1,j+1, k-1)  ===>                A(i,j,k) = A(i-1,j+1,k-1)
    end for                                         end for
  end for                                         end for
end for                                         end for
```

Loop Reversal

Reversing a loop may be ncessary to enable loop interchanges. By reversal of a loop with lower bound L and upper bound U, it is made to iterate from $-U$ to $-L$ with the same stride. A dependence (a, b) becomes $(a, -b)$. Note that, whenever there are dependences carried by the outermost loop, it is illegal to reverse it. For example, a dependence $(1, 2)$ becomes $(-1, 2)$ after interchanging the outer loop, which is illegal as the new dependence is negative.

```
for i = 0, n                                for i = 0, n
  for j = 0, n                                for j = -n, 0
    A(i,j) = A(i-1,j+1)    ====>                  A(i,-j) = A(i-1,-j+1)
  end for                                     end for
end for                                     end for
```

Loop skewing

A dependence $(d1, d2)$ becomes $(d1, fd1 + d2)$ where f is the skew factor. A valid skew can always be found for a given loop.

```
With a skew factor of +1.
for i = 0, n                                for i = 0, n
  for j = 0, n                                for j = i, i+n
    A(i,j) = A(i-1,j)    ====>                  A(i,j-i) = A(i-1,j-i)
  end for                                     end for
end for                                     end for
```

Wavefront

The idea is to find a family of hyperplanes in the iteration space along which the iterations can be executed in parallel. Then the loop is transformed in such a way that the hyperplanes are executed sequentially, and all iterations on a hyperplane are executed parallelly. (This to inner loop parallelization.) A wavefront transformation results in dependences carried by the outer most loop. A valid wavefront can always be found.

Loop skewing is just a simple instance of wavefront transformation.

Unimodular

A unified transformation that subsumes permutation, reversal, and skewing. It is characterized by a unimodular matrix; the new loop bounds and new dependences are computed from this matrix. A 2d loop interchange has the transformation matrix $\begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}$. Techniques are not discussed here. See loop interchange for an example.

Internalization

A unimodular transformation specialized to exploit outer loop parallelism and locality for general loops. Techniques not discussed here. The basic idea is to transform a loop in such way that maximum number of dependences are not carried by the outer loop, and the size of the outer loop is maximum. The technique applies to increasing the dynamic locality as well.

Access Normalization

It is similar to unimodular loop transformations in that a a matrix transformation is used. However, this matrix is derived from array access patterns, and need not be uimodular. The matrix is made invertible if needed. The objective is to simplify the array subscript expressions so as to match the simple data distributions and hence does not consider any other loop optimizations.

```
for i= p, N1-1, P                        for u= p, b-1, P
  for j= i, i+b-1                          for v= u, u+N1+N2-2
    for k= 0, N2-1                           read A(*,v)
      B(i,j-i)= B(i,j-i)+ A(i,j+k)           for w= 0, N1-1
    end for                                    B(w,u)= B(w,u)+ A(w,v)
  end for                                    end for
end for                                    end for
                                         end for
```

Note that addition of the line `read A(*,v)` is a *secondary* transformation.

Scaling

It is the converse of normalization in that it changes the stride from unit to non-unit. Characterized by an invertible integer matrix (generally not unimodular). Since most of the transformations operate on normalized loops, scaling is not generally employed. Scaling can provide more *natural looking* subscripts and bounds.

```
for i = 0, n/2                         for i = 0, n, 2
  for j = 0, n                         for j = 0, n
    A(2i,j) = A(2i-2,j)   ===>          A(i,j) = A(i-2,j)
  end for                              end for
end for                              end for
```

Rotation

Iterations in some dimensions can be shifted with respect to other dimensions so that the communication patterns become uniform. It is like executing the iterations in a toroidal fashion. Although it does not change the shape of the iteration space, dependences are changed, and the transformation is not always legal.

```
for i = 0, n                              for i = 0, n
  for j = 0, n                              for j = 0, n
    A(i,j)=         ===>                      A(i,(j-i) mod (n+1)) =
  end for                                   end for
end for                                   end for
```

The elements of A computed will be in the order $(0,0),(0,1),...,(0,n),(1,n),(1,0),...,(1,n-1),(2,n-1),(2,n),...,(2,n-2),...$ instead of usual lexicographic order in the original loop.

Combing

This is equivalent to interchanging the loops of a strip mined loop. It is not always legal.

```
for i = 0, n                               for i = 0, 63
  A(i) = A(i-1)       ===>                   for ii = i, n, 64
end for                                        A(i) = A(i-1)
                                             end for
                                           end for
```

Tiling

Less popularly called blocking, strips different loop levels to decrease the effective sizes of the inner loops so as to increase reuse. It is like increasing the grain size from an iteration to a collection of iterations (tile), outer loops stepping through the tiles and inner loops stepping through the iterations in a tile. Valid only if the loop levels to be tiled are fully permutable.

```
for i = 0, n                         for ii = 0, n, Si
  for j = 0, n                         for jj = 0, n, Sj
     ...            ====>                for i = ii, min (ii+Si-1,n)
  end for                                  for j = jj, min (jj+Sj-1,n)
end for                                                 ...
                                           end for
                                         end for
                                       end for
                                     end for
```

C Secondary Transformations

These are intended as improvements over transformed programs.

- Introducing communication primitives and changing for efficient primitives (like block transfer)
- Prefetching
- Register binding
- Temporaries for index computation
- Propagation of alignments

References

[Ban88] Utpal Banerjee. *Dependence Analysis for Supercomputing.* Kluwer Academic Publishers, 1988.

[Ban90] Utpal Banerjee. Unimodular transformations of double loops. In *Proceedings of Third Workshop on Programming Languages and Compilers for Parallel Computing*, Irvine, CA, August 1990.

[BFKK] V. Balasundaram, G. Fox, Ken Kennedy, and U. Kremer. A static performance estimator to guide data partitioning decisions. In *Proceedings of the 3rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, volume 26.

[CF87] Ron Cytron and Jeanne Ferrante. What's in a name? In *Proceedings of the 1987 International Conference on Parallel Processing*, pages 19–27, 1987.

[D'H89] E.H. D'Hollander. Partitioning and labeling of index sets in do loops with constant dependence vectors. In *Proceedings of the 1989 International Conference on Parallel Processing*, pages 139–144, 1989.

[Dow90] M.L. Dowling. Optimum code parallelization using unimodular transformations. *Parallel Computing*, 16:155–171, 1990.

[GB92] M. Gupta and P. Banerjee. Demonstration of automatic data partitioning techniques for parallelizing compilers on multicomputers. *IEEE Trans. Parallel Distributed Systems*, 3(2):179–193, 1992.

[Ger89] H.M. Gerndt. Automatic parallelization for distributed memory multiprocessing systems. Technical report, PhD thesis, Bonn University, FRG, 1989.

[GJ79] M.R. Garey and D.S. Johnson. *Computers and Intractability, A guide to the theory of NP-Completeness.* W.H. Freeman and Co., 1979.

[HKK+91] S. Hiranandani, K. Kennedy, C. Koelbel, U. Kremer, and C. Tseng. An overview of the fortran d programming system. Technical Report CRPC-TR91121, Dept of computer Science, Rice University, 1991.

[HS92] M. Holliday and M. Stumm. Performance evaluation of hierarchical ring-based shared memory mutliprocessors. Technical Report CS-1992-18, Computer Science Department, Duke University, 1992.

[IT88] F. Irigoin and R. Triolet. Supernode partitioning. In *Conference Record of the 15th Annual ACM Symposium on Principles of Programming Languages*, pages 319–329, San Diego, CA, 1988.

[KKB91] K.G. Kumar, D. Kulkarni, and A. Basu. Generalized unimodular loop transformations for distributed memory multiprocessors. In *Proceedings of the International Conference on Parallel Processing*, Chicago, MI, 1991.

[KKB92] K.G. Kumar, D. Kulkarni, and A. Basu. Deriving good transformations for mapping nested loops on hierarchical parallel machines in polynomial time. In *Proceedings of the 1992 ACM International Conference on Supercomputing*, Washington, 1992.

[KKBP90] K.G. Kumar, D. Kulkarni, A. Basu, and A. Paulraj. Aldims: A language for mimd parallelism on distributed memory multiprocessors. In *PARCOM-90*, Pune, India, 1990.

[KKBP91] D. Kulkarni, K.G. Kumar, A. Basu, and A. Paulraj. Loop partitioning for distributed memory multiprocessors as unimodular transformations. In *Proceedings of the 1991 ACM International Conference on Supercomputing*, Cologne, Germany, 1991.

[KN89] C.T. King and L.M. Ni. Grouping in nested loops for parallel execution on multicomputers. In *Proceedings of the 1989 International Conference on Parallel Processing*, pages 31–38, 1989.

[KSa] D. Kulkarni and M. Stumm. Architecture specific optimal loop transformations. In *In preparation.*

[KSb] D. Kulkarni and M. Stumm. On computation alignment. In *In preparation.*

[Kum93] K.G. Kumar. Personal communication. 1993.

[Lam74] L. Lamport. The parallel execution of do loops. *Communications of the ACM*, 17(2), 1974.

[LC91] J. Li and M. Chen. The data alignment phase in compiling programs for distributed memory machines. *Journal of parallel and distributed computing*, 13:213–221, 1991.

[LP92] W. Li and K. Pingali. A singular loop transformation framework based on non-singular matrices. In *Proceedings of the Fifth Workshop on Programming Languages and Compilers for Parallel Computing*, 1992.

[LYZ90] Z. Li, P. Yew, and C. Zhu. An efficient data dependence analysis for parallelizing compilers. *IEEE Trans. Parallel Distributed Systems*, 1(1):26–34, 1990.

[Mac87] M. Mace. *Memory storage patterns in parallel processing.* Kluwer Academic Publishers, 1987.

[MHL91] 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*, volume 26, pages 1–14, Toronto, Ontario, Canada, 1991.

[OH92] M.F.P O'Boyle and G.A. Hedayat. Data alignment: Transformations to reduce communication on distributed memory architectures. In *Proceedings of the Scalable High Performance Computing Conference, IEE Press*, Williamsburg, 1992.

[PC87] J.K. Peir and R. Cytron. Minimum distance – a method for partitioning recurresnces for multiprocessors. In *Proceedings of the 1987 International Conference on Parallel Processing*, pages 217–224, 1987.

[Pug92] W. Pugh. A practical algorithm for exact array dependence analysis. In *Communications of the ACM*, volume 35, pages 102–114, 1992.

[Ram92a] J. Ramanujam. Non-singular transformations of nested loops. In *Supercomputing 92*, pages 214–223, 1992.

[Ram92b] J. Ramanujam. Tiling of iteration spaces for multicomputers. In *Proceedings of the Supercomputing 1992*, pages 179–186, 1992.

[RS90] J. Ramanujam and P. Sadayappan. Tiling of iteration spaces for multicomputers. In *Proceedings of the 1990 International Conference on Parallel Processing*, pages 179–186, 1990.

[RS91] J. Ramanujam and P. Sadayappan. Compile time techniques for data distribution in distributed memory machines. *IEEE Trans. Parallel Distributed Systems*, 2(4):472–482, 1991.

[Sch86] A. Schrijver. *Theory of linear and integer programming.* Wiley, 1986.

[SF88] W. Shang and J.A.B. Fortes. Independent partitioning of algorithms with uniform dependences. In *Proceedings of the 1988 International Conference on Parallel Processing*, pages 26–33, 1988.

[SZ91] K.C. Sevcik and S. Zhou. Performance benefits and limitations of large numa mutliprocessors. Technical Report CSRI-256, Computer Systems Research Institute, University of Toronto, 1991.

[Who92] S. Wholey. Automatic data mapping for distributed memory parallel computers. In *Proceedings of the 1992 ACM International Conference on Supercomputing*, 1992.

[WL90] M.E. Wolf and M.S. Lam. An algorithmic approach to compound loop transformation. In *Proceedings of Third Workshop on Programming Languages and Compilers for Parallel Computing*, Irvine, CA, 1990.

[WL91] M.E. Wolf and M.S. Lam. A data locality optimizing algorithm. In *Proceedings of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation*, volume 26, pages 30–44, Toronto, Ontario, Canada, 1991.

[WT92] Michael Wolfe and Chau-Wen Tseng. The power test for data dependence. *IEEE Trans. Parallel Distributed Systems*, 3(5):591–601, 1992.