Linear Loop Transformations in Optimizing Compilers for Parallel Machines

Dattatraya Kulkarni*† and Michael Stumm

October 26, 1994

Abstract

We present the linear loop transformation framework which is the formal basis for state of the art optimization techniques in restructuring compilers for parallel machines. The framework unifies most existing transformations and provides a systematic set of code generation techniques for arbitrary compound loop transformations. The algebraic representation of the loop structure and its transformation give way to quantitative techniques for optimizing performance on parallel machines. We discuss in detail the techniques for generating the transformed loop and deriving the desired linear transformation.

Key Words: Dependence Analysis, Iteration Spaces, Parallelism, Locality, Load Balance, Conventional Loop Transformations, Linear Loop Transformations

---

*Corresponding author.

†Parallel Systems Group, Department of Computer Science, 10 King's College Road, University of Toronto, Toronto, ON M5S 1A4, CANADA. Email: *kulki@cs.toronto.edu*

1 Introduction

In order 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 paper deals with the restructuring of nested loops 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 partitioning are computationally hard, efficient approximate solutions often exist. In this paper we describe linear loop transformations, the state of the art loop transformation technique, and their advantages.

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 $\mathtt{A(k, *)}$, where $*$ denotes all elements in that dimension of the array, and element $\mathtt{B(j)}$ is stored in $(j \bmod (m+1))-$th processor's memory.

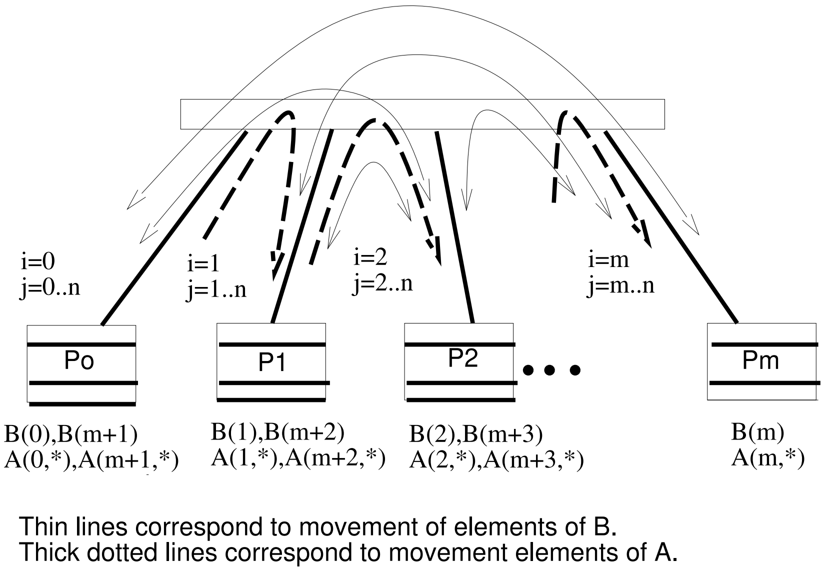

Figure 1: An example mapping

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 $\mathtt{B()}$ must be obtained remotely from other processors and processor k must fetch $\mathtt{A(k-1, *)}$ 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
end for
```

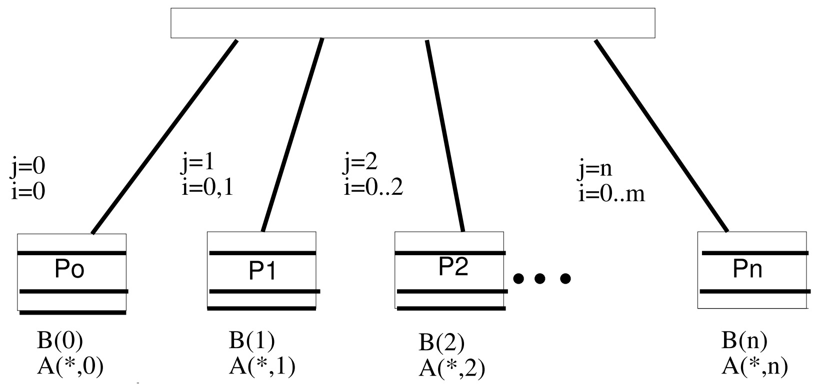

Figure 2: Another mapping

If processor k executes all iterations with $j = k$, and has stored in its local memory `B(k)` and `A(*,k)` then processor k will execute $\min(k, m) + 1$ iterations. Because `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 fact, there is no parallelism in the first mapping. 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 and hence the reuse of accesses to `B` are not exploited.

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 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 considered, such as *replicating* array `B` on all processors. The objective of the restructuring process is to obtain a program that is semantically equivalent, yet performs better on the target parallel machine due to improvements in parallelism, locality, and load balance. 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.

The paper is organized as follows. We discuss the algebraic representation of the loop structure in Section 2. The mathematics of linear loop transformations is presented in Section 3 along with techniques to generate the transformed loop. Section 3 concludes summarizing the advantages of linear transformations over conventional approach. Finding an optimal linear transform is a computationally hard problem, and Section 4 presents two classes of linear transforms aimed at improving parallelism and locality which can be derived in polynomial time. We conclude with Section 5.

2 Representing the Loop Structure

2.1 Affine Loop

Consider the following generic nested loop which serves as a program model for the loop structure.

```
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. 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 loop body 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 functions. A loop of the above form with linear subscript functions is called an *affine loop.*

Definition 1 (Iteration space) $\mathcal{I} \subseteq \mathbf{Z}^n$ *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.*

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 2 (Lexicographic order $<$) [1] *Tuples* $\mathtt{i} = (i_1, \ldots, i_n)$ *and* $\mathtt{j} = (j_1, \ldots, j_n)$ *satisfy* $\mathtt{i} < \mathtt{j}$ *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* $i_k < j_k$*.*

A linear transformation of a loop is a linear transformation of its iteration space.

2.2 Loop Bounds

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

$$S_L \mathtt{I} \geq \mathtt{l}$$

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

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

.

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

$$S = \left[\frac{\mathtt{S_L}}{-\mathtt{S_U}} \right], \quad \mathtt{c} = \left[\begin{array}{c} \mathtt{l} \\ -\mathtt{u} \end{array} \right]$$

the polyhedral shape of the iteration space can now be represented by

$$S \mathtt{I} \geq \mathtt{c}$$

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 $\mathtt{I_2}$ is $\max(\mathtt{I_1}, \mathtt{10} - \mathtt{I_1})$, then $\mathtt{I_2}-\mathtt{I_1} \geq \mathtt{0}$ and $\mathtt{I_2}+\mathtt{I_1} \geq \mathtt{10}$ both belong to the set of inequalities S_L.

[1] 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.

Consider the following doubly nested loop which serves as a running example to illustrate the techniques in linear loop transformations. We refer to this loop as loop `L` in future.

```
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
```

"Loop `L`"

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

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

Thus the loop bounds can be represented by

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

2.3 Data Dependence

The analysis of precedence constraints on the execution of the statements of a program is a fundamental step in parallelizing the program. The dependence relation between two statements constrains the order in which the statements may be executed. 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]. In this paper, we concern ourselves only with data dependence. 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. 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 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. 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 $\mathtt{S}_1$ to $\mathtt{S}_2$ from now on.

Definition 3 (Flow Dependence) *An iteration* $\mathtt{j} \in \mathcal{I}$ *is flow dependent on iteration* $\mathtt{i} \in \mathcal{I}$*, denoted* $\mathtt{i}\ \delta \mathtt{j}$*, iff there exists an element of an array assigned to in* $\mathtt{i}$ *and referenced in* $\mathtt{j}$*, such that* $\mathtt{i} < \mathtt{j}$ *and there is no* $\mathtt{k}, \mathtt{i} < \mathtt{k} < \mathtt{j}$ *with* $\mathtt{k}\delta\mathtt{j}$ *for the same array element.*

The sequential semantics of a loop suggests that a flow dependence is always positive. This property determines the *validity* or *legality* of a loop transformation.

Finding the dependence information information however is a computationally hard problem. Given iterations $\mathtt{i}$ and $\mathtt{j}$, two references to an m-dimensional array $\mathtt{A}$, $\mathtt{A}(f_1(\mathtt{i}), ..., f_m(\mathtt{i}))$ and $\mathtt{A}(g_1(\mathtt{j}), ..., g_m(\mathtt{j}))$ are to the same element iff $f_1(\mathtt{i}) = g_1(\mathtt{j})$ $, ..., f_m(\mathtt{i}) = g_m(\mathtt{j}))$. For conciseness we denote this set of equalities by $F(\mathtt{i}) = G(\mathtt{j})$. The dependence problem can then be stated as follows: Do there exist iterations $\mathtt{i}$ and $\mathtt{j}$ in the iteration space such that $F(\mathtt{i}) = G(\mathtt{j})$? In other words, we need to know whether the following integer programming problem has integer solutions:

$$S\mathtt{i} \geq \mathtt{c}$$

$$S\mathtt{j} \geq \mathtt{c}$$

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

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

Definition 4 (Dependence Distance Vector) *For a pair of iterations* $\mathtt{i} = (i_1, \ldots, i_n)$ *and* $\mathtt{j} = (j_1, \ldots, j_n)$ *such that* $\mathtt{i}\ \delta \mathtt{j}$*, the vector* $\mathtt{j}$ $\mathtt{-i} = (j_1 - i_1, ..., j_n - i_n)$ *is called the dependence distance vector.*

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*. In this paper "dependence" refers to a uniform dependence distance vector. (A vector of $+, -$ or 0 corresponding to whether a dependence component is positive, negative or zero is called a direction vector.)

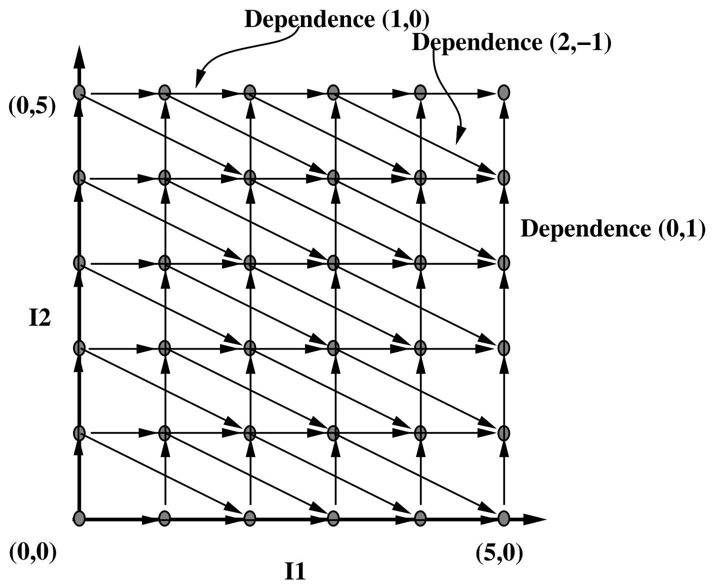

Figure 3: Dependences in iteration space (only a 5×5 space is shown for clarity)

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 *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. That is, each column of D contains a dependence distance vector D_i.

We return to loop `L` and notice that 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. With iterations $\mathtt{i} = (i_1, i_2)$ and $\mathtt{j} = (j_1, j_2)$ such that $\mathtt{i} < \mathtt{j}$, if $i_1 = j_1 - 2$ and $i_2 = j_2 + 1$ then $\mathtt{i}\delta\mathtt{j}$. 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 3 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 have much lower average complexity [Pug92]. A modified *Fourier-Motzkin* variable elimination [Sch86] produces exact solutions in a reasonable amount of time for many problems.

3 Linear Loop Transformations

3.1 Motivation for the Theory

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 linear transformations for loop partitioning were introduced independently by Banerjee [Ban90], and by Wolf and Lam [WL90].

In the conventional approach there exist several loop apparently independent transformations, including *loop interchange*, *loop permutation*, *skew*, *reversal*, *wavefront*, and *tiling*.[2]

Loop interchange[Wol90], 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[Lam74] 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[RS90, WL91, IT88] *strips* several loop levels so that outer loops step through a space of iteration tiles, and inner loops step through the iterations in a tile. Since the block size is smaller than the original loop size, the chances are that the reused data 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.

To fully exploit the features of complex architectures it is generally necessary to apply compound transformations, a combination of two or more of the above 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

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

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.

Linear loop transformations [Ban90, Ban93, Ban94, WL90, Dow90, KKBP91, KKB91, LP92, KP92] unify all possible sequences of most of the existing transformations and alleviate the problems in applying sequences of many different kinds of transformations. A non-singular integer matrix 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 the legality criterion. When this matrix is unimodular it ensures that the mapping is one to one and onto with unit stride. Further elaboration on this aspect and the advantages of the unified approach are discussed in a following subsection.

3.2 Mathematics of Linear Transforms

A loop transformation can be viewed as a reorganization of the loop iterations. It can thus be characterized by a transformation of the iteration space. A linear transformation is defined by an n by n unimodular integer matrix U, which maps the iteration vector $\mathtt{I}$ into a new iteration vector $\mathtt{K} = (K_1, ...K_n)^T$,

$$U\mathtt{I} = \mathtt{K} \tag{1}$$

and each dependence vector D_i in D into a new vector D'_i:

$$UD_i = D'_i \tag{2}$$

The transformation is legal iff each of the D'_i is lexicographically positive. The positivity of transformed dependences is a necessary and sufficient condition to ensure the legality of the transform.

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:

$$U^{-1}\mathtt{K} = \mathtt{I} \tag{3}$$

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

$$S\mathtt{I} \geq \mathtt{c}$$

So,

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

and therefore,

$$S\ U^{-1}\ \mathtt{K} \geq \mathtt{c} \tag{4}$$

This new 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 (in both upper and lower halves) 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.

The basic idea in a variable elimination procedure is as follows. 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, $\beta \leq bK_n$ and $aK_n \geq \alpha$ 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 $a\beta \leq b\alpha$ 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.

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[Ram92] and Kumar[Kum93] compute the new bounds for any non-singular transformation by *stepping aside* the non-existent iterations. The loop strides can be obtained by diagonalizing the transformation. Consider the following loop on the left hand which is linearly transformed to the one on the right hand.

```
for  I1 = n1, N1                    for  K1 = m1, M1
  for  I2 = n2, N2                    for  K2 = m2(K1), M2(K1)
    H(I1, I2)              ==>          H(f(K1, K2), g(K1, K2))
  end for                             end for
end for                             end for
```

For simplicity let the transformation U be unimodular

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

From Equation 1 the new iteration vector is,

$$(K_1, K_2)^T \ = U \ (I_1, I_2)^T \ = (u_{11}I_1 + u_{12}I_2, u_{21}I_1 + u_{22}I_2) \tag{5}$$

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 from Equation 3 we have,

$$(I_1, I_2)^T \ = U^{-1} \ (K_1, K_2)^T \ = (\Delta u_{22}K_1 - \Delta u_{12}K_2, -\Delta u_{21}K_1 + \Delta u_{11}K_2) \tag{6}$$

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 Equation 5 and the upper and lower bounds of I_1 and I_2.

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

$$m_1 = min(u_{11}I_1 + u_{12}I_2, n_1 \leq I_1 \leq N_1, n_2 \leq I_2 \leq N_2)$$

$$M_1 = max(u_{11}I_1 + u_{12}I_2, n_1 \leq I_1 \leq N_1, n_2 \leq I_2 \leq N_2)$$

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

$$\begin{aligned} m_1 &= u_{11}^+ n_1 - u_{11}^- N_1 + u_{12}^+ n_2 - u_{12}^- N_2 \\ M_1 &= u_{11}^+ N_1 - u_{11}^- n_1 + u_{12}^+ N_2 - u_{12}^- n_2 \end{aligned} \tag{7}$$

(since one of 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 $n_2 \leq I_2 \leq N_2$, we can substitute for I_1 and I_2 from Equation 6 to obtain

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

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

This 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

$$m_2(K_1) = \lceil\, max(lb1, lb2)\,\rceil \text{ and } M_2(K_1) = \lfloor\, min(ub1, ub2)\,\rfloor \tag{8}$$

In order to illustrate the techniques consider loop `L` again. We noticed that the dependence matrix for `L` is

$$D = \{(1,0),(0,1),(2,-1)\}$$

Suppose transformation U is applied to `L`.

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

From Equation 3 we have

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

and from Equation 2

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

All the transformed dependences are lexicographically positive, so U is a legal transformation. From Equation 7, K_1 bounds are :

$$\begin{aligned} m_1 &= u_{11}^{+}n_1 - u_{11}^{-}N_1 + u_{12}^{+}n_2 - u_{12}^{-}N_2 \\ &= 1*0 + 1*0 = 0 \end{aligned}$$

and

$$\begin{aligned} M_1 &= u_{11}^{+}N_1 - u_{11}^{-}n_1 + u_{12}^{+}N_2 - u_{12}^{-}n_2 \\ &= 1*10 + 1*10 = 20 \end{aligned}$$

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

$$\begin{aligned} ub1 &= \frac{(n_1 - \Delta u_{22}K_1)}{-\Delta u_{12}} = (0 - K_1)/-1 = K_1 \\ lb1 &= \frac{(N_1 - \Delta u_{22}K_1)}{-\Delta u_{12}} = (10 - K_1)/-1 = K_1 - 10 \\ lb2 &= \frac{(n_2 + \Delta u_{21}K_1)}{\Delta u_{11}} = (0+0)/1 = 0 \\ ub2 &= \frac{(N_2 + \Delta u_{21}K_1)}{\Delta u_{11}} = 10/1 = 10 \end{aligned}$$

The transformed loop is therefore

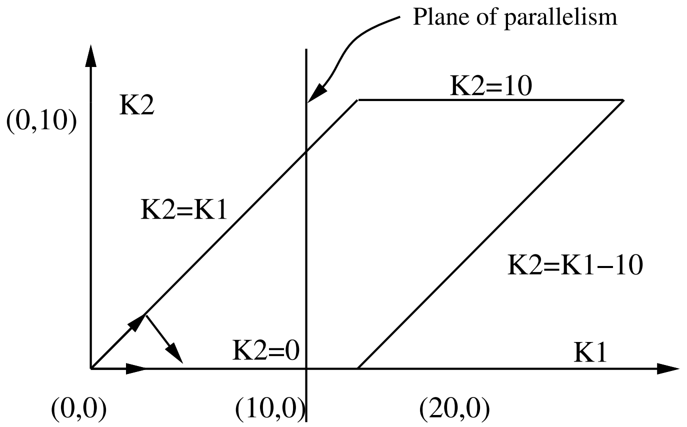

Figure 4: Example Transformation

```
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 4 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.

$$S\mathtt{I} \geq \mathtt{c}$$

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

From Equation 4 we have,

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

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

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

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 5: Example 2d transformations

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,

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

Thus the bounds for K_2

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

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

$$K_1 \ge 0 \text{ and } K_1 \le 20$$

3.3 Advantages of Linear Transforms

Linear loop transformations have several advantages over conventional approach. A linear transformation can be a primary transformation such as an interchange or it can be a combination of two or more primary transformations. Figure 5 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 "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 communication, 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.[3] 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.

3.4 Optimal Linear Transform

Unfortunately, the derivation of a linear transformation 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 matrix. 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

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

following section we discuss two instances of identifying the structure of a transformation given specific goals for the transformed loop.

4 Two Classes of Linear Transforms

4.1 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.[4] 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, the (1,1) dependence in the left hand loop is internalized to obtain the transformed loop on the right hand.

for $i = 0, n$
  for $j = 0, n$
    $A(i, j) = A(i-1, j-1)$
  end for
end for
	$\Longrightarrow$	for $K_1 = -n, n$
		  for $K_2 = \max(0, -K_1), \min(n, n-K_1)$
		    $A(K_1+K_2, K_2) = A(K_1+K_2-1, K_2-1)$
		  end for
		end for

The intended transformation matrix should transform should internalize (1,1), in other words change to (0,1) dependence. This would render a fully parallel outer loop. One unimodular matrix U (of many) that achieves the above internalization $(1,1)$ is:

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

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

The general framework for internalization and Algorithms to find a good internalization in polynomial time can be found in [KKB91, KKB92, KKB].

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[KKB92].

4.2 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. For example, consider the loop on the left hand below. 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  u = p, b - 1, P
  for  j = i, i + b - 1                              for  v = u, u + N1 + N2 - 2
    for  k = 0, N2 - 1                                 for  w = 0, N1 - 1
      B(i, j - i) = B(i, j - i) + A(i, j + k)  ==>       B(w, u) = B(w, u) + A(w, v)
    end for                                            end for
  end for                                            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 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 access normalized loop is shown on the right above. 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].

5 Concluding Remarks

We presented linear loop transformation framework which is the formal basis for state of the art optimization techniques in restructuring compilers for parallel machines. The framework unifies most existing transformations and provides a systematic set of code generation techniques for arbitrary compound transformations. The algebraic representation of the loop structure and its transformation give way to quantitative techniques for optimizing performance on parallel machines. We also discussed in detail the techniques for generating the transformed loop. The framework is extended recently [KSb, KP92] to handle imperfectly nested loops. The new framework [KSb] reorganizes computations at a much finer granularity than existing techniques and helps implement a class of flexible computation rules. Most of the current work of interest involves combining loop transformation, data alignment and partitioning techniques for local and global optimization [AL93, KSb].

Acknowledgements

The first author thanks Utpal Banerjee who provided impetus to his and KG Kumar's joint work. He also thanks KG Kumar for his continued encouragement.

References

[AL93] J. Anderson and M. Lam. Global optimizations for parallelism and locality on scalable parallel machines. In *Proceedings of the ACM SIGPLAN '93 Conference on Programming Language Design and Implementation*, volume 28, June 1993.

[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.

[Ban93] Utpal Banerjee. *Loop Transformations for Restructuring Compilers.* Kluwer Academic Publishers, 1993.

[Ban94] Utpal Banerjee. *Loop Parallelization.* Kluwer Academic Publishers, 1994.

[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.

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

[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.

[KKB] K.G. Kumar, D. Kulkarni, and A. Basu. Mapping nested loops on hierarchical parallel machines using unimodular transformations. *Journal of Parallel and Distributed Computing*, page (revising).

[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, July 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, July 1992.

[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, June 1991.

[KP92] W. Kelly and W. Pugh. A framework for unifying reordering transformations. Technical Report UMIACS-TR-92-126, University of Maryland, 1992.

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

[KSb] D. Kulkarni and M. Stumm. Computational alignment: A new, unified program transformation for local and global optimization. Technical report.

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

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

[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*, August 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.

[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.

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

[Ram92] J. Ramanujam. Non-singular transformations of nested loops. In *Supercomputing 92*, pages 214–223, 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.

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

[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, August 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.

[Wol90] Michael Wolfe. *Optimizing supercompilers for supercomputers.* The MIT Press, 1990.

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