COMPILING FOR NUMA PARALLEL MACHINES

A Dissertation
Presented to the Faculty of the Graduate School
of Cornell University
in Partial Fulfillment of the Requirements for the Degree of
Doctor of Philosophy

by
Wei Li
August 1993

COMPILING FOR

NUMA PARALLEL MACHINES

Wei Li, Ph.D.

Cornell University 1993

A common feature of many scalable parallel machines is non-uniform memory access (NUMA) — data access to local memory is much faster than to non-local memories. In addition, when a number of remote accesses must be made, it is usually more efficient to use block transfers of data rather than to use many small messages. Almost every modern processor is designed with a memory hierarchy organized into several levels – each smaller and faster than the level below. In general, the effective use of parallel machines requires careful attention to the following issues: (1) exposing and exploiting parallelism; (2) accessing local memory instead of remote memory; (3) using block transfers for remote accesses; (4) reusing data in the cache; and (5) load balancing.

We have built a system called *Pnuma* for programming NUMA machines. We make the following contributions: First, we propose a parallelization scheme for both parallelism and data locality. Second, we develop a framework based on *non-singular* matrices and integer lattice theory for the systematic development of loop transformations. Program transformations, such as loop restructuring,

are critical to achieving high performance. The framework can be used in parallelizing compilers for both coarse-grain and fine-grain parallel architectures. We have implemented a loop restructuring tool-kit called *Lambda* based on this framework. Third, using this loop transformation framework, we develop algorithms for improving memory locality. The memory locality algorithm restructures loop nests to expose opportunities for parallel execution and for block transfers, while keeping data accesses local wherever possible. Fourth, for cache locality, we introduce a new simple cache model based on *reuse distances*, which is more precise than the existing *reuse vector space* model. We develop a new loop transformation technique that optimizes directly on reuse distances, so that no exhaustive search is necessary. Fifth, we use our loop transformation framework to improve parallelism as well. We develop a unified algorithm for parallelism, memory locality and cache locality.

System evaluations have been conducted on a multiprocessor machine without cache (BBN GP1000), a uniprocessor workstation with cache (HP 9000/720) and a multiprocessor machine with caches (KSR1), using programs from linear algebra, NASA benchmarks and SIMPLE hydrodynamics benchmark.

To my parents, my sister, Ting, and my teachers

Acknowledgements

It is a great pleasure to work with my advisor, Professor Keshav Pingali. He has the intuition and insights for solving a problem in a simple way. The Pingali-rule, "everything can be reduced to at most 3 points", has helped me always try to understand and tackle a problem from its basics. His intolerance of mediocre research has helped me maintain a decent research standard.

My work would have been impossible without the stimulating and friendly research environment at Cornell. Tom Coleman always found time to listen to my ideas, and helped me learn integer lattice theory. Charlie Van Loan was always happy to answer my questions on matrix computation. His elegant matrix framework for FFT algorithms influenced my work on matrix-based loop transformation framework. Steve Vavasis, Keith Marzullo and Anil Nerode are very kind to have served on my committee. Juris Hartmanis provided guidance and understanding.

I have benefited from discussions with Danny Ralph, Radha Jagadeesan, and Shirish Chinchalkar. My work have been improved by the comments from Micah Beck, Mark Charney, Richard Huff, Richard Johnson, Mayan Moudgill, Anne Rogers, and Paul Stodghill. Sudeep Gupta, Scott DeVine and Nikos Pitsianis helped implement part of Pnuma. Paul Stodghill has been a wonderful officemate for the past two years. He helped in the implementation of an earlier version of the compiler. Richard Huff deserves special thanks for his helpful comments on my papers and my talks. Paul, Sudeep, and Shirish carefully read my thesis draft and provided invaluable comments. I would like to thank Charlie DeVine, Anne Elster, Doug Ierardi, Aiping Liao, Yuying Li, Shmuel Onn, Eva Tardos, Mike Todd, Leslie Trotter, Zhijun Wu, Wei Yuan for listening to my ideas, and patiently answering my random questions.

Helene Croft provided great support for ACRI. Cindy Robinson was always willing to help. Anne Gockel provided excellent systems support.

I will miss the exciting games of badminton, bridge, tennis and volleyball at Cornell. Jiaqi Luo has been a perfect bridge partner. Anil Pannikkat has improved my badminton skills by playing with me. Thomas Bressoud organized the weekly CS/OR volleyball games.

It is impossible to enumerate all the friends I would like to thank. Finally, I would like to thank Ting for her friendship and company during all these years.

During my stay at Cornell, I was supported by a McMullen Fellowship from Cornell University, assistantships from Cornell, and a grant from Hewlett-Packard.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Motivation

There is a wide range of applications that may find parallel machines useful. They come from areas such as linear programming, matrix algebra, computational physics, financial modeling, weather and climate modeling, electromagnetic fields simulation, and numerical aerodynamic simulation [CVL88, EA87,MFL+92].

Scalable parallel machines are often organized as networks of processor-memory pairs; examples of such machines are the BBN Butterfly, the Kendall Square Research KSR1, and multi-computers like the Intel iPSC/i860. These machines are called *non-uniform memory access* (NUMA) machines because a processor can access data in its local memory ten to a thousand times faster than it can access non-local data. For example, in the Kendall Square Research "all-cache" machine, accesses to local memory take 18 cycles, while accesses to non-local memory take 175 cycles [Ken91]. Distributed memory machines like the Intel iPSC/i860 have even greater non-uniformity in access times because access to non-local data must be orchestrated through the exchange of messages [Int91]. If non-local accesses are on the critical path through a program, making these accesses local through proper data management will speed up program execution.

Almost every modern processor is designed with a memory hierarchy organized into several levels – each smaller, faster and more expensive than the level below. Typically, a cache hit takes only one cycle, while a cache miss takes 8-32 cycles [HP90]. Therefore, for good performance, programs must possess *cache locality*.

In general, the effective use of parallel machines requires careful attention to the following issues.

- Parallelism: Unless the algorithm underlying a program has lots of parallelism, it is pointless to run the code on a parallel machine. Moreover, the compiler must be able to expose and exploit this parallelism.

- Memory Locality: Scalable high-performance machines are built as interconnections of processor-memory pairs in which a processor can access its local memory ten to a thousand times faster than non-local memory. Therefore, proper data management to maximize local accesses is essential.

- Block transfers: Communication between processors can be viewed as a pipelined process in which the start-up time is large compared to the time to transfer a unit of data. Therefore, when non-local accesses must be made, it is more efficient to use a single block transfer of data rather than many individual transfers.

- Cache Locality: Each node of a multiprocessor may have data cache that is much faster than memory. Cache locality techniques can be applied to uniprocessors, since almost every modern uniprocessor has a cache. Programs are required to have data locality to achieve high performance.

- Load balancing: It is important to avoid swamping a few processors with work when other processors are idle.

1.2 Thesis Overview

We have built a system called *Pnuma* for programming NUMA machines. Pnuma takes programs written in FORTRAN extended with data distribution information and generates code for parallel machines such as the KSR1 and the BBN Butterfly. The system can also produce uniprocessor code optimized for cache locality.

In the traditional approach to parallelization (pioneered by the Cedar project at Illinois), iterations of the loops in a loop nest are distributed among the processors. Synchronization instructions are introduced to take care of dependences between "iterations". To reduce the amount of synchronization, transformations like loop interchange are performed to move parallel loops outermost [MP87, Pol89]. This 'outside-in' strategy is simple but it does not perform any data management, and may result in many non-local accesses during the execution of the loop nest.

A different approach to compiling is to generate code 'inside-out' using the so-called ownership rule — the owner of the variable on the left-hand side of an assignment statement is responsible for computing the expression on the right-hand side. A processor executes a loop iteration if it has any work to do in the body for that iteration. Although this strategy takes data mappings into account, code generation is very complex, compared to the traditional approach, and the code generated can be very inefficient if the structure of the loop nest does not match the data distribution [RP89,HKT91].

This thesis makes the following contributions:

- We propose a parallelization scheme based on both parallelism and data locality. Loop iterations are distributed according to parallelism and locality. We use data distribution information to drive our loop transformation strategy. Once a loop nest has been transformed for parallelism and data locality, we can generate code by distributing outermost parallel loop iterations among the processors. Data accesses in the resulting code are local wherever possible, and non-local accesses are performed using block transfers if possible.

- We develop a framework based on *non-singular* matrices and integer lattice theory for the systematic development of loop transformations. Program transformations, such as loop restructuring, are critical to achieving high performance. The main benefit of this framework is that it provides an approach to tackling the so-called 'phase-ordering problem' — for many problems where there is no obvious order in which the transformations should be performed, it is often possible to generate a non-singular matrix from which the desired order of loop transformations can be determined easily. This framework can be used in parallelizing compilers for MIMD machines as well as in compilers for fine-grain parallel architectures such as VLIW and superscalar machines. This framework is more general than the *unimodular* loop transformation framework [Ban90,WL91b]. We have implemented a loop restructuring toolkit called *Lambda* based on this framework.

- Using this loop transformation framework, we develop algorithms for improving both memory locality. The memory locality algorithm called *access normalization* restructures loop nests to expose opportunities for parallel execution and for block transfers, while keeping data accesses local wherever possible. Loop restructuring is followed by a code generation phase that generates parallel code and makes use of block transfers.

- For cache locality, we introduce a new simple cache model based on *reuse distances*, which is more precise than the existing *reuse vector space* model [WL91a]. We develop a new loop transformation technique that optimizes directly on reuse distances, so that no exhaustive search is necessary.

- We use our loop transformation framework to improve parallelism as well. Furthermore, we develop a unified algorithm for parallelism, memory locality and cache locality.

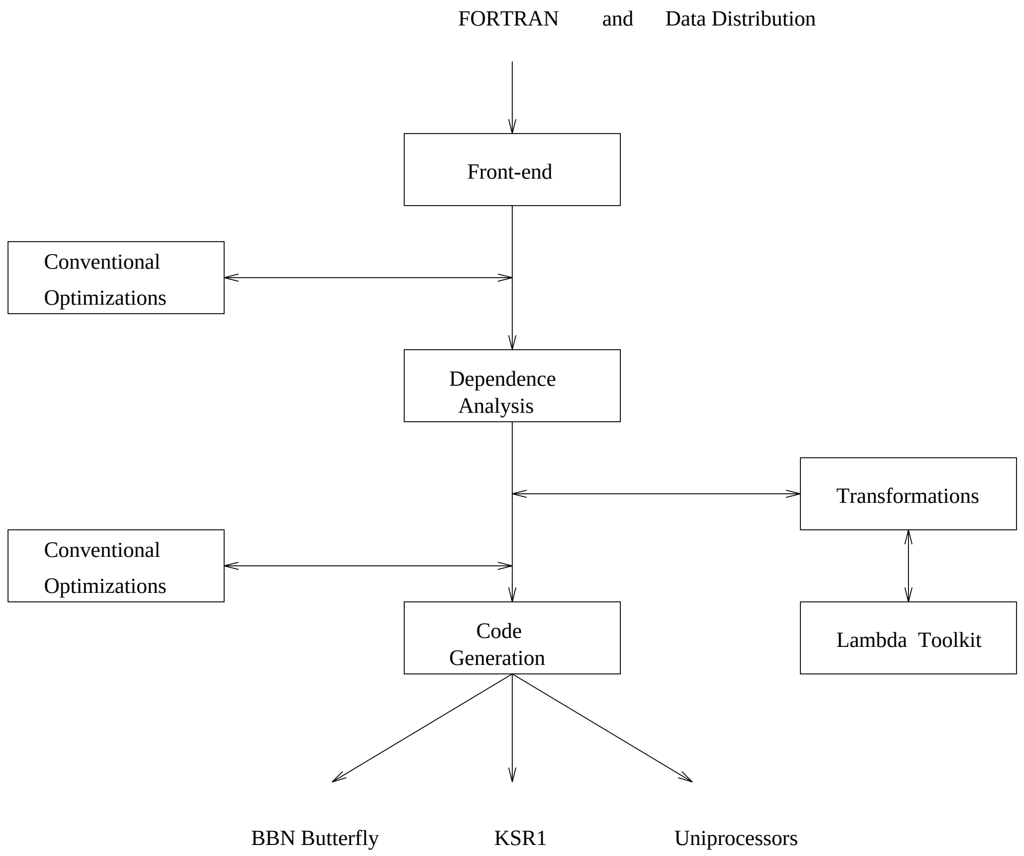


Figure 1.1: *Pnuma* System Overview

Pnuma has been evaluated on a multiprocessor without cache, a uniprocessor workstation with cache and a multiprocessor with caches. Experiments were conducted on BBN GP1000, HP 9000/720 and KSR1 using programs from linear algebra, the NASA benchmarks, and the SIMPLE hydrodynamics benchmark.

The individual modules in the Pnuma system are shown in Figure 1.1. A front-end from SIGMA [GJG88] is used to generate abstract syntax trees with data dependence information from FORTRAN programs. *Access normalization* matches code and data distributions to exploit both parallelism and memory locality. As an added bonus, access normalization exposes opportunities for block transfers of data. For uniprocessor machines, programs are transformed to improve data reuse. For NUMA architectures, programs are transformed to improve parallelism and data locality. We generate parallel FORTRAN for the KSR1, parallel C code for the BBN Butterfly, and optimized FORTRAN for uniprocessor, respectively.

The Lambda toolkit is the implementation of our loop transformation theory.

The toolkit has been integrated with Parascope, the parallelizing environment from Rice University [BP93].

1.3 Organization

The rest of the thesis is organized as follows: in Chapter 2, we describe the locality-driven code generation scheme. We introduce the loop transformation theory in Chapter 3.

The transformation construction algorithms for memory locality are developed in Chapter 4 with the experimental results from a BBN GP1000, which is a multiprocessor machine without caches. Then we develop a cache locality model, and optimization algorithms in Chapter 5 with experimental results from an HP 9000/720, which is a uniprocessor with data cache. The algorithm to improve parallelism, memory and cache locality is presented in Chapter 6 with experimental results from a KSR1, which is a multiprocessor machine with caches.

We conclude, and point out future work in Chapter 7. A brief description of the Pnuma compiler is included in Appendix A, and the implementation of the loop transformation framework called the *Lambda* toolkit is included in Appendix B.

Chapter 2

Generation of Parallel Code

2.1 Introduction

There are two ways of generating parallel code from a sequential program. One way is *control-driven*, i.e. distribute iterations of a loop among the processors. If the loop is a DOALL loop, then no synchronization is needed. Otherwise, synchronization instructions are introduced to take care of dependences carried by this loop, called a DOACROSS loop [MP87,Pol89]. This approach is simple, and has a clean process decomposition, but since it was designed for uniform memory access machines, it does not take data locality into account.

A different approach is *data-driven*, i.e. generate code using the *ownership* rule — the owner of the variable on the left-hand of an assignment statement is responsible for computing the expression on the right-hand side. A processor executes a loop iteration if it has any work to do in the body for that iteration. Although this strategy takes data mappings into account, code generation is very complex, compared to the traditional approach, and the code generated can be very inefficient if the structure of the loop nest does not match the data distribution [RP89,HKT91].

Our approach combines the simplicity of the control-driven approach and the locality sensitivity of the data-driven approach.

2.2 Locality-driven Loop Parallelization

In this section, we first define the data distributions we support, then present our parallelization scheme.

2.2.1 Data Distribution

Our compiler accepts programs written in FORTRAN extended with data distribution declarations that specify how arrays are to be distributed across the local memories of the machine. We support most of the data distributions commonly used by programmers of NUMA machines, such as *wrapped* and *blocked* column and row distributions. In a wrapped column distribution, the columns of an array are distributed in a round-robin fashion to the processors. If P is the number of processors, then processor 0 gets columns *0, P, 2P* and so on, while processor 1 gets columns *1, P+1, 2P+1,* etc. Most of the examples in this paper use a *wrapped column* distribution. Blocked column distribution is similar, except that a processor gets a contiguous set of columns.

Data distributions can be specified precisely using a distribution function.

Definition 2.2.1 A *distribution function* is a function from array indices to integers between 0 and P-1, where P is the number of processors in the machine. An array dimension is a *distribution dimension*, if that dimension is used in the distribution function for the array. The expression in a distribution dimension is called *distribution expression*.

For example, the distribution function for the wrapped column distribution of a two dimensional array is $W_2(i,j) = j\ mod\ P$, the second dimension of the array is a distribution dimension, and the expression in the second dimension is the distribution expression.

2.2.2 Computing Distributed Loops

Our scheme parallelizes the outermost parallel loops that carry data locality.

Definition 2.2.2 A loop *carries* a distribution expression e, if e is a function *uniquely* defined by the loop index variable.

For example, in a loop nest with loops i and j, the distribution expression $i + j$ is carried by loop j but not loop i, since i is a loop invariant in loop j.

A loop may carry multiple expressions. We use a heuristic to decide a unique expression for a loop, and this expression is used in section 2.3 to generate the parallel code if the loop is decided to be parallelized.

The algorithm in Figure 2.2 computes the distribution expressions carried by loops in a loop nest. Let J be the loop index vector of the loop nest. The *Access set* is a set of distribution expressions with three attributes: level, weight, block. *level* is the nesting level of the loops; *weight* is the number of occurrences; and *block* indicates whether this is a wrapped or blocked distribution. If t is a distribution expression, then $t.l, t.w$ and $t.b$ to represent the attributes level, weight and block respectively. Two expressions can be compared by the lexicographical order of $(t.l, t.w)$. Two set operations, union ($\uplus$) and intersection ($\otimes$), on access sets are defined in Figure 2.1.

$$S_1 \uplus S_2 = \{t : t \in S1, t \in S_2, t.b1 = t.b2\}.$$

- $t.b = t.b1$;
- if $(t.l1 > t.l2)$ $t.l = t.l1, t.w = t.w1$;
- if $(t.l1 < t.l2)$ $t.l = t.l2, t.w = t.w2$;
- if $(t.l1 = t.l2)$ $t.l = t.l1, t.w = t.w1 + t.w2$;

$$S_1 \otimes S_2 = \{t : t \in S_1, t \in S_2, t.b1 = t.b2\}.$$

- $t.b = t.b1$;
- if $(t.l1 > t.l2)$ $t.l = t.l1, t.w = t.w1$;
- if $(t.l1 < t.l2)$ $t.l = t.l2, t.w = t.w2$;
- if $(t.l1 = t.l2)$ $t.l = t.l1, t.w = max(t.w1, t.w2)$;

Figure 2.1: Access Set Operators

Access(x)	= $\emptyset$
Access(A[..., f(J),...])	= { f(J) with level=0, weight=1 }
Access(Exp1 op Exp2)	= Access(Exp1) $\uplus$ Access(Exp2)
Access(Var := Exp)	= Access(Exp) $\uplus$ Access(Var)
Access(S1; S2)	= Access(S1) $\uplus$ Access(S2)
Access(if Exp then S1 else S2)	= Access(Exp) $\uplus$ (Access(S1) $\otimes$ Access(S2))
Access(for i=l,u do S)	= choose the expression carried by i-loop with the highest $(t.l, t.w)$ from Access(S); return Access(S) deleting expressions carried by the loop.

Figure 2.2: Computing Distribution Expressions

$$for\ i = l,\ u$$

(a) unit step

$$for\ i = \lceil \frac{l-p}{P} \rceil * P + p,\quad u,\quad step\ P$$

(b) task p for wrapped distribution

$$for\ i = max(l,\ p * S),\quad min(u,\ (p+1) * S - 1)$$

(c) task p for blocked distribution

$$for\ i = l,\ u,\ step\ s$$

(d) non-unit step

$$for\ i = l + n_0 * s,\quad u,\quad step\ P/(P, s) * s$$

(e) task p for wrapped distribution

$$for\ i = max(l,\ p * S),\quad min(u,\ (p+1) * S - 1),\quad step\ s$$

(f) task p for blocked distribution

Figure 2.3: Distributing loops among processors

2.3 Localizing Data Accesses

After the distribution expressions are computed, we must generate the code that will run on each processor. We generate the same code for each processor, but this code is parameterized by the processor number so that each processor does only the work for which it is responsible.

First, consider a loop with step size 1 (Figure 2.3(a)). For a wrapped distribution, processor p owns the *data segments* p, $p + P$, $p + 2P$, .. etc, where a data segment is a column in the wrapped column distribution or a row in the wrapped row distribution. Since the iterations that access the data segments on processor p are assigned to processor p, it is easy to verify that the iterations executed by processor p are the ones shown in Figure 2.3(b). The lower bound $\lceil \frac{l-p}{P} \rceil * P + p$ is the first iteration between l and u that belongs to process p. For a blocked mapping, the corresponding iterations are shown in Figure 2.3(c).

When the step size is not 1 (Figure 2.3(d)), we must solve a linear congruence for the wrapped distribution. Assume that the step size is positive, since the solution can be easily extended to handle the case when the step size is negative. The iterations can be represented by $i = l + n * s$ where n is a parameter with integer values. The iterations that belong to process p are those satisfying the equation $l + n * s = p \pmod{P}$. Using results from number theory, we know

Input: *An equation* $l + n * s = p \pmod{P}$.
Output: *A special solution* $0 \le n_0 < P$.

Algorithm Asolution(l, s, p, P : Integer) : Integer

```
begin
  if gcd(s, P) doesn't divide (p-l) then no solution;

  s0 = s mod P;
  c0 = (p - l) mod P;
  /* solve n * s0 = c0  (mod P) */
  c = c0;
  while ( s0 doesn't divide c )
    c = c + P;
  n0 = c/s0;
  return(n0);
end
```

Figure 2.4: Computing a special solution

that the when the g.c.d. of s and P, written as (s, P), divides $(l - p)$, there is an infinite number of solutions in the form of $n = n_0 + t * P/(s, P)$ for some integer solution $0 \le n_0 < P/(s, P)$ and integer free variable t. However, only certain t's are solutions for the iterations within the loop bounds. Since $l \le i \le u$ and $i = l + (n_0 + t * P/(s, P)) * s$, the range of t is $\lceil \frac{-n_0}{P/(s,P)} \rceil = 0 \le t \le \lfloor \frac{u - l - n_0 * s}{P/(s,P) * s} \rfloor$. Therefore the bounds of the loop for processor p are in Figure 2.3(e). The remaining question is how to compute the special solution n_0. This can be done by the algorithm in Figure 2.4. The equation $s_0 n = c_0 \pmod{P}$ has the same solutions as $l + s * n = p \pmod{P}$. If there is any solution to the equation $s_0 n = c_0 \pmod{P}$ then there is unique solution within $[0, P/(s, P))$ [HW79]. The algorithm finds that solution. For the blocked distribution, the resulting bounds of the loop for processor p are shown in Figure 2.3(f).

For multi-dimensional block distribution, we need to distribute multiple loops (loop *tiling*). For example, an $n \times n$ array A has a subblock distribution where each subblock is of size $b \times b$. Imagine that the processor names are also two dimensional. Then the element $A[i, j]$ is owned by processor $(i/b, j/b)$. If A has more complicated subscripts or the subscripts are not the loop indices of the two outermost loops, access normalization will attempt to normalize the reference to $A[u, v]$ with u and v as the outermost loops.

Given this assignment of iterations to processors, we must generate synchronization instructions to take care of dependences carried by the outermost loop, and insert block transfers wherever possible. These steps are routine [ZC90], and are omitted.

2.4 Discussion and Related Work

In this chapter, we have discussed the locality driven loop scheduling scheme.

Previous work has been focused on trying load balancing and reducing synchronization. A simple scheduling scheme is the dynamic *self scheduling* scheme proposed by Tang and Yew [TY86]. A central work queue of iterations is maintained. Idle processors will remove iterations from the work queue until it is empty. This scheme achieves very good load balancing, since the granularity of the work unit is one iteration. However, the overhead of maintaining the central queue is high. Various other schemes have been proposed to improve upon it. *Guided self-scheduling* by Polychonopoulos [Pol88] schedules a chuck of iterations at a time, where the chuck size depends on the number of iterations left in the queue and the total number of processors. Other scheduling algorithms include *Adaptive guided scheduling* by Eager and Zahorjan [EZ92]. Unfortunately, none of the above algorithms address data locality. The tradeoff between load balancing and locality has been studied by Markatos and LeBlanc [ML92], who showed that locality is more important than load balancing.

Chapter 3

A Loop Transformation Theory

3.1 Introduction

The importance of loop transformations in generating good code for vector and parallel machines is widely recognized [PW86,AK87,Wol89]. A recent advance in this area is the use of *unimodular* matrices to model three important loop transformations — *permutation*, *skewing* and *reversal*. Unimodular matrices have integer entries and a determinant that is 1 or -1; therefore, they are closed under matrix product. It follows that any sequence of these loop transformations can also be represented as a unimodular matrix; conversely, any unimodular matrix can be interpreted as representing a sequence of permutation, skewing and reversal transformations. The main benefit of the unimodular abstraction is that it provides an approach to tackling the so-called 'phase-ordering problem' — for many problems where there is no obvious order in which the transformations should be performed, it is often possible to generate a unimodular matrix from which the desired order of loop transformations can be determined easily. Banerjee has used this framework to address the problem of generating parallel loops [Ban90]; Wolf and Lam have used this framework extensively to address both this problem and that of promoting data reuse for improving cache performance [WL91a,WL91b].

In this chapter, we propose to use *non-singular* matrices, rather than unimodular matrices, as a foundation for modeling loop transformations. Non-singular matrices include unimodular matrices as a special case, and permit us to include a new transformation called *loop scaling* in this framework. Surprisingly, code generation is somewhat more intricate for non-singular matrices than for unimodular matrices, and it is the main concern of this chapter.

Another advantage of our approach is that it is easier to generate non-singular matrices than it is to generate unimodular matrices. A typical algorithm that uses the matrix framework, such as the generation of parallel outermost loops [Ban90] or the exploitation of locality in NUMA architectures [LP93a], determines the first few rows of the matrix, and then 'pads out' the remaining rows to generate a matrix that represents a legal transformation. It is easier to generate a non-singular matrix than a unimodular matrix since there are fewer constraints to be satisfied, and in Chapter 4, we give a completion procedure that produces a non-singular matrix, given the first few rows. This completion procedure is non-trivial since we must ensure that the result matrix respects the dependencies of the loop nest.

The rest of the chapter is organized as follows. In Section 3.2, we define the problem formally, outline our loop restructuring framework and discuss the difficulties in generating the transformed loop nest. In Section 3.3, we sketch the code generation technique for the case of unimodular matrices, and discuss why it cannot be used directly for non-singular matrices. In Section 3.4, we solve the code generation problem for non-singular matrices. The key technical result is that a non-singular matrix can be decomposed into the product of a lower triangular matrix with positive diagonal elements and a unimodular matrix. Using these two matrices, we generate the transformed loop nest. The last section discusses related work.

3.2 Linear Loop Transformations

In this section, we introduce integer lattices as a model of the iteration space of loops, and non-singular matrices as a model of loop transformations.

3.2.1 Iteration Spaces and Integer Lattices

Consider the loop nest in Figure 3.1(a) whose iteration space is shown in Figure 3.1(c). The points in the iteration space of this loop can be modeled as integer vectors in the two dimensional space $\mathbb{Z}^2$, where $\mathbb{Z}$ is the set of integers. For example, the iteration $(i = 2, j = 3)$ can be represented by the vector $(2, 3)$. In general, points in the iteration space of a loop nest of depth n can be represented by integer vectors from the space $\mathbb{Z}^n$. It is convenient to use the theory of *integer lattices* [Cas59] and view the points in the iteration space as being generated by integral linear combinations of a set of basis vectors. For example, it is easy to see that the points in the iteration space shown in Figure 3.1(c) can be generated by integral linear combinations of two integer vectors $\begin{pmatrix} 0 \\ 1 \end{pmatrix}$ and $\begin{pmatrix} 1 \\ 0 \end{pmatrix}$. Similarly, the iteration space in Figure 3.1(d) is generated by linear combinations of the integer vectors $\begin{pmatrix} -2 \\ 1 \end{pmatrix}$ and $\begin{pmatrix} 4 \\ 1 \end{pmatrix}$.

For future reference, we define these concepts more precisely.

Definition 3.2.1 Let $a_1, a_2, ..., a_m$ be a set of linearly independent integer vectors. The set $\Lambda = \{\lambda_1 a_1 + \lambda_2 a_2 + ... + \lambda_m a_m \mid \lambda_1, ..., \lambda_m \in \mathbb{Z}\}$ is called an *integer lattice* generated by the *basis* $a_1, a_2, ..., a_m$.

We will call an integer matrix a *basis matrix*, if its columns are a basis.

The loop nest in Figure 3.1(a) has the property that every integer point within the loop bounds is a point in the iteration space of the loop nest. We will call this a *dense* iteration space. By contrast, Figure 3.1(d) shows a *sparse* iteration space because the integer point (2, 3), for example, is within the loop bounds but does not represent a point in the iteration space of the loop. The notion of *dense* and *sparse* can be formally defined as follows.

Definition 3.2.2 An iteration space is *dense*, if for any two integer vectors v_1 and v_2 representing loop iterations, any integer vector $v_3 = \lambda v_1 + (1 - \lambda)v_2$ for some $0 \leq \lambda \leq 1$ also represents a loop iteration. An iteration space is *sparse* if it is not dense.

The significance of this classification of iterations spaces is that it is considerably more difficult to generate code for a loop nest if the iteration space is sparse, than if it is dense, as we will show in Section 3.3.

Let e_i be an n-dimensional vector with 1 in the ith entry and 0 elsewhere.

Theorem 3.2.1 *The integer vectors from a n-dimensional dense iteration space form an integer lattice with the basis e_1, e_2,... e_n.*

Proof: Obvious. □

3.2.2 Loop Transformations

We will focus on transformations that can be represented by linear, one-to-one mappings from the iteration space of the source program to the iteration space of the target program. This class of transformations includes permutation, skewing and reversal, as well as a new transformation called *scaling*. Examples of these transformations are shown in Figure 3.2. These transformations are standard except for scaling which corresponds to replacing a loop iteration variable by an integer multiple of it. Loop scaling gives the ability to transform sparse iteration spaces, which is important in iteration space tiling. For example, if a two dimensional iteration space is partitioned into 2×2 tiles, each tile can be represented by its bottom-left corner. The space of these representatives is a sparse space, and it can be viewed as the result of loop scaling by a factor of 2. Loop scaling is also useful in *access normalization* [LP92] which is a loop transformation for improving data locality.

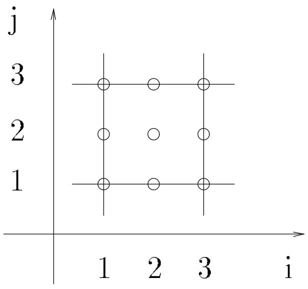
Linear, one-to-one mappings between iteration spaces can be modeled using *integer, non-singular matrices*. The reader can verify that the matrices shown in Figure 3.2 perform the desired mappings from the source iteration space to the target iteration space, and are integer and non-singular. Similarly, in Figure 3.1(d), the points in the target iteration space are the image of the source iteration space points under the integer, non-singular matrix $T = \begin{pmatrix} -2 & 4 \\ 1 & 1 \end{pmatrix}$.

```
for i = 1, 3
  for j = 1, 3
    A[4j-2i+3, i+j] = j;
```

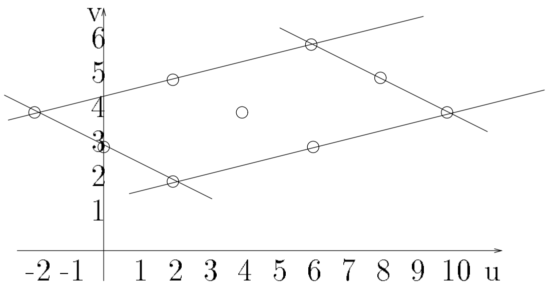
(a) The original code

for u = -2, 10 step 2
  for v = $-\frac{u}{2}$ + 3max(1, $\lceil \frac{\frac{u}{2}+1}{2} \rceil$),
    $-\frac{u}{2}$ + 3min(3, $\lfloor \frac{\frac{u}{2}+3}{2} \rfloor$)
    step 3
  A[u+3, v] = (u + 2v)/6;

(b) The target code



(c) The original iteration space



(d) The target iteration space

$$1 \le \ i \ \le 3$$
$$1 \le \ j \ \le 3$$

(e) Loop Bounds

$$-2 \le u \le 10$$
$$max((u+6)/4, (6-u)/2) \le v$$
$$v \le min((u+18)/4, (18-u)/2)$$

(f) Image of Bounds

Figure 3.1: The working example

```
for i = 1, 3
  for j = 1, 3
    A[i, 2j] = j
```

The original loop nest

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

```
for u = 1, 3
  for v = 1, 3
    A[v, 2u] = u
```

(a) loop interchange

$$\begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix}$$

```
for u = 1, 3
  for v = -3, -1
    A[u, -2v] = -v
```

(b) loop reversal

Figure 3.2: Primitive Transformations

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

```
for u = 1, 3
  for v = u+1, u+3
    A[u, 2(v-u)] = v-u
```

(c) loop skewing

$$\begin{pmatrix} 1 & 0 \\ 0 & 2 \end{pmatrix}$$

```
for u = 1, 3
  for v = 2, 6, 2
    A[u, v] = v/2
```

(d) loop scaling

Figure 3.3: Primitive Transformations (Cont.)

Definition 3.2.3 A loop transformation is called a Λ-*transformation* if it can be modeled by an integer non-singular matrix.

Performing a sequence of transformations corresponds to composing the mappings between iteration spaces, which, in turn, can be modeled as the product of the matrices representing the individual transformations. Since the product of any number of integer, non-singular matrices is integer and non-singular, it follows that the set of Λ-*transformations* is closed under composition.

Conversely, we can show that the transformation represented by any integer non-singular matrix can be viewed as a composition of the four basic transformations. More precisely, we have the following result.

Theorem 3.2.2 *The transformation represented by any integer non-singular matrix can be viewed as a composition of permutation, skewing, reversal and scaling.*

Proof: By applying the appropriate elementary row and column operations, an integer non-singular can be reduced to a diagonal matrix. The elementary operations can be represented by multiplying interchange, reversal and skewing matrices on the left hand side and on the right hand side the non-singular matrix. The diagonal matrix can be further reduced to a product of scaling matrices. $\square$

As mentioned earlier, unimodular transformations are the subset of non-singular transformations without loop scaling. An important property of unimodular transformations is the following.

Theorem 3.2.3 *Unimodular transformations map a dense (sparse) iteration space to another dense (sparse) iteration space.*

Proof: The lattice remains the same, since only the basis is changed. $\square$

3.2.3 Generating Code

To generate code for the target loop nest, we must generate DO-loops that scan the points of the target iteration space in lexicographic order, and replace occurences in the loop body of the old loop indices with the new loop indices. The first problem is non-trivial and is discussed in Sections 3.3 and 3.4. On the other hand, the problem of transforming the loop body is relatively straightforward and we sketch a solution here for completeness. If vectors S_i and S_j represent the source and target iteration variables, notice that $S_i = T^{-1}S_j$. This is just a set of equations expressing the old subscripts in terms of the new ones, and it can be used to eliminate occurrences of the source iteration variables in the body of the loop in favor of the new ones. For our running example, this set of equations is the following:

$$\begin{pmatrix} i \\ j \end{pmatrix} = \begin{pmatrix} -1/6 & 4/6 \\ 1/6 & 2/6 \end{pmatrix} \begin{pmatrix} u \\ v \end{pmatrix}$$

The transformed loop body is shown in Figure 3.1(b).

Note that $T^{-1}S_j$ will always be an integer point even though T^{-1} may be a rational matrix; therefore, expressions like $u/2$ and $(u+2v)/6$ in Figure 3.1(b) should be strength reduced.

3.3 Difficulties in Generating Code

The difficulty in generating DO-loops to scan the target iteration space is that a Λ-transformation, in general, does not preserve lexicographic order (two iterations may be performed in one order in the source loop nest but in a different order in the target loop nest), so there is no obvious way to use the source loop nest to generate code. As a first attempt, we can find the image of the original bounds (the four inclined lines in Figure 3.1(d) for the running example), and then generate a loop nest that visits in lexicographical order all the integer points in the area bounded by the image. In this section, we show that this approach works well when the target iteration space is dense; sparse iteration spaces will require additional machinery.

3.3.1 Computing Image of Bounds

There are many ways to compute the image of the original bounds; here, we describe a simple method that uses Fourier-Motzkin elimination.

Given a non-singular matrix representing the transformation, the image bounds for the target loop can be computed using the inverse of the transformation. Let $S_i = (i_1, .., i_n)^T$ and $S_j = (j_1, .., j_n)^T$ be source and target loop indices respectively under the non-singular transformation T. Let the loop bounds for loop i_k be an affine function of loop indices $i_1, ..i_{(k-1)}$. Each lower bound is in the form of $a_{j1}i_1 + .. + a_{j(k-1)}i_{(k-1)} + b_j \leq i_k$. There may be many such lower bounds whose maximum is the lower bound for i_k. Similarly for upper bounds, there may be many affine bounds whose minimum is the upper bound for i_k. The bounds in the loop nest can be written in the following matrix form:

$$L_b S_i + b_l \leq I_l S_i \quad \text{and} \quad I_u S_i \leq U_b S_i + b_u$$

where $L_b(U_b)$ is an $m_l \times n$ $(m_u \times n)$ matrix, $b_l(b_u)$ is a vector of length $m_l(m_u)$. $I_l(I_u)$ is an identity matrix with some of its rows replicated to an $m_l \times n$ $(m_u \times n)$ matrix. Each row of L_b (U_b) plus the corresponding row from $b_l(b_u)$ form one lower(upper) bound.

Then the source iteration space is bounded by the following inequalities:

$$AS_i \leq b, \quad \text{where} \quad A = \begin{pmatrix} L_b - I_l \\ I_u - U_b \end{pmatrix}, b = \begin{pmatrix} -b_l \\ b_u \end{pmatrix}$$

The bounds for S_j are found by replacing S_i by $T^{-1}S_j$.

$$AT^{-1}S_j \leq b \tag{3.1}$$

These inequalities can not be used directly as loop bounds, since the bounds for a loop can only be a function of outer loop indices. We use the Fourier-Motzkin elimination algorithm [DE73] suggested in [AI91] to compute the suitable bounds. The Fourier-Motzkin algorithm may introduce redundant constraints, but these may be eliminated [AI91].

The Fourier-Motzkin algorithm is quite simple. Consider a system of linear inequalities

$$\textstyle\sum_{j=1}^{n} a_{ij}x_j \leq b_j, \qquad i = (1, .., m).$$

This system can be partitioned into three sets of inequalities according to the sign of the coefficient of x_n.

$$\begin{array}{rcll} x_n & \leq & D_i(\overline{x}), & i = (1, .., p) \\ x_n & \geq & E_j(\overline{x}), & j = (1, .., q) \\ 0 & \leq & F_k(\overline{x}), & k = (1, .., r) \end{array}$$

where D_i, E_j and F_k are linear functions of $\overline{x} = (x_1, ..x_{(n-1)})$.

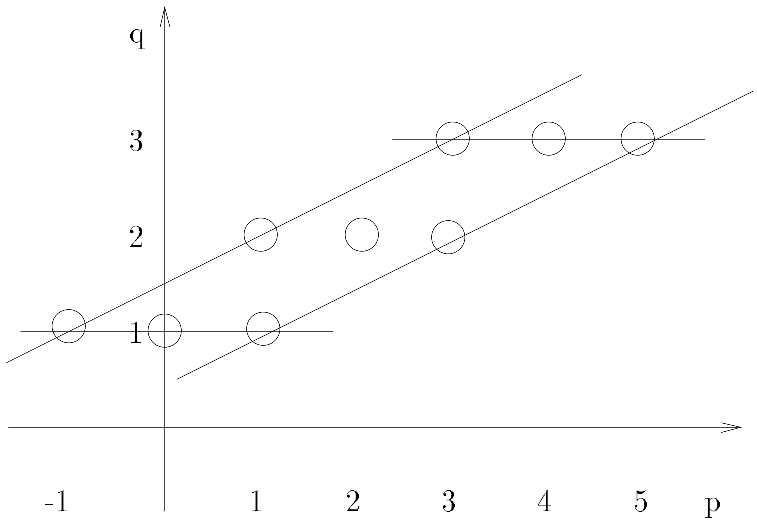
Now, we can eliminate x_n from the system to get the following *reduced system*.

$$\begin{array}{rcll} E_j(\overline{x}) & \leq & D_i(\overline{x}), & i = (1, .., p), \quad j = (1, .., q) \\ 0 & \leq & F_k(\overline{x}), & k = (1, .., r) \end{array}$$

This process can be repeated until there is exactly one variable left. The bounds for this variable can be determined from inspection of the reduced system of equations.

Going back to our problem, consider the system of inequalities for S_j. The loop bounds for j_n can be computed by solving the inequalities for j_n. The bounds for j_k can be computed by first eliminating $j_{(k+1)}, ..j_n$ from the system using Fourier-Motzkin elimination, then solving for j_k etc.

Consider the working example. The iteration space (Figure 3.1(c)) is represented by the integer vectors bounded by the system of linear inequalities in



$$-1 \le p \le 5$$
$$max(1, \tfrac{p+1}{2}) \le q$$
$$q \le min(3, \tfrac{p+3}{2})$$

(a) Image of Bounds

$$-1 \le p \le 5$$
$$max(1, \lceil \tfrac{p+1}{2} \rceil) \le q$$
$$q \le min(3, \lfloor \tfrac{p+3}{2} \rfloor)$$

(b) Exact Bounds

Figure 3.4: Dense Iteration Space

Figure 3.1(e). By computing i, j in terms of u, v, replacing i, j by u, v in the inequalities and using the Fourier-Motzkin elimination, we have the image of the source bounds (Figure 3.1(f)). Unfortunately, we cannot use these inequalities directly to generate code for the target loop nest. There are two problems. First, the lower and upper bounds may not even be integers — for example, when $u = 4$, the lower bound for v is $\frac{5}{2}$. Furthermore, even though the source iteration space is dense, the target iteration space is sparse. This means that we must find some way to skip over points (like (2,3) in our example) that are not in the iteration space of the target loop nest.

3.3.2 Dense Spaces

For the special case when the target iteration space is dense (such as when a unimodular matrix is used to transform a loop nest with a dense iteration space (Theorem 3.2.3)), both these problems can be solved easily. If the target iteration space is dense, there is no need to skip over points that are not in the iteration space of the target loop nest. Furthermore, we can use floor and ceiling operations to get the nearest integers within the image bounds.

For example, consider the unimodular transformation $U = \begin{pmatrix} -1 & 2 \\ 0 & 1 \end{pmatrix}$ on the working example.

$$\begin{pmatrix} p \\ q \end{pmatrix} = U \begin{pmatrix} i \\ j \end{pmatrix}$$

For the source bounds in Figure 3.1(e), we can compute the image bounds shown in Figure 3.4(a). Since the target space is dense, we can use the ceiling and floor operations to compute the exact bounds shown in Figure 3.4(b).

3.3.3 Discussion

For sparse iterations spaces, the ceiling and floor operations cannot solve the problem. For the example in Figure 3.1(d), (4, 3) is the closest integer point to the boundary of v when $u = 4$, but the starting point of the target loop nest is (4, 4). One possibility is to use conditional tests in the loop body to avoid executing the loop body at points that do not correspond to points in the target iteration space. This approach has been used by other researchers [Lu91], but it involves visiting integer points that are not necessary; moreover, the conditional tests are expensive.

3.4 Algorithm for Code Generation

The key insight to solving the general problem is that an integer non-singular matrix T can be decomposed into the product of a lower triangular matrix H

with positive diagonal elements and a unimodular matrix U. This decomposition is related to the *Hermite normal form* of the transformation matrix [Sch86]. We show that if U is used to transform the program, the resulting program executes iterations in the same lexicographic order as the program obtained by using T as the transformation matrix. We also show that the diagonal elements of H correspond to loop step sizes. Putting these observations together gives an algorithm that generates efficient code for the general case of non-singular matrices.

3.4.1 Auxiliary Iteration Space

By applying column operations to an integer non-singular matrix T, we can reduce it to an integer lower triangular matrix with positive diagonal elements. This lower triangular matrix is related to the *Hermite normal form* [Sch86] of the matrix T. It follows that T can be written as the product of a lower triangular matrix H with positive diagonal elements and a unimodular matrix U that represents the composition of the column operations. This decomposition is not unique, but for our purpose, any such decomposition is adequate; to avoid being pedantic, we will abuse terminology and refer to any such H as the Hermite form of the transformation matrix T. Figure 3.5 shows how to compute H and U.

Let $T = HU$, and let the source space be S_i, and the target space be S_j. Define $S_k = US_i$. Then,

$$S_j = TS_i = HUS_i = HS_k$$

Definition 3.4.1 The iteration space S_k is called the *auxiliary iteration space* of S_i with respect to the decomposition HU.

Theorem 3.4.1 *The auxiliary iteration space is a dense space if the source space is dense.*

Proof: Follows from Theorem 3.2.3 since U is unimodular. $\Box$

Therefore the exact loop bounds of the auxiliary space can be computed using the algorithm in Section 3.3.

An important property of the auxiliary space is that it executes iterations in the same lexicographic order as the target iteration space. To see this, consider our running example.

$$T = \begin{pmatrix} -2 & 4 \\ 1 & 1 \end{pmatrix} \quad H = \begin{pmatrix} 2 & 0 \\ -1 & 3 \end{pmatrix} \quad U = \begin{pmatrix} -1 & 2 \\ 0 & 1 \end{pmatrix}$$

Input: *An $n \times n$ integer matrix T.*
Output: *The Hermite form H of T and a unimodular matrix U.*

Algorithm Hermite

begin
  $U = I$, *where* I *is an* $n \times n$ *identity matrix.*
  For i = *1 to* n *do*
    /* *Consider the submatrix T[i:n, i:n]* */
      While $T[i, i{+}1{:}n] \neq \vec{0}$ *do*
        Apply elementary column operations U_e *to make T[i, i]*
          positive and T[i, i+1:n] zero.
        $U = U_e^{-1} U$
      End-While
  End-For
  $H = T$
end

Figure 3.5: Computing the Hermite form

H is lower triangular with positive diagonal elements, and U is unimodular. Consider using U to transform the source program.

$$\begin{pmatrix} p \\ q \end{pmatrix} = U \begin{pmatrix} i \\ j \end{pmatrix}$$

This is the unimodular transformation considered in Figure 3.4 in Section 3.3, and the bounds of the auxiliary iteration space are shown in Figure 3.4(b). To develop the readers insight into the relationship between the source, auxiliary and target iteration spaces, the mappings between these spaces are shown below — notice that both (p, q) and (u, v) are traversed in the same lexicographical order, but that this order is different from that of the source. This can also be seen by comparing the iteration space diagrams in Figures 3.1(d) and 3.4.

$$\begin{array}{ccccc}
(i, j) & \rightarrow & (p, q) & \rightarrow & (u, v) \\
(3,\ 1) & \rightarrow & (\text{-}1,\ 1) & \rightarrow & (\text{-}2,\ 4) \\
(2,\ 1) & \rightarrow & (0,\ 1) & \rightarrow & (0,\ 3) \\
(1,\ 1) & \rightarrow & (1,\ 1) & \rightarrow & (2,\ 2) \\
(3,\ 2) & \rightarrow & (1,\ 2) & \rightarrow & (2,\ 5) \\
(2,\ 2) & \rightarrow & (2,\ 2) & \rightarrow & (4,\ 4) \\
(1,\ 2) & \rightarrow & (3,\ 2) & \rightarrow & (6,\ 3) \\
(3,\ 3) & \rightarrow & (3,\ 3) & \rightarrow & (6,\ 6) \\
(2,\ 3) & \rightarrow & (4,\ 3) & \rightarrow & (8,\ 5) \\
(1,\ 3) & \rightarrow & (5,\ 3) & \rightarrow & (10,\ 4)
\end{array}$$

To show that this property is true in general, let $\prec$ be the lexicographical order.

Theorem 3.4.2 *If the auxiliary iteration space is traversed in the lexicographical order, then the target iteration space is also traversed in the lexicographical order.*

Proof: $S_j = HS_k$, where H is a lower triangular matrix with positive diagonal. Let $\vec{k_1} \prec \vec{k_2}$ be two iterations in the auxiliary iteration space, and $\vec{d_1} = \vec{k_2} - \vec{k_1}$ be the distance of the two vectors. Clearly $\vec{d_1} \succ 0$. To see that the lexicographical order is preserved, consider the new distance $\vec{d_2}$.

$$\vec{d_2} = \vec{j_2} - \vec{j_1} = H\vec{k_2} - H\vec{k_1} = H\vec{d_1}$$

If $\vec{d_1}(i)$, the ith element of $\vec{d_1}$, is the leading nonzero, $\vec{d_1}(i)$ must be positive, since $\vec{d_1} \succ 0$. Then the leading nonzero of $\vec{d_2}$ is $h_{ii}\vec{d_1}(i)$, which is also positive. Therefore $\vec{d_2} \succ 0$, and $\vec{j_1} \prec \vec{j_2}$. $\square$

This result yields a technique for code generation - decompose T into HU, generate the DO-loops for traversing the auxiliary space using the technique of Section 3.3 (or any other technique that works for unimodular matrices) and compute the target iteration space variables in the loop body. Using the bounds for the auxiliary space computed earlier, the target code for our running example is the following:

for p = -1, 5
 for q = max(1, $\lceil\frac{p+1}{2}\rceil$*), min(3,* $\lfloor\frac{p+3}{2}\rfloor$*)*
  $\begin{pmatrix} u \\ v \end{pmatrix} = \begin{pmatrix} 2 & 0 \\ -1 & 3 \end{pmatrix} \begin{pmatrix} p \\ q \end{pmatrix}$
  A[u+3,v] = (u+2v)/6;

Although this code avoids making conditional tests, it can be improved considerably. Notice that the computation of u is invariant in the inner loop; moreover, u is a linear function of the outer loop index and it can be strength reduced. Similarly, v is a linear function of p and q and it can be strength reduced. Although such optimizations can be left to a later optimization phase, it is preferable to use the induction variables u and v directly as the loop control variables instead of p and q. We show how to do this next.

3.4.2 Target Iteration Space

Since H is lower triangular, it is easy to convert the bounds in the auxiliary space into bounds in the target space. For our example, the relation between these two spaces is given by the following equation:

$$\begin{pmatrix} u \\ v \end{pmatrix} = \begin{pmatrix} 2 & 0 \\ -1 & 3 \end{pmatrix} \begin{pmatrix} p \\ q \end{pmatrix}$$

From the first equation, it follows that the bounds for u are the bounds of p multiplied by 2. Therefore, the bounds for u are the following:

$$-2 \leq u \leq 10$$

The bounds for v are the bounds of q multiplied by 3 with the offset $-p$ which is $-\frac{u}{2}$. Therefore, the bounds for v are the following:

$$-\frac{u}{2} + 3max(1, \lceil\frac{\frac{u}{2}+1}{2}\rceil) \leq v \leq -\frac{u}{2} + 3min(3, \lfloor\frac{\frac{u}{2}+3}{2}\rfloor)$$

These bounds on u are constant, and the bounds on v depend only on u. Therefore, these bounds can be used directly to construct the loop nest, as is shown in Figure 3.1(b). The general algorithm is given in Figure 3.6.

The proof of correctness of this algorithm depends on the following lemma and the fact that the diagonal elements of H are positive, and is omitted.

Input: *The Hermite form H, and the bounds of the auxiliary space S_k*
Output: *The bounds of the target space S_j.*

Algorithm Bounds(l^k, u^k) : (l^j, u^j)

begin
  $S_k = H^{-1}S_j$
  For i = 1 to n do
    /* Compute the offset by replacing k_1, .., $k_{(i-1)}$ by j_1, .. $j_{(i-1)}$ */
    $v_i = h_{i1}k_1 + .. + h_{i(i-1)}k_{(i-1)} = f_i(j_1, .., j_{(i-1)})$

    /* Compute lower bound with k_1, .., $k_{(i-1)}$ in l_i^k
      replaced by j_1, .. $j_{(i-1)}$ using $S_k = H^{-1}S_j$. */
    $l_i^j = v_i + h_{ii}l_i^k$

    /* Compute upper bound with k_1, .., $k_{(i-1)}$ in u_i^k
      replaced by j_1, .. $j_{(i-1)}$ using $S_k = H^{-1}S_j$. */
    $u_i^j = v_i + h_{ii}u_i^k$
  End-For
end

Figure 3.6: Computing the loop bounds

Lemma 3.4.1 *Let P1 = $(j_1, ..., j_n)$ be a point in the target iteration space that is the image of a point P2 = $(k_1, ..., k_n)$ in the auxiliary space. Any co-ordinate j_i can be written as a function of $k_1, ..., k_i$; similarly, k_i can be written as a function of $j_1, ..., j_i$.*

Proof: Follows from the observation that any leading principal sub-matrix $H[1:i, 1:i]$ of H is lower triangular and non-singular, and that the inverse of a lower triangular matrix is also lower triangular [GVL89]. □

To complete the generation of code, we need to skip over points within the bounds that are not in the target iteration space. This would be difficult to do if these points appeared in some irregular pattern within the loop nest bounds; fortunately, we can show that this is not the case. In fact, we show that it suffices to use DO-loops with constant step sizes, and that these step sizes are the integers in the diagonal of the Hermite form.

For the working example, H has the diagonal $[2, 3]$, which means that the loop step is 2 for the outer loop, and 3 for the inner loop. More generally, we have the following theorem.

Theorem 3.4.3 *The positive integers on the diagonal of the Hermite form are the gaps in each dimension.*

Proof: Consider two points P1 = $(k_1, k_2, ...k_i, k_{(i+1)}...k_n)$ and P2 = $(k_1, k_2, ...k_i + 1, a_{(i+1)}...a_n)$ in the auxiliary space which have the same co-ordinates in the first $i-1$ dimensions. Let P3 and P4 be their images in the target space. From Lemma 3.4.1, it follows that P3 and P4 have the same co-ordinate for the first $(i-1)$ dimensions; moreover, their difference in the ith dimension is h_{ii} since H is lower triangular. □

Hence a loop nest can be constructed to traverse the target space directly. For our running example, the target program is the following:

for u = -2, 10 step 2
  for v = $-\frac{u}{2} + 3max(1, \lceil\frac{\frac{u}{2}+1}{2}\rceil), -\frac{u}{2} + 3min(3, \lfloor\frac{\frac{u}{2}+3}{2}\rfloor)$ *step 3*
    A[u+3,v] = (u+2v)/6;

Notice the auxiliary space is used only to compute the bounds for the target space.

3.4.3 Sparse Source Iteration Space

So far, we have considered only the case when the source iteration space is dense. Our technique also works when the source iteration S_i is sparse as long as the source space is *regular*. A *regular* sparse space is one that can be represented by an integer lattice Λ_i. The *base* space S_b is always dense. The lattice basis Λ_i can be thought of as a Λ-transformation from S_b to S_i. The bounds for S_b can be computed by the Fourier-Motzkin elimination. Then any Λ-transformation T on S_i can be considered as a new Λ-trasnformation $T\Lambda_i$ on the base space S_b, since Λ-transformations are closed under composition.

This observation lets us handle source loops in which lower bounds are affine functions of loop indices, upper bounds are piece-wise affine functions of loop indices and step sizes are constant, as shown below.

Theorem 3.4.4 *The following loop nest with constant steps is regular.*

for $i_1 = 0$ to ub_1 step s_1
  for $i_2 = a_{21}i_1$ to ub_2 step s_2
    ...
      for $i_n = a_{n1}i_1 + a_{n2}i_2 + ... + a_{n(n-1)}i_{n-1}$ to ub_n step s_n

Proof: Without loss of generality, the loop nest can be *shifted* so that $\vec{0}$ is a loop point. This just changes the *origin* of the lattice. It is easy to show that the triangular matrix S whose jth row comes from the coefficients of the lower bound and the loop step of the jth loop forms a basis for the lattice of the loop points.

$$S = \begin{pmatrix} s_1 & 0 & . & 0 \\ a_{21}s_1 & s_2 & . & 0 \\ . & . & . & . \\ a_{n1}s_1 & a_{n2}s_2 & . & s_n \end{pmatrix}$$

3.5 Data Dependences

Not every Λ-transformation is valid with respect to the data dependencies in the original loop nest. Data dependencies can be represented by *distance* or *direction* vectors that are lexicographically positive. For example, a distance vector $d = \begin{pmatrix} 3 \\ 2 \end{pmatrix}$ means that the iteration (i, j) depends on the iteration $(i-3, j-2)$.

3.5.1 A Dependence Algebra

In this section, we define an algebra on data dependence vectors.

There are three kinds of data dependences between statements. A *data flow-dependence* occurs when a value computed in one statement is used in another statement. A *data anti-dependence* occurs when a variable used in one statement before being reassigned by another statement. A *data output-dependence* occurs

Table for Addition									
$+$	0	p	n	$<$	$\leq$	$>$	$\geq$	$\neq$	$*$
a	a	$a+p$	$a+n$						
$<$	$<$	$<$	$*$	$<$					
$\leq$	$\leq$	$<$	$*$	$\leq$	$\leq$				
$>$	$>$	$*$	$>$	$*$	$*$	$>$			
$\geq$	$\geq$	$*$	$>$	$*$	$*$	$\geq$	$\geq$		
$\neq$	$\neq$	$*$	$*$	$\neq$	$*$	$\neq$	$*$	$\neq$	
$*$	$*$	$*$	$*$	$*$	$*$	$*$	$*$	$*$	$*$

Table for Multiplication			
$\times$	0	p	n
c	0	$c \times p$	$c \times n$
$<$	0	$<$	$>$
$\leq$	0	$\leq$	$\geq$
$>$	0	$>$	$<$
$\geq$	0	$\geq$	$\leq$
$\neq$	0	$\neq$	$\neq$
$*$	0	$*$	$*$

Figure 3.7: Operators for Directions

when a variable is computed before being recomputed by another statement. When the exact distance is unknown at compile-time, direction vectors provide a conservative approximation.

Definition 3.5.1 A *distance* is an integer, and a *direction* can be one of “$<$”, “$>$” ,”$=$”, “$<=$”, “$>=$″, “$<>$” and “$*$”.

For instances, “$<$” means that the distance is positive; “$>$” negative, and “$*$” unknown.

A data dependence in the loop nest of depth n is represented by a vector of distances or directions. For example, the distance vector $\begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix}$ tell us that the dependence is between successive iterations of the innermost loop. A dependence vector has the property that its leading non-zero is always positive; a legal transformation must preserve this property for each dependence, since the source of the dependence must be executed before its destination. More information on data dependences and techniques of dependence analysis can be found in [Ban88]. A direction vector can be $(< \; > \; =)$ or $(= \; < \; *)$, as long as the leading nonzero is positive.

Definition 3.5.2 A *dependence matrix* is a matrix whose columns are dependence vectors.

In order to describe the computation over these direction symbols, we define operations similar to addition and multiplication on integers. In Figure 3.7, let c and a be any constant, p be a positive and n be a negative.

3.5.2 Legality of Λ-transformations

Once we have the dependence algebra, the legality test for a given non-singular transformation T is quite simple. For a dependence vector d in the original loop nest, Td is the dependence vector in the new iteration space, since T is linear. A transformation T is legal if and only if Td is lexicographically positive.

3.6 Discussion and Related Work

We have introduced a loop transformation framework called Λ-transformations based on integer non-singular matrices. Efficient code can be generated for target loop nests using integer lattice theory. We have also presented a simple completion algorithm that generates correct transformations from partial transformations.

The use of compound loop transformations to parallelize loop nests goes back to Lamport's hyperplane method [Lam74]. These compound transformations can be viewed as compositions of loop interchange, skewing and reversal. Unimodular matrices were used by Dowling to parallelize loop nests [Dow90]. The unimodular framework was further developed by Banerjee [Ban90], and Wolf and Lam [WL91b]. Ancourt and Irigoin [AI91] have developed algorithms for scanning polyhedra using loop nests. Other approaches to extending the unimodular framework can be found in Lu and Chen [Lu91], Ramanujam [Ram92], and Barnett and Lengauer [BL92]. We are not aware of any prior work on general completion procedures.

Chapter 4

Transformations for Memory Locality

4.1 Introduction

Scalable parallel machines are often organized as networks of processor-memory pairs; examples of such machines are the BBN TC 2000, the Kendall Square Research 'all-cache' machine, and multi-computers like the Intel iPSC/i860. These machines are called *non-uniform memory access* (NUMA) machines because a processor can access data in its local memory ten to a thousand times faster than it can access non-local data. For example, in the Kendall Square Research 'all-cache' machine, accesses to local memory take 18 cycles while accesses to non-local memory take 175 cycles [Ken91]. Distributed memory machines like the Intel iPSC/i860 have even greater non-uniformity in access times because access to non-local data must be orchestrated through the exchange of messages [Int91]. If non-local accesses are on the critical path through a program, making these accesses local through proper data management will speed up program execution.

A second feature of most NUMA architectures is that block transfer of data between processors is more efficient than sending this data using many small messages. Data transfer between processors can be viewed as a pipeline with a large setup time compared to the time per stage. For example, on the Intel iPSC/i860, it takes 70 microseconds to start up communication, but it takes only 1 microsecond to transfer a double precision floating point number between nearest neighbors once the communication has been setup [Rue]. Therefore, when a number of data items must be sent from one processor to another, it is preferable to use a single long message to amortize startup time.

Contention in the network has the effect of increasing the expected latency of non-local references; therefore, data management to avoid non-local references has the added benefit of reducing contention, thereby improving performance.

Interestingly, analytical studies show that long messages can increase the latency of non-local accesses [Aga91]. This is an argument against long messages, but on current machines, this effect seems to be of secondary importance compared to the benefits of amortizing start-up time, as we show in Section 4.7.

In this chapter, we present a systematic approach to loop restructuring for parallel machines with a memory hierarchy. As in the ownership approach, our starting point is a language like FORTRAN-D with user-specified data decomposition. We use the data distribution information to drive *access normalization*. The objective of the restructuring is to transform loop nests so that code can be generated by distributing iterations of the outermost parallel loops among the processors without compromising locality. The structure of inner loops is chosen so that data can be transferred using block transfers wherever possible.

The rest of the chapter is organized as follows. In Section 4.2, we discuss a simple example that gives an overview of our compiling strategy. We also introduce the *data access matrix*, which plays a key role in the development. For some programs, the data access matrix is non-singular and can be used directly to transform the loop nest, as we show in Section 4.3. In general, however, this matrix may not be non-singular, and the techniques of Section 4.4 must be used to produce a non-singular matrix for the transformation. The final problem is guaranteeing that the transformation respects program dependences; this is done in Section 4.5. We present experimental results in Section 4.7 that demonstrate that our methods work well on programs of practical interest such as routines from the BLAS (Basic Linear Algebra Subroutines) library [CVL88]. Finally, we discuss related work in Section 4.8.

4.2 Data Access Matrix

In this section, we introduce a key data structure called the *data access matrix*.

To understand the need for loop restructuring, consider the program in Figure 4.1(a), which is a simplified version of the SYR2K code discussed in Section 4.7. Assume that both A and B have a wrapped column distribution. Distributing iterations of the outer loop among the processors (Figure 4.1(b)) results in processor p executing iterations p, $p + P$, etc. Consider accesses to elements of array B. Each iteration of the outer loop makes $N_2(b-b/P)$ non-local accesses, and the total number of non-local accesses is $N_1N_2b(1 - 1/P)$.

The ownership rule uses data decomposition information to generate code. A processor is involved in the execution of an iteration (i, j, k) if it owns any of the elements referenced in the body of the loop in that iteration. Therefore, processor p has work to do in iteration (i, j, k) if $(j - i)$ mod $P = p$ (it must update an element of B) or if $(j + k)$ mod $P = p$ (it must send an element of A to whichever processor is updating B in that iteration). This is accomplished by placing these conditional tests in front of the statement, and having all the

```
for i = 0, N_1 - 1
  for j = i, i+b-1
    for k = 0, N_2 - 1
      B[i, j-i] = B[i, j-i] + A[i, j+k]
```

(a)

```
for i = p, N_1 - 1, step P
  for j = i, i+b-1
    for k = 0, N_2 - 1
      B[i, j-i] = B[i, j-i] + A[i, j+k]
```

(b)

```
for u = 0, b-1
  for v = u, u + N_1 + N_2 - 2
    for w = 0, N_1 - 1
      B[w, u] = B[w, u] + A[w, v]
```

(c)

```
for u = p, b-1, step P
  for v = u, u + N_1 + N_2 - 2
    read A[*,v];
    for w = 0, N_1 - 1
      B[w, u] = B[w, u] + A[w, v]
```

(d)

Figure 4.1: Transformation and Code Generation for a Simple Example

processors execute all iterations 'looking for work to do' [CK88,ZC90]. In simple programs, these conditional tests can be optimized away, but in general they must be executed at runtime, which is inefficient. Moreover, in our program, the code cannot make use of block transfers of elements of A since the elements of A referenced during one iteration of the j loop are referenced by different processors.

Now, consider the program of Figure 4.1(c). This program computes the same function as Figure 4.1(a), but if we distribute the outermost loop among the processors as before (Figure 4.1(d)), there are no non-local accesses to B. There are non-local accesses to A but these can be performed using block transfers. The loop transformations described in this paper transform the program of Figure 4.1(a) to that of Figure 4.1(c). Given the transformed program, the code generation techniques described in Chapter 2 generate the parallel code shown in Figure 4.1(d).

Since the transformations are driven by the data access patterns, it is convenient to define a data structure to represent array subscripts in a loop nest in a convenient way. This data structure is called the *data access matrix*. It is used by our loop restructuring system as the starting point for determining what transformations to apply to the loop nest. For the loop nest in Figure 4.1(a), the data access matrix is

$$\begin{pmatrix} -1 & 1 & 0 \\ 0 & 1 & 1 \\ 1 & 0 & 0 \end{pmatrix}.$$

for i = *0*, $N_1 - 1$	*for* u = *0*, *b-1*
for j = i, *i+b-1*	*for* $v = u$, $u + N_1 + N_2 - 2$
for k = *0*, $N_2 - 1$	*for* w = *0*, $N_1 - 1$
B[i, j-i] = B[i, j-i] + A[i, j+k]	*B[w, u] = B[w, u] + A[w, v]*
(a) Source Program	(b) Restructured Program

Figure 4.2: The running example

This matrix represents the subscripts in the sense that the product of the data access matrix with the column vector $[i, j, k]^T$ yields a column vector in which each element is a subscript from the program. For our example, this product is the column vector $[j - i, j + k, i]^T$ which corresponds to the three subscripts of the program. Constants in a subscript are omitted from the corresponding entry in the data access matrix.

The order in which these subscripts are represented in the data access matrix is important and corresponds to an estimate of their relative importance for achieving good performance. A reasonable heuristic is to give highest importance to subscripts in the distribution dimension(s) of arrays; in our example, the subscripts $j - i$ and $j + k$ dominate the subscript i since they occur in the distribution dimensions of arrays B and A. Notice that $j - i$ occurs twice, but $j + k$ occurs only once. Therefore, we let $j - i$ dominate $j + k$. This yields the data access matrix shown above.

The technical development in the rest of the paper is independent of how subscripts were ordered to obtain the data access matrix. In addition, a subscript that is 'overly complex' for any reason (such as a non-linear function of loop indices) may be omitted from the data access matrix without affecting correctness.

4.3 Non-singular Data Access Matrices

In this section, we consider the simple case where the data access matrix is non-singular. Consider the program of Figure 4.1 again, which is reproduced in Figure 4.2(a,b) for convenience.

The data access matrix for the program is

$$X = \begin{pmatrix} -1 & 1 & 0 \\ 0 & 1 & 1 \\ 1 & 0 & 0 \end{pmatrix}.$$

It is easy to verify that X is non-singular, and that the program shown in Figure 4.2(b) is the result of transforming the source program using X as the transformation matrix. Consider what happens when code is generated for the new loop nest by distributing iterations of the outermost loop among the processors in a round-robin manner. Since the outermost loop index is also the the subscript of the distribution dimension of array B, all references to B will be purely local. We cannot accomplish this for both A and B simultaneously since the subscripts in the distribution dimensions of A and B are different; therefore, there will be non-local accesses to A. However, since the subscript in the distribution dimension of the reference to A was placed second in the data access matrix, this subscript in the new loop nest corresponds to the second loop index and we can perform block transfers for accesses to A, as was shown in Figure 4.1(d).

For future reference, we define the following notion.

Definition 4.3.1 Given an array reference, an array subscript is *normal* with respect to loop i, if it is equal to the loop index variable i.

In this example, the data access matrix yielded the transformation without any complications. This is not the case in general. First, the data access matrix may not be non-singular. We handle this case in Section 4.4. Second, the transformation suggested by the data access matrix may violate one or more data dependences. We take care of this problem in Section 4.5. In both cases, the goal is to produce an non-singular matrix that retains as many rows of the data access matrix as possible.

4.4 Singular Data Access Matrices

In general, the data access matrix is not non-singular, so it cannot be used directly to transform the loop nest. The techniques in this section convert such a matrix into a non-singular matrix that retains as many rows (subscripts) of the data access matrix as possible. This is done in two stages — first, we eliminate linearly dependent rows from the data access matrix using Algorithm *BasisMatrix*, and second, we pad this reduced matrix with additional rows using Algorithm *Padding*, to get a matrix that is non-singular.

4.4.1 Basis Matrix

It is easy to design an inefficient algorithm that takes a data access matrix and selects as many linearly independent rows as possible: we simply go down the rows of the matrix in sequence, discarding a row if it is linearly dependent on the rows before it, and keeping it otherwise. It is important to traverse the rows in sequence since it ensures that less important rows are discarded in favor of

for i = ... *for j = ...* *for k = ...* *for l = ...* *R[i+j-k, 2i+2j-2k, k-l] = ...*	*for u = ...* *for v = ...* *for w = ...* *for z = ...* *R[u, 2u, v]*	$\begin{pmatrix} 1 & 1 & -1 & 0 \\ 2 & 2 & -2 & 0 \\ 0 & 0 & 1 & -1 \end{pmatrix}$
(a) Original	(b) Transformed	(c) X

$\begin{pmatrix} 1 & 1 & -1 & 0 \\ 0 & 0 & 1 & -1 \end{pmatrix}$	$\begin{pmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$	$\begin{pmatrix} 1 & 1 & -1 & 0 \\ 0 & 0 & 1 & -1 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} i \\ j \\ k \\ l \end{pmatrix} = \begin{pmatrix} u \\ v \\ w \\ z \end{pmatrix}$
(d) B	(e) H	(f) Invertible mapping

Figure 4.3: A loop nest with a singular data access matrix

more important ones. For future reference, let us call the resulting matrix the *basis matrix* corresponding to the data access matrix.

Definition 4.4.1 The *basis matrix* of a data access matrix A is the first row basis of A.

The algorithm described informally above is simple, but it is expensive to keep checking rows for independence. A more efficient algorithm, which makes use of a variation of computing the Hermite normal form, is given in Figure 4.4. Given a data access matrix, Algorithm *BasisMatrix* returns a permutation matrix P, and the rank d of the data access matrix (the number of linearly independent rows). The first d rows of the permutation matrix P tell us which rows of the data access matrix are in the basis matrix. The following example should make this clear.

Consider the data access matrix X shown in Figure 4.3(c). This data access matrix can arise from the program shown in Figure 4.3(a). Algorithm *BasisMatrix(X)* returns the permutation matrix $P = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \end{pmatrix}$ and rank $d = 2$. The first two rows of the permutation matrix tell us which rows of A form a linearly independent basis: the position of the non-zero entry in these rows of P indicates which row of A is in the basis. In this example, the first and third rows form the basis matrix B in Figure 4.3. The significance of this in terms of transformations is that only the first and third subscripts can be normalized. This is reasonable because the subscript $2i + 2j - 2k$ is just a multiple of the subscript $i + j - k$.

Input: *An $m \times n$ data access matrix A.*
Output: *An $m \times m$ permutation matrix and the rank of A.*
Algorithm BasisMatrix(A : AccessMatrix) : (PermMatrix, Rank)
begin
  $P = I$, where I is the $m \times m$ identity matrix.
  done = false; i = 1;
  While not done do
    / Consider the submatrix A[i:m, i:n] */*
    Search for the first $j \geq i$ such that A[j, i:n] $\neq \vec{0}$;
    If no such j exists Then done = true;
    Else
      If $j \neq i$ then Exchange A[i, 1:n] and A[j, 1:n], P[i, 1:n] and P[j, 1:n]
      Apply the elementary column operations to make A[i, i] nonzero
        and A[i, i+1:n] zero.
      i = i + 1;
    End-If
  End-While
  return (P, i-1);
end

Figure 4.4: Computing a Basis Matrix

Input: *An $m \times n$ basis matrix B.*
Output: *An $(n-m) \times n$ padding matrix H.*

```
Algorithm Padding(B : BasisMatrix) : PadMatrix

begin
  H = I, where I is an n × n identity matrix.
  For i = 1, m do
    /* Consider the submatrix B[i:m, i:n] */
    apply the elementary column operations to make B[i, i] nonzero
      and B[i, i+1:n] zero.
    If columns i and j have been exchanged Then
      exchange rows i and j of H
    End-If
  End-For
  return (H[m+1:n, 1:n]);
end
```

Figure 4.5: Computing a Padding Matrix

4.4.2 Padding Matrix

To extend the basis matrix to a non-singular matrix, we need to add additional mutually independent rows which are also independent of the rows of the basis matrix. There are many possible choices for these rows, and we will use this flexibility when we take care of dependences in Section 4.5. Algorithm *Padding*, described in Figure 4.5, shows one simple way to pad the basis matrix. For an $m \times n$ full row rank matrix, we need to pad with an $(n-m) \times n$ matrix to form a non-singular matrix. It is well known that for a full row rank matrix, there exist m columns that are linearly independent. We simply need to pad these columns with $\vec{0}$ and the rest of the columns with columns from the $(n-m) \times (n-m)$ identity matrix I. For the program in Figure 4.3, since the first column and the third column are linearly independent, the padding matrix is H in Figure 4.3(e). The mapping between the old and new iteration spaces is in Figure 4.3(f). In the transformed program, shown in Figure 4.3(b), the reference becomes *R[u, 2u, v]*, and second index is not normalized.

The correctness of Algorithm *BasisMatrix* and Algorithm *Padding* follows from the following result.

Theorem 4.4.1 *Let A be an $m \times n$ integer matrix with rank d. There exists an $m \times m$ permutation matrix P and an $n \times n$ non-singular matrix Q such that $A = P\begin{pmatrix} B & 0 \\ D & 0 \end{pmatrix}Q$, where B is $d \times d$ lower triangular and nonsingular.*

Proof: Suppose at step k, by elementary column operations and permutation row operations, $P_k A Q_k = \begin{pmatrix} B_k & 0 \\ D_k & E_k \end{pmatrix}$ where B_k is $k \times k$ lower triangular.

- case 1: E_k is a zero matrix, then done.
- case 2: the first row of E_k is $\vec{0}$, then by a row permutation on E_k, the first row can be made non-zero, which falls into case 3.
- case 3: the first row of E_k is not $\vec{0}$. By a sequence of elementary column operations, The top-left position of the first row can be made positive and others zero.

After step d, where the matrix E_d is zero, $P_d A Q_d = \begin{pmatrix} B_d & 0 \\ D_d & 0 \end{pmatrix}$. Let $P = P_d^{-1}$ and $Q = Q_d^{-1}$, since both are non-singular matrices. Then $A = P\begin{pmatrix} B & 0 \\ D & 0 \end{pmatrix}Q$, where $B = B_d$ and $D = D_d$. $\square$

4.5 Data Dependences

The results of Section 4.4 showed that a basis matrix can always be padded to yield an integer, non-singular matrix. However, there is no guarantee that the transformation corresponding to this final matrix is legal, because this transformation may violate data dependences. To understand the problem, consider Figure 4.7 in which A is a basis matrix and D_A is the dependence matrix. Each column of the dependence matrix represents the *distance vector* of a dependence in the loop nest [Ban90]. In our example, there is just one dependence, and the distance values tell us that the dependence is between successive iterations of the innermost loop. A distance vector has the property that its leading non-zero is always positive; a legal transformation must preserve this property for each dependence, since the source of the dependence must be executed before its destination. If T is a non-singular matrix representing a loop transformation, it is easily shown that TD is the dependence matrix of the restructured loop nest; therefore, the leading non-zero element in each column of TD must be positive.

In Figure 4.7, A is the basis matrix; by looking at the product $AD_A = \begin{pmatrix} 0 \\ -1 \end{pmatrix}$, we can see at once that A cannot be padded to give us a transformation that respects data dependences. The intuition is that the first two rows of A determine the two outermost loops of the transformed loop nest. In the original program, the dependence was carried by the innermost loop, but in the new program, the dependence is 'carried' by the second loop. Unfortunately, the negative value of the second dimension of AD_A means that the source of the dependence will be executed after the sink. Clearly, there is nothing we can do in the *inner* loops that would remedy this situation, so it is impossible to pad A to yield a legal transformation.

To get around this problem, we proceed in two steps. We start with the basis matrix and use Algorithm *LegalBasis* to produce a new basis matrix that does not violate dependences. Then, we pad this matrix using the Completion Algorithm in Section 4.6 to yield the final transformation.

4.5.1 Generating a Legal Basis

Algorithm *LegalBasis* , shown in Figure 4.6, takes a basis matrix and checks each row against the dependences. For example, consider the product of the first row and D. This gives us a row vector in which entries can be positive, zero or negative. If an entry is positive, it means that the corresponding dependence will be carried by the new outermost loop. Therefore, the structure of the inner loops does not matter as far as this dependence is concerned, and we may delete it from the D matrix for the rest of the algorithm. If the entry is zero, then the dependence will not be carried by the potential outermost loop, so we leave the dependence in the D matrix. However, if we have a negative entry, the dependence is 'carried' by the potential outer loop, but the order of the iterations is wrong. Notice that if all of the entries of the row vector are 0 or negative (intuitively, for all dependences, the potential outer loop either does not carry the dependence or the source of the dependence is executed after the sink), we can simply reverse the direction of the loop. Problems arise only if some entries are positive and others negative — in that case, we cannot keep that row of the basis matrix, and we delete it from the basis matrix[1] For the example in Figure 4.7, *LegalBasis (A)* generates the basis A_1 shown in Figure 4.7.

4.5.2 Legal Padding Matrix

To pad a legal basis matrix, we need to satisfy two constraints. First, any row added must be linearly independent of other rows, so that the final matrix is non-singular. Second, the row must not violate dependence constraints. Once a new row has been added during padding, all dependences carried by the loop corresponding to this row may be dropped from consideration when filling in the rest of the matrix. When there are no further dependences to be satisfied,

[1] A better scheme, which is more complex to implement, is to maintain a list of rows that have been removed from the data access matrix, and reconsider these rows whenever dependencies are deleted from the dependence matrix.

Input: *An $m \times n$ basis Matrix B and a dependence matrix D.*
Output: *A legal basis Matrix.*

```
Algorithm LegalBasis (B : BasisMatrix, D : DepMatrix) : BasisMatrix
begin
  Let B_i be the ith row of B and d_i be the ith column of D.
  For i = 1, m
    f^T = B_i D
    If each element of f is non-negative then
      D = D - d_j, where f[j] > 0
    Elseif each element of f is non-positive then
      B_i = (-1) B_i;
      D = D - d_j, where f[j] < 0
    Else
      B = B - B_i;
    End-If
  End-For
  return B;
end
```

Figure 4.6: Algorithm LegalBasis: Computing a Legal Basis Matrix

$$\begin{pmatrix} -1 & 1 & 0 \\ 0 & 1 & -1 \end{pmatrix} \qquad \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix} \qquad \begin{pmatrix} -1 & 1 & 0 \\ 0 & -1 & 1 \end{pmatrix}$$

$$A \qquad D_A \qquad A_1$$

$$\begin{pmatrix} -1 & 1 & 0 \end{pmatrix} \qquad \begin{pmatrix} -1 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

$$B \qquad B_1$$

$$\begin{pmatrix} 0 & 0 \\ 1 & 0 \\ 0 & 1 \end{pmatrix} \qquad \begin{pmatrix} -1 & 1 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \end{pmatrix}$$

$$D \qquad T$$

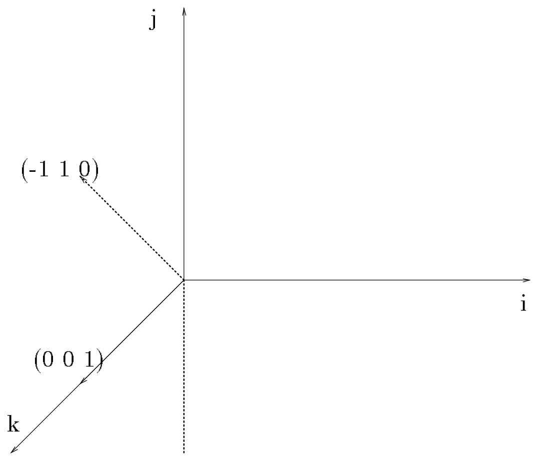


Figure 4.7: Legal basis and padding matrices

we can apply Algorithm *Padding* of Section 4.4.2 to complete the generation of a legal, non-singular matrix.

As an example, consider the basis matrix B which is legal with respect to the dependence matrix D in Figure 4.7. The first dependence is carried by the new outermost loop represented by the first row of B, and can be dropped from consideration for the rest of the procedure. The inner product of the first row with the second dependence is 0, meaning that this dependence is not carried by the new outermost loop; therefore, it must be taken into account when padding the matrix. To pad B, we need to find a row whose inner product with the second dependence vector is non-negative.

It is not immediately clear that such a vector exists; fortunately, Algorithm *LegalInvt* in Section 4.6 gives a positive answer by computing such a vector using a standard result about projections [Sch86]. This vector can be written as $x = c(I - P(P^TP)^{-1}P^T)e_k$ for some positive scaling integer c that makes all of the entries integers, where $e_i^T = [0, 0, .., 1, .., 0]$, with the 1 in the ith position, I is the identity matrix, and P is the partial transformation.

For our example, the remaining dependence to be satisfied is e_3. The new row vector for the padding is $x = e_3$. Since the dependence is carried by the loop corresponding to this new row vector, we can drop the dependence from consideration now. The dependence matrix is empty at this point. The new legal basis matrix is B_1 in Figure 4.7. Then we can use the Algorithm *Padding* to produce a linear, non-singular matrix. The final matrix T in Figure 4.7 is a linear, non-singular matrix and the corresponding transformation satisfies all of the dependences.

4.6 Completion Algorithm

One advantage of using integer non-singular matrices is that there is a simple *completion procedure* that takes the first few rows of a desired transformation matrix and generates a complete transformation matrix that respects dependences. As we have seen from the previous section, it is quite useful to have such completion algorithm. In the rest of the thesis, we will see that the completion algorithm makes the construction for parallelization and cache locality much easier.

4.6.1 Completion Procedure

Our completion procedure requires that the following precondition be satisfied.
Precondition: The partial transformation must have full row rank, and should not violate dependences.

These conditions are reasonable: if a row of the transformation matrix is linearly dependent on the others, it is clearly impossible to generate a non-singular matrix by adding additional rows. Similarly, if some row of the partial transformation violates one or more dependences, this cannot be rectified by extending the matrix with additional rows.

First, we delete all dependence vectors that are carried by the loops corresponding to the rows of the partial transformation, since they do not have to be considered when filling in the rest of the matrix. The completion procedure works by finding a vector that is independent of the existing row vectors in the partial transformation and within 90 degrees of each dependence vector. This vector is appended a new row to the partial transformation and all dependencies carried by the loop corresponding to this row are dropped from consideration. This technique is repeatedly applied until there are no further dependencies to be satisfied, at which point we can apply standard linear algebra techniques to complete the generation of a non-singular matrix.

To find the desired rows, we make use of the following invariant:

Invariant: The dependence vectors are in the orthogonal complement of the subspace spanned by the rows of the partial transformation.

We find a vector that is within 90 degrees of every dependence vector and strictly within 90 degrees with at least one dependence vector. This vector can be found by looking for the first row of the dependence matrix with nonzero entries. Let k be that row index.

Lemma 4.6.1 *e_k is within 90 degrees of every dependence vector, and strictly within 90 degrees of at least one dependence vector.*

Proof: It is easy to check that e_k satisfy the condition, since for any dependence vector d, $e_k^T d \geq 0$, and $e_k^T d > 0$ for at least one d. $\Box$

But e_k is not necessarily linearly independent of the rows in the partial transformation. Therefore, we project e_k to the orthogonal complement of the subspace spanned by the rows of the partial transformation so that the projected vector is linearly independent of the existing rows. If P^T is the partial transformation, it is easy to see that the projector is $Q = (I - P(P^T P)^{-1} P^T)$. The projected vector is $y = Qe_k$. Let $x = cy$ for some positive scaling number c that makes all of the entries integers and relative primes. For $P^T = (2\ \text{-}3)$ and $D = \begin{pmatrix} 3 \\ 2 \end{pmatrix}$ the projection is shown in Figure 4.8.

Theorem 4.6.1 *The projected vector y is linearly independent of the rows and within 90 degrees of every dependence vector.*

Proof:

- Linear independence:

  We can prove a even stronger result, i.e. y is orthogonal to the rows in P^T.

$$P^T y = P^T(I - P(P^T P)^{-1} P^T) e_k = 0$$

- Within 90 degrees:

  For any dependence vector d, $y^T d = e_k^T Q^T d$. Q is symmetric and d is already in the orthogonal complement by the invariant. This means $Q^T d = Qd = d$. Then $y^T d = e_k^T d$. Hence y is within 90 degrees of every dependence vector by Lemma 4.6.1.

□

The complete algorithm is presented in Figure 4.9.

Theorem 4.6.2 *The algorithm Completion generates a legal Λ-transformation.*

Proof: immediately follows from Theorem 4.6.1. □

We now apply the algorithm to the example in Figure 4.8. It can be satisfied by choosing $t_{21} = 3$ and $t_{22} = 2$. The new dependence vector is $\begin{pmatrix} 0 \\ 13 \end{pmatrix}$, which means that the new outer loop is a parallel loop. The matrix $\begin{pmatrix} 2 & -3 \\ 3 & 2 \end{pmatrix}$ satisfies the conditions.

4.6.2 Discussion

The completion technique discussed here works even when dependences are represented using direction vectors. There is considerable flexibility in the choice of the projector, and we have shown just one possibility; the choice of the most desirable projector will depend on the application.

4.7 Experiments on Multiprocessor without Caches: BBN GP1000

In this section, we demonstrate the power of our techniques using routines from the BLAS (Basic Linear Algebra Subprograms) library.

4.7.1 The Machine Architecture

The target machine is a BBN Butterfly GP-1000. On this machine, a processor can access its local memory in about 0.6 microsecond, but a non-local access takes about 6.6 microseconds even in the absence of contention in the network. For block transfers, the startup time is about 8 microseconds, and after that,

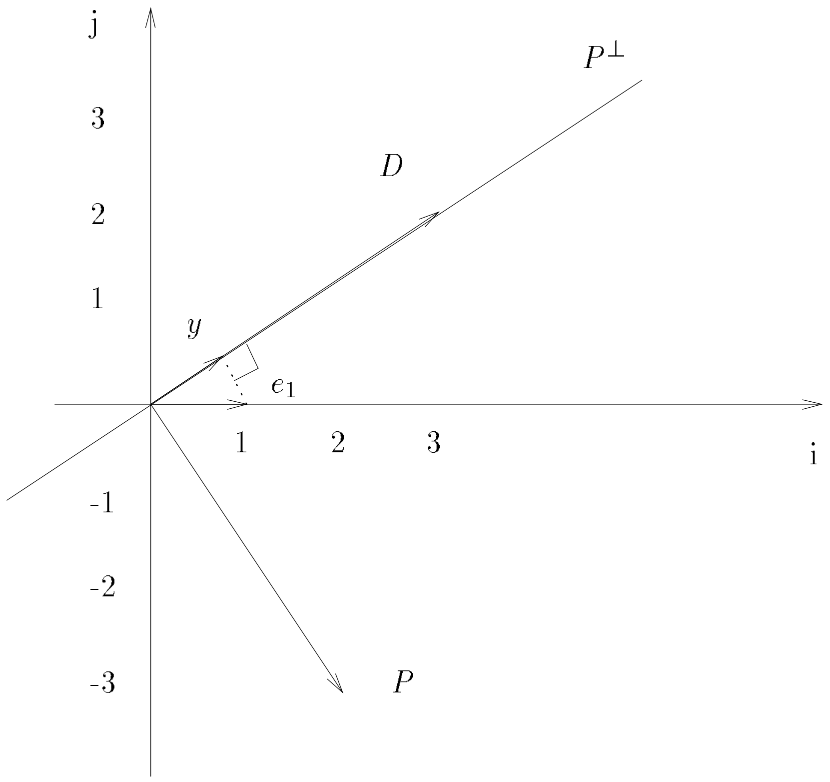


Figure 4.8: Extending Partial Transformation by Projection

Input: *An $m \times n$ partial transformation matrix P^T*
and a dependence matrix D.
Output: *An $n \times n$ legal transformation matrix T.*
Algorithm Completion(P^T, D) : Matrix
begin
/ Let P_i^T be row i of P^T, and d_i be column i of D */*
For i = 1, m
$f^T = P_i^T D$
$D = D$ - d_j, *where* $f[j] > 0$
End-For
$r = m + 1$;
While D is not empty do
let k be the first nonzero row of D;
$x = c(I - P(P^T P)^{-1} P^T) e_k$;
where c is a positive number that makes x
an integer vector and the entries relative primes.
$D = D$ - d, *if d is a dependence vector and* $d[k] > 0$
$P_r^T = x^T$;
$r = r + 1$;
End-While

Figure 4.9: Computing a Legal Full Transformation (first part)

$R = I$, *where I is an $n \times n$ identity matrix.*
For i = 1, r do
/ Consider the submatrix P^T[i:m, i:n] */*
apply the elementary column operations to make P^T[i, i] nonzero
and P^T[i, i+1:n] zero.
If columns i and j have been exchanged Then
exchange rows i and j of R
End-If
End-For
return(append(P^T, R[r+1:n, 1:n]));
end

Figure 4.10: Computing a Legal Full Transformation (second part)

a byte is transferred every 0.31 microseconds [BBN89]. Our compiler takes as input FORTRAN-77 programs with data distribution information, and it generates C code for each processor; this node program is compiled into native code using the Green Hills C compiler (Release 1.8.4). The C compiler performs only conventional code optimizations, so our experimental results are not skewed by any restructuring performed by this compiler. We will use pseudo-code in discussing examples.

For the GEMM code, our techniques are successful in eliminating non-local accesses significantly, so block transfers contribute just a small amount to overall performance. In the SYR2K code, the reduction of non-local accesses is less significant, so block transfers of non-local data are important for good performance. We develop a very simple performance model to explain these results.

4.7.2 GEMM

General matrix multiplication (GEMM) is one of the central subroutines in BLAS. The sequential code for this routine is shown in Figure 4.11(a). All arrays are of size 400 by 400 and are distributed in wrapped column manner. By distributing the outermost loop among the processors without doing any transformations, we obtain the graph labeled *gemm* in Figure 4.11(i).

The data access matrix and dependence matrix produced by our system are shown in Figure 4.11(e) and (f). The non-singular matrix for the transformation is shown in Figure 4.11(g). The transformed loop nest of Figure 4.11(b) yields the code of Figure 4.11(c), and the corresponding execution times and speed-ups are labeled *gemmT* in Figure 4.11(h,i). Inserting block transfers for accesses to A, we get the code shown in Figure 4.11(d), and the performance of this program is labeled *gemmB* in Figure 4.11(h,i).

After access normalization, accesses to C and B are local, but there are non-local accesses to A. Since three out of four data structure accesses in each iteration have become local, the effect of block transfers is relatively small. To understand the behavior of this program, a simple performance model can be used. Since the outer loop of GEMM does not carry any dependences, the time to execute GEMM can be expressed as follows:

$$T_P(\alpha) = \frac{n_f t_f}{P} + \frac{n_m(\alpha t_l + (1-\alpha)t_r(P))}{P} + o(P)$$

where

- P: number of processors.
- α: the proportion of local memory accesses.
- $T_P(\alpha)$: time to execute program — a function of P and α

```
for i = 1, N
  for j = 1, N
    for k = 1, N
      C[i, j] = C[i, j]
          + A[i, k] * B[k, j]
```

(a) original code

```
for u = 1, N
  for v = 1, N
    for w = 1, N
      C[w, u] = C[w, u]
          + A[w, v] * B[v, u]
```

(b) transformed code

```
for u = p, N, step P
  for v = 1, N
    for w = 1, N
      C[w, u] = C[w, u]
          + A[w, v] * B[v, u]
```

(c) parallel code for node p

```
for u = p, N, step P
  for v = 1, N
    read A[*, v];
    for w = 1, N
      C[w, u] = C[w, u]
          + A[w, v] * B[v, u]
```

(d) parallel code with block transfer

Figure 4.11: GEMM

$$X = \begin{pmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

(e) X

$$D = \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix}$$

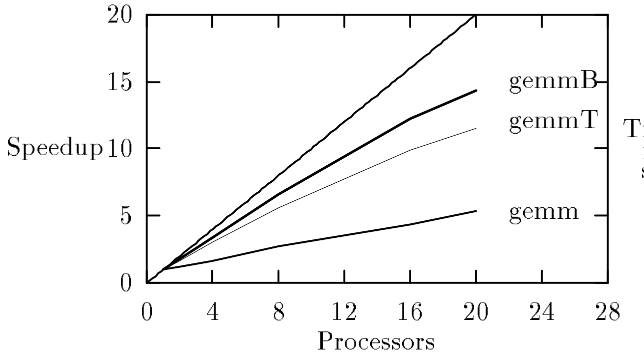
(f) D

$$T = \begin{pmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \end{pmatrix}$$

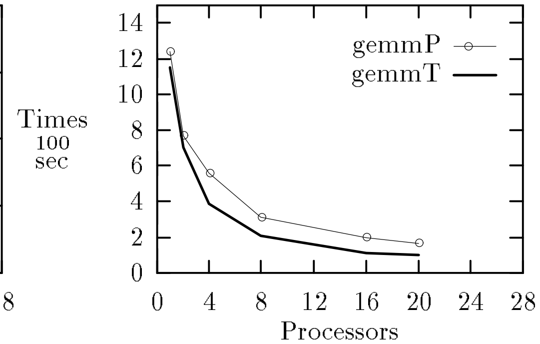
(g) T

Nodes/Prog	gemm	gemmT	gemmB
1	1,245.2	1,150.0	1,117.8
2	1,072.5	704.3	630.6
4	784.4	387.5	328.1
8	462.5	206.2	169.3
16	283.1	116.8	92.5
20	230.6	100.0	78.7

(h) Execution Times (sec)



(i) Speedup



(j) Projected times

Figure 4.12: GEMM (cont.)

- n_f: number of arithmetic operations.
- t_f: effective time to execute an arithmetic operation.
- n_m: number of memory accesses.
- t_l: time for a local access.
- $t_r(P)$: time for a remote access. For a given program, it is an increasing function of P because of network contention.
- $o(P)$: overhead to spawn processes etc.

Access normalization increases the proportion of local accesses. Let α_0 and α be the proportions of local accesses before and after access normalization. The reduction in execution time due to access normalization is given by this equation:

$$T_{nb} = T_P(\alpha_0) - T_P(\alpha) = (\alpha - \alpha_0)(t_r(P) - t_l)\frac{n_m}{P}$$

GEMM has $4N^3$ memory accesses. The proportion of local accesses in the original version is $\alpha_0 = \frac{1}{P}$, and in the transformed version, it is $\alpha = \frac{1}{4}(3 + \frac{1}{P})$. Substituting these values, and assuming a non-local access time of 6 microseconds, we would expect that the time to execute the program is

$$T_P(\alpha_0) - T_{nb} = T_P(\alpha_0) - \frac{1200}{P}(1 - \frac{1}{P})$$

Since access normalization reduces network contention, and we have ignored network contention in these calculations, the actual execution times will be less than the prediction. The projected and realized times shown in Figure 4.11(j) bear these calculations out.

4.7.3 SYR2K

When remote accesses are necessary due to the problem structure, it is beneficial to use block data transfers to amortize the cost of the startup time. Consider the rank *2k* update SYR2K from BLAS (Basic Linear Algebra Subroutines) [CVL88]. The subroutine computes $C = \alpha A^T B + \alpha B^T A + C$. Suppose A and B are banded matrices with band width b, then C is symmetric and banded with band width $2b - 1$. The banded matrices A, B are stored in $n \times 2b - 1$ arrays A_b, B_b such that the elements $A[i, j], B[i, j]$ are in $A_b[i, j - i + b - 1]$ and $B_b[i, j - i + b - 1]$. C is symmetric so only the upper triangular matrix is stored in an $n \times (2b - 1)$ array C_b such that $C[i, j]$ is in $C_b[i, j - i]$. The program is shown in Figure 4.13(a).

```
for i = 1, N
  for j = i, min(i+2b-2, N)
    for k = max(i-b+1, j-b+1, 1),
          min(i+b-1, j+b-1, N)
      Cb[i,j-i+1] = Cb[i,j-i+1]
        + αAb[k,i-k+b]*Bb[k,j-k+b]
        + αAb[k,j-k+b]*Bb[k,i-k+b]
```

(a) original code

```
for u = 1, 2b-1
  for v = 1-b, b-u
    for w = max(1, u+v), min(N, N+v)
      Cb[-u-v+w+1, u] = Cb[-u-v+w+1, u]
        + αAb[w, -u-v+b]*Bb[w, -v+b]
        + αAb[w, -v+b]*Bb[w, -u-v+b]
```

(b) transformed code

```
for u = p, 2b-1, step P
  for v = 1-b, b-u
    for w = max(1, u+v), min(N, N+v)
      Cb[-u-v+w+1, u] = Cb[-u-v+w, u]
        + αAb[w, -u-v+b]*Bb[w, -v+b]
        + αAb[w, -v+b]*Bb[w, -u-v+b]
```

(c) parallel code for node p

```
for u = p, 2b-2, step P
  for v = 1-b, b-u
    read Ab[*,-u-v+b]; read Ab[*,-v+b];
    read Bb[*,-v+b]; read Bb[*,-u-v+b];
    for w = max(1, u+v), min(N, N+v)
      Cb[-u-v+w+1, u] = Cb[-u-v+w+1, u]
        + αAb[w, -u-v+b]*Bb[w, -v+b]
        + αAb[w, -v+b]*Bb[w, -u-v+b]
```

(d) parallel code with block transfer

Figure 4.13: SYR2K

$$\begin{pmatrix} -1 & 1 & 0 \\ 0 & 1 & -1 \\ 0 & 0 & 1 \\ 1 & 0 & -1 \\ 1 & 0 & 0 \end{pmatrix}$$

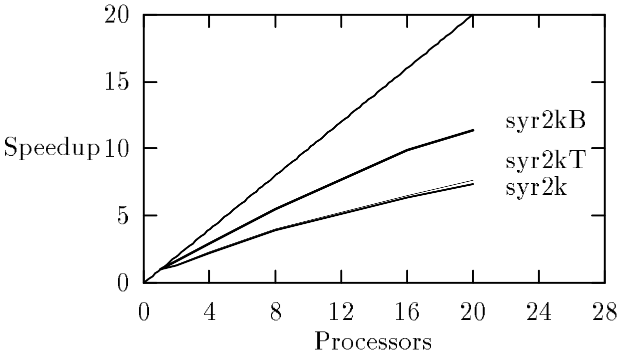
(e) X

$$\begin{pmatrix} -1 & 1 & 0 \\ 0 & -1 & 1 \\ 0 & 0 & 1 \end{pmatrix}$$

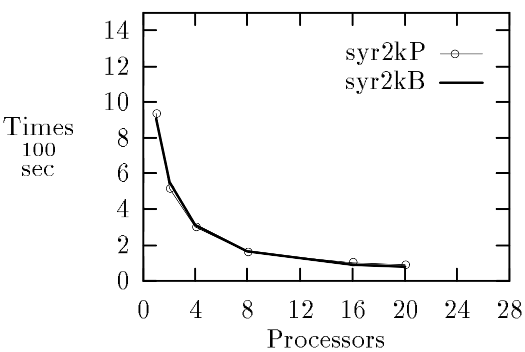
(f) T

Nodes/Prog	syr2k	syr2kT	syr2kB
1	1,289.3	933.5	904.3
2	961.8	675.6	554.3
4	581.8	415.8	310.0
8	334.3	241.2	166.8
16	200.0	143.1	91.8
20	173.8	121.8	79.3

(g) Times (sec) for SYR2K



(h) Speedup



(i) Projected times

Figure 4.14: SYR2K (cont.)

Assume that we are given a wrapped-column mapping for each array. The data access matrix is matrix X shown in Figure 4.13(e). If we apply Algorithm *BasisMatrix*, we get a base matrix B consisting of the first three rows of X. However, the dependence matrix is $[0,0,1]^T$. The legal base mapping is B_{legal}, which is B with the second row negated, as is shown in Figure 4.13(f). This matrix is non-singular. Using B_{legal} as the transformation matrix gives the transformed code in Figure 4.13(b). Distributing the outermost loop iterations and generating block transfers, we get the parallel code in Figure 4.13(d) for processor p. Figure 4.13(g) shows the experimental results. Block transfers are relatively important in this example, since there are many non-local accesses left in the transformed code.

To understand the experimental results, let us include block transfers into our model. When block transfers are done, a block of remote data is transferred to a piece of local memory, and the accesses to the remote location are replaced by accesses to the local copy. Let β to be the proportion of remote data that are transferred using block transfer and t_b be the amortized access time for remote references. The formula for the execution time of the program is refined as follows:

$$T_P(\alpha,\beta) = \frac{n_f t_f}{P} + \frac{n_m}{P}(\alpha t_l + \beta(1-\alpha)t_l + \beta(1-\alpha)t_b(P) + (1-\beta)(1-\alpha)t_r(P)) + o(P)$$

The amortized remote access time can be estimated as follows. Let the cost function of a block transfer be $a + bB$, where a is the startup time, b is the time between successive byte transfers, and B is the block size. Let n_b be the number of block transfers and $s_1, s_2, \ldots s_{n_b}$ be the sizes of blocks. The amortized access time for data transferred through block transfers can be computed using the following formula:

$$t_b = \frac{sizeof(float)\sum_{i=1}^{n_b}(a + b * s_i)}{\sum_{i=1}^{n_b} s_i} + t_l$$

SYR2K has $12Nb^2$ number of memory references, in which $\alpha = \frac{1}{3}(1+\frac{2}{P})$ of them are local after transformation. The rest of the remote references are performed using block transfers. The number of block transfers of size N is $8b^2(1-\frac{1}{P})$. The problem size is 500 with a band size of 200. Thus the amortized remote access time is $t_b \simeq 2\mu s$ per floating point. Therefore the time saved due to block transfers is T_b:

$$T_b = T_P(\alpha,0) - T_P(\alpha,1) = \frac{640}{P}(1-\frac{1}{P})$$

The projected times and actual running times are shown in Figure 4.13(i). They are very close since contention plays less of a role in this program.

4.8 Discussion and Related Work

This chapter is a contribution to the state of the art of compiling programs in languages like FORTRAN-D that permit user-defined data decomposition for parallel machines with a memory hierarchy, which is the goal of a number of projects including Parascope, Superb, Id Nouveau, Crystal, Kali, PARTI and ASPAR projects [CK88,HKT91,ZC90,RP89,Tse89,LC91,KM91,MSS+88, IFKF90]. The emphasis in these projects has been on code generation mechanisms (such as the ownership rule discussed in Section 4.2) and on recognizing and exploiting special patterns of computation and communication such as reductions. Although it is well-known that loop restructuring before code generation can improve performance, no general loop restructuring mechanism has been available until now.

We require the programmer to specify data distributions. Automatic deduction of this information for special programs has been investigated by Balasundaram and others [BFKK90], by Gannon *et al* [GJG88] on CEDAR-like architectures, by Hudak and Abramham [HA90] for sequentially iterated parallel loops, by Knobe *et al* [KLS90] for SIMD machines, by Li and Chen [LC89] for *index domain alignment* and by Ramanujam and Sadayappan[RS91] who find communication-free partitioning of arrays in fully parallel loops. These efforts focus on deducing good data distributions for particular kinds of programs such as fully parallel loops, and no general solution to this problem is known.

We have opted to generate code by distributing outer loop iterations among the processor. However, access normalization can also be integrated with the ownership rule. Rather than construct the data access matrix by choosing the dominant subscripts from *all* references as described in Section 4.2, we can choose the dominant subscripts in the distribution dimension(s) of the array references on the *left* hand side of the assignments only, since the processor that owns the array element being defined executes the assignment statement. Loop nests with multiple assignment statements present other opportunities for generating better code. In this paper, loop restructuring works on entire loop nests and every statement in the loop body undergoes the same transformation. A possible extension is to have different transformations for different statements. For example, statement S_1 has transformation T_1 and S_2 has T_2. Let the new iteration spaces for S_1 and S_2 be J_1 and J_2 respectively. The new loop nest is an union of J_1 and J_2 with appropriate guards (conditionals) for S_1 and S_2 to insure that they only get executed in their subspace. A special case is when the data dependences allow J_1 and J_2 to be two separate iteration spaces; in that case, we can construct different loop nests for S_1 and S_2 respectively — this is the same as *loop distribution*[Wol89].

Chapter 5

Transformations for Cache Locality

5.1 Introduction

Almost every modern processor is designed with a memory hierarchy organized into several levels, each of which is smaller, faster, and more expensive than the level below.

The cache is the memory level between CPU and the main memory. It is usually between 1KB to 256KB, and is divided into *blocks (lines)* of 4-128 bytes. The unit of data transfer between main memory and cache is a block. Typically, a cache hit takes only one cycle, while a cache miss takes 8-32 cycles [HP90]. It is ideal for all memory references to hit in the cache, however, since cache size is much smaller than main memory, once new data needs to be brought in from main memory, some data in the cache has to be replaced. There will be a cache miss when the replaced data is accessed in the future.

Therefore, high performance requires programs to possess *cache locality*, i.e. having data reuse of the data in the cache before it has been replaced. If the same data in the cache is reused, this is called *temporal locality*. Since the unit of data in cache is a block (line), once the block is brought into the cache, any access to the data elements in the block will be a cache hit. This is called *spatial locality*.

A program may have data reuse, but might not be able to exploit cache locality due to the replacement of the data in the cache. Program transformations can improve the performance significantly. The idea is to change the data access sequence so that a data reuse can be translated into a cache hit than a cache miss. One important approach is to use *loop tiling*, invented by Wolfe [Wol89], to block the innermost loop nest so that data reuse within these smaller tiles can be exploited by the cache. Loop tiling can be improved by loop transformations such as loop interchange to bring some outer loops to the innermost position, and then apply loop tiling on the new innermost loops. Wolf and Lam have created a cache locality model, and applied unimodular transformations to choose the best innermost loop nest to tile [WL91a]. Exploiting cache locality has been an active research area. We will discuss more related work in Section 5.6.

Program transformations, such as tiling, have been proven to be very useful in improving cache locality. However, they have been usually applied in an ad hoc way. Wolf and Lam have proposed a cache model based on *reuse vector space*, where tiling can be applied to exploit cache locality. Unimodular transformations are employed to choose an innermost loop nest. They also have showed that it is difficult to choose a tile size for real machines, which usually have direct mapped caches or low associativity caches [LRW91]. The performance can be very erratic, if the block size is not chosen carefully.

In this chapter, we make the following contributions:

- We introduce a new simple cache model based on *reuse distances* that is more precise than the *reuse vector space* model.
- We develop a new loop transformation technique called *height reduction*, which optimizes directly on reuse distances, so that no exhaustive search is necessary.
- We also integrate height reduction with tiling. This algorithm is called *width reduction*, which combines both height reduction and tiling, and the benefit of tiling is retained. Height reduction helps reduce cache interferences when tiling for real machines.

The implementation of the algorithms is also quite simple when used with the *Lambda* loop transformation toolkit [LP93b]. We have observed speedups of 1.7 to 10 performance improvement on programs from linear algebra and NASA benchmark suite.

The rest of the chapter is organized as follows: in Section 5.2, we define a simple cache reuse model, and give a formulation of the problem. The height reduction algorithm is described in Sections 5.3, and the width reduction algorithm is presented in Section 5.4. Empirical evaluations were conducted on an HP 9000/720 workstation, and the results are reported in Section 5.5. Finally, we discuss related work in Section 5.6.

5.2 A Simple Cache Reuse Model

A program must have data reuse in order to exploit cache locality. For example, consider the program in Figure 5.1. The data element *A[i, j]* is used in both iterations (i, j) and $(i+1, j-1)$.

for i = 1, n
for j = 1, n
= A[i, j] + A[i-1, j+1]
end for
end for

Figure 5.1: A Simple Example

5.2.1 Reuse Vectors

We can define data reuse concisely using an integer vector that represents two loop iterations that access the same data location. This vector is called *reuse vector*. Consider a loop nest of n loops. An iteration of the loop nest can be represented by an integer vector of dimension n.

Definition 5.2.1 Let $\vec{j}_1$ and $\vec{j}_2$ be two iterations that access the same data location. $\vec{r} = \vec{j}_2 - \vec{j}_1$ is called a *reuse vector*.

A reuse vector is an n-dimensional vector in a loop nest of depth n. The reuse vector for the example in Figure 5.1 is $\begin{pmatrix} 1 \\ -1 \end{pmatrix}$.

An important attribute of a reuse vector is the *height*.

Definition 5.2.2 The *height* of a reuse vector is the number of dimensions from the first non-zero entry to the last entry.

For example, the height of the reuse vector $\vec{r} = (0, 2, 0, -1)^T$ is 3, since the first non-zero is at the second dimension.

Another important term is the notion of a reuse vector *carried* by some loop.

Definition 5.2.3 If the ith dimension has the leading non-zero in the reuse vector r, then r is said to be *carried* by the loop i.

For example, the reuse vector above is carried by the second loop. If a reuse vector is carried by the loop i, then the height of the reuse vector is $n - i + 1$ [1].

The data reuse vector can be computed by solving a system of linear equations [GJG88,WL91a]. There is a reuse between $A[i, j]$ and $A[i-1, j+1]$, if for two iterations (i_1, j_1) and (i_2, j_2), $A[i_1, j_1]$ and $A[i_2 - 1, j_2 + 1]$ reference the same data. Therefore, the system of equations is as follows:

$$i_1 = i_2 - 1$$

[1] The outermost loop is loop 1.

$$j_1 = j_2 + 1$$

Let $r = \begin{pmatrix} i_2 \\ j_2 \end{pmatrix} - \begin{pmatrix} i_1 \\ j_1 \end{pmatrix}$ be the reuse vector. The solution is $r = \begin{pmatrix} 1 \\ -1 \end{pmatrix}$.
We can rewrite the equations using matrix notation.

$$A_1 K_1 = A_2 K_2 + c_2$$

where $A_1 = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$, $A_2 = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$, $c_2 = \begin{pmatrix} -1 \\ 1 \end{pmatrix}$, $K_1 = \begin{pmatrix} i_1 \\ j_1 \end{pmatrix}$, and $K_2 = \begin{pmatrix} i_2 \\ j_2 \end{pmatrix}$.

The first row of A_1 is the coefficients of the subscript $1 * i + 0 * j$ in the first dimension of $A[i, j]$, the second row of A_1 is from the second subscript. A_2 is computed in a similar way as A_1. c_2 is the constant vector, where the first element is from the constant in the first dimension of $A[i + 1, j - 1]$, and the second element is from the second dimension.

In general, for an m-dimensional array R in a loop nest of depth n, $R[a_{11}i_1 + ... + a_{1n}i_n + a_{10}, ..., a_{m1}i_1 + ... + a_{mn}i_n + a_{m0}]$ can be written as $R[A\vec{i} + a]$, where $A = (a_{ij})$. There is temporal reuse between references $A\vec{i} + a_1$ and $A\vec{i} + a_2$ iff there exist iterations $\vec{i}_1$ and $\vec{i}_2$ such that

$$A\vec{i}_1 + a_1 = A\vec{i}_2 + a_2.$$

Then, the reuse vector $\vec{r} = \vec{i}_2 - \vec{i}_1$ is the solution to the following equation.

$$A\vec{r} = a_1 - a_2 \tag{5.1}$$

Any reuse vector is a linear combination of a basis of the null space of A plus a special solution to the system.

$$\vec{r} = a_1\vec{r}_1 + ... + a_k\vec{r}_k + \vec{r}_0 \tag{5.2}$$

The set $\{\vec{r}_1, ..., \vec{r}_k\}$ is a basis of the null space of A. $\vec{r}_0$ is special solution to the system 5.1.

For spatial locality, data reuse depends on the memory layout of arrays. In FORTRAN, where arrays are column major, reuse vectors are computed by solving the following system.

for i = 1, n *for j = 1, n* *= A[j]* *end for* *end for* (a)	*for i = 1, n* *for j = 1, 5* *= A[j]* *end for* *end for* (b)

Figure 5.2: Reuse Distances

$$\begin{aligned}\overline{A}\vec{d} &= \overline{b} \\ |r_1^A\vec{d} - b_1| &< s\end{aligned}$$

where $\overline{A}$ is A with first row deleted, $\overline{b}$ is b with the first element deleted, r_1^A is the first row of A, b_1 is the first element of b, and s is the size of a cache line.

5.2.2 Reuse Distance

Reuse vectors are simple, but they are not sufficient for predicting cache locality. Consider the examples in Figure 5.2. Both have the same reuse vector $(1,0)^T$. If n is much larger than the cache size, program (a) will have less chance to reuse the data in the cache. On the other hand, program (b) will almost certainly have cache reuse.

Therefore, the trip count of loops should be included in our model. Formally, we define the *width* of a loop.

Definition 5.2.4 Let *width* of loop k be the trip count of loop k.

Definition 5.2.5 The *reuse distance* is defined to be the number of loop iterations between two reuse iterations.

Given r the reuse vector, the reuse distance d can be computed easily.

$$d = r_1S_1 + ... + r_nS_n$$

where $S_i = \prod_{k=i+1}^{n} w_k$, and w_k is the width of loop k.

For example, the reuse distance for program (a) in Figure 5.2 is n, and for program (b) the distance is 5.

There is a useful fact we will use for optimizing the reuse distance.

Lemma 5.2.1 *If every loop has more than one iteration, then* $\forall_i S_i > S_{(i+1)}$.

To increase the probability of cache locality, we need to shorten the reuse distance. From the definition of reuse distance and Lemma 5.2.1, we see there are two ways of achieving that. First, we can introduce more leading zeros in the reuse vector, since $\{ S_1, ..., S_n \}$ is a strictly decreasing sequence. Second, we can reduce the width of a loop. The first optimization should be more powerful, since S_i is usually much larger than $S_{(i+1)}$.

Then the optimization problem for cache locality can be decomposed into two subproblems.

- Height Reduction: lowering the loop that carries the reuse.
- Width Reduction: decreasing loop trip counts.

5.2.3 Data Reuse Matrix

To make the optimization on reuse vectors more convenient, we introduce another important data structure called the *data reuse matrix*, similar to the data access matrix. The set of reuse vectors is a linear combination of a basis for the null space of A plus a special solution as shown in equation 5.2. A data reuse matrix is a matrix that has the basis vectors and the special solution as its columns. Note that all reuses, *self, group, temporal* and *spatial*, defined in [WL91a] are included in the reuse matrix. In other words, the reuse matrix for the reuse computed from equation 5.2 is as follows:

$$R = (\vec{r}_1, ..., \vec{r}_k, \vec{r}_0)$$

In general, there are many individual reuse matrice (for example, every pair of array references that have reuse generate a reuse matrix.). The *global* reuse matrix is the union of all these individual reuse matrices. It is important to have priority on the reuse vectors. A simple heuristic is to construct the global reuse matrix as follows. Let u be the number of individual reuse matrices. We choose the highest priority for the vector space intersection of all individual reuse matrices (each matrix represents a vector space). We compute a basis for that intersection, and make the set of basis vectors as the first columns of the global reuse matrix. The second step is to compute a basis for the intersection of any $u-1$ individual reuse matrices, and add the basis vectors as the columns to the right of the global reuse matrix. This set of vectors has the second priority. Then we compute the intersections of any $u - 2$ reuses and so on. The basis vectors from individual reuses have the lowest priority. Therefore, the matrix is ordered from the left to the right with the decreasing priority.

In the rest of the chapter, we show loop transformations can be used to achieve height reduction and width reduction. Since our loop transformations are linear, the transformed reuse vector can be easily computed.

Lemma 5.2.2 *If T is a linear transformation and $\vec{r}$ is a reuse vector, then $T\vec{r}$ is the new reuse vector.*

Theorem 5.2.1 *If T is a non-singular transformation and R is a reuse matrix, then the new reuse matrix is TR.*

Proof:

Consider any reuse vector r.

$$\vec{r} = a_1\vec{r}_1 + ... + a_k\vec{r}_k + \vec{r}_0$$

Then

$$T\vec{r} = a_1T\vec{r}_1 + ... + a_kT\vec{r}_k + T\vec{r}_0.$$

Since T is non-singular, $T\vec{r}_1, ..., T\vec{r}_k$ are linearly independent of each other.
□

5.2.4 Discussion

To see the difference with the *reuse vector space* model, we consider the example in Figure 5.3. In version (a), the reuse vector for the reference $A[i, j]$ is $(1, 0)^T$ (spatial reuse), the reuse vectors for $A[i, k]$ are $(0, 1)^T$ (temporal reuse) and $(1, 0)^T$ (spatial reuse), and the reuse vectors for $A[j, k]$ are $(1, 0)^T$ (temporal reuse) and $(0, 1)^T$ (spatial reuse). The Wolf/Lam model would conclude that the reuse vector space is the whole space ij, since both i-loop and j-loop carry reuse. Similarly, the reuse vector space for version (b) is also the whole space ij. Therefore, the Wolf/Lam model does not distinguish between these two versions. In summary, the cost function computed using the Wolf/Lam model is a function of the *dimension* of the reuse vector space, i.e. the *number* of loops that carry reuse. Therefore, the order of loops is irrelevant as long as the loops carry reuse. If data dependences prevent a subset to become the innermost loop nest, unimodular transformations are applied so that the innermost loop nest may be tilable.

As our experimental results in Section 5.6 show, version (b) performs much better than version (a). In our *reuse distance* model, the reuse vector $(1, 0)^T$ is regarded as the important vector to be optimized, since it occurs much more frequently than the other one, and is the basis vector for the intersection of all reuses. Version (b) will be the optimized version generated by the algorithms presented in the rest of the chapter.

We describe the height reduction algorithm in Section 5.3, and the width reduction algorithm in Section 5.4.

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

(a)

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

(b)

Figure 5.3: Comparison of Models

5.3 Height Reduction

Given a data reuse matrix, the goal of height reduction is to find the minimum s such that rows 1 to $n-s$ of the data reuse matrix are all zeros. When we construct a legal transformation, we need to make sure that the transformation is non-singular, and does not violate data dependences.

Let P^T be the transformation matrix being constructed, D be the dependence matrix (every column is a dependence vector), and R be the reuse matrix (every column is a reuse vector).

5.3.1 Non-singularity

First, we satisfy the condition of non-singularity. To construct the first row of the transformation so that the first row of the new reuse matrix is 0, from Theorem 5.2.1, we need to find a vector in the null space of the old reuse matrix. Suppose we have formed a matrix P^T of s rows that satisfies the condition. To extend the matrix, we must find a vector v such that the following three conditions are satisfied:

1. v is linearly independent of P^T, and
2. v is in the null space of R.
3. v does not violate D, the data dependence matrix.

The first condition guarantees that the resulting transformation matrix is non-singular, and the second condition guarantees that data reuse is not carried by loop $s+1$. For example, consider the reuse matrix $R = \begin{pmatrix} 1 \\ -1 \\ 0 \end{pmatrix}$. To compute the first row of the transformation matrix, we can use the vector $(0,0,1)^T$, which is an element of the null space of R. Now the partial transformation P^T is $(0, 0, 1)$. To find the second row, we need to find a vector v that satisfies both the above conditions. In this case, $(1, 1, 0)$ would be a solution.

These three conditions can be satisfied using more efficient algorithms. For example, instead of finding a linear independent vector v, we can find a vector from the null space of P^T. It is easy to compute a vector from the null space of both P^T and R. Let Q be the column union of P^T and R. If Q does not span the whole vector space, any vector in the null space is a linear combination of a set of basis vectors: $v = a_1 x_1 + ... + a_t x_t$.

If Q spans the whole space, then some reuse vectors must be deleted before any row can be added to P^T. The vectors deleted are the ones with the lowest priority. Recall that the reuse matrix is ordered from the left to the right, with the highest priority column as the first column, and the lowest priority column as the last column.

Now we show that finding a vector from the null space of P^T is *not* less general than finding a vector that is linearly independent of P^T.

Theorem 5.3.1 *If there exists a linearly independent vector v_1 of P^T that is in the null space of R and does not violate D, then there is an orthogonal vector v of P^T that is in the null space of R and does not violate D.*

Proof: Let v be the resulting vector of v_1 projected to the orthogonal space of P^T by the projector J (e.g. $J = I - P(P^T P)^{-1} P^T$). Since $v_1^T R = 0$, $v^T R = (Jv_1)^T R = v_1^T J^T R$. Notice that J is a symetric matrix, and R is already in the orthogonal space of P^T. Therefore, $J^T R = JR = R$. Hence $v^T R = v_1^T R = 0$.

Similarly, $v^T D = v_1^T J^T D = v_1^T D \geq 0$, if $v_1^T D \geq 0$. $\square$

5.3.2 Dependences

Now, we consider data dependences. Any row added must not violate data dependences. Any dependence carried by the existing rows can be deleted from the dependence matrix. Since $v = a_1 x_1 + ... + a_t x_t$, we need to choose a set of values for $\{a_1, ..., a_t\}$ such that $v^T D \geq 0$. We first consider the case where there is only one data dependence vector. Let d be the dependence vector, then

$$v^T d = a_1 x_1^T d + ... + a_t x_t^T d.$$

To decide the values of each a_i, we use the following rules. Let *dir* be $x_i^T d$ computed using the dependence algebra defined in Section 3.5.1.

- if dir = "<" then $a_i = 1$

- if dir = "$>$" then $a_i = -1$
- if dir = "$=$" then $a_i =$ any value
- if dir = "$<=$" then $a_i = 1$
- if dir = "$>=$" then $a_i = -1$
- if dir = "$<>$" then $a_i = 0$
- if dir = "$*$" then $a_i = 0$

In the general case where there are more than one dependence vector, the value of a_i must be consistent with every dependence vector. If this fails, some reuse vectors must be deleted. For the special cases where the dependence vector is a distance vector, these rules can be further optimized.

5.3.3 Algorithm

The algorithm is shown in Figure 5.4. Let P^T be the transformation matrix being constructed, D be the dependence matrix (every column is a dependence vector), and R be the reuse matrix (every column is a reuse vector). The following invariants are maintained during the construction: $P^T D = 0$ and $P^T R = 0$.

We start from an empty transformation matrix. The notation P^T denotes a row matrix. A new matrix Q^T is created by simply combining the rows of P^T and R^T. If Q^T spans the whole space, then its null space is empty. Some lower priority reuse vectors have to be deleted until Q^T does not span the whole space. Once a solution, which consists of a set of all valid vectors, for the null space of Q^T is found, we need to choose a vector that satisfies the data dependences as discussed in the previous section.

For example, we consider the loop nest in Figure 5.3(a) again. The reuse matrix is $R = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$. The first column $(1,0)^T$ is the intersection of all three reuse vecter sets computed from references A[i,j], A[i,k] and A[j,k] as shown in Section 5.2.4. Therefore the second column is dropped while computing the first row of the transformation. The partial transformation is $P^T = (0,1)$. The completion algorithm is then called to complete the partial transformation to get the full transformation $T = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}$. The transformed version is in Figure 5.3(b).

Input: *A reuse matrix R, a dependence matrix D and dimension n.*
Output: *A legal transformation matrix T that maximizes cache reuse.*

```
Algorithm HeightReduction(R, D, n) : Matrix
begin
  P^T = ∅; s = 0;
  While ( (s < n) and (R not empty) do
    Q^T = union(P^T, R^T);
    While ( rank(Q) = n ) do
      delete the lowest priority reuse vector from R
    solve Q^T v = 0, let v = a_1 x_1 + ...a_t x_t
    choose a_1, ..., a_t such that v^T D = a_1 x_1^T D + .. + a_t x_t^T D ≥ 0.
    if success then
      D = D - d, if d is a dependence vector and v^T d > 0
      P_s^T = v^T; s = s + 1;
    else
      delete the reuse vector with the lowest priority from R
    end if
  End-While
  return(P^T);
end
```

Figure 5.4: Height Reduction

```
for i = 1, n
  for j = 1, n
    A[i, j] = A[i-1, j-1]
```

(a) Source Code

```
for t_i = 0, nb-1
  for t_i = 0, nb-1
    for i = t_i * b + 1, min(n, (t_i + 1) * b - 1)
      for j = t_i*b + 1, min(n, (t_i + 1) * b - 1)
        A[i, j] = A[i-1, j-1]
```

(b) Tiled

Figure 5.5: Loop Tiling

5.4 Width Reduction

Loop width can be reduced using *loop tiling* [Wol89]. For example, the reuse vectors for the loop nest in Figure 5.5 are of the form of $a * \begin{pmatrix} 1 \\ 0 \end{pmatrix} + \begin{pmatrix} 0 \\ 1 \end{pmatrix}$. The reuse matrix spans the whole space. The loop nest in (b) is the tiled version of the loop nest in (a), where b is the block size, and nb is the number of blocks computed from b and n. Note that extra "tile" loops have been introduced. This adds some overhead to the execution time.

Loop tiling is not always legal. If we replace the statement in Figure 5.5(a) by *A[i,j] = A[i-1,j+1]*, then the dependence vector is $(1, -1)^T$, which prevents tiling. The reader can easily check that tiling would violate data dependences However, loop transformations may be applied to make a loop nest tilable. A loop reversal on the second loop will change the dependence vector to $(1, 1)^T$, and make tiling legal. Before we construct such transformation, we need to know the dependence condition for tiling.

For a loop nest $i_1, .., i_n$, loops $i_k, ..., i_n$ can be tiled if any dependence carried by loops $i_k, ..., i_n$ has no negative entries. This is the simplified case of the *fully permutable* loop nest defined by Wolf and Lam [WL91a]. Any negative entry in the dependence vectors carried by the loops $i_1, ...i_{(k-1)}$ will not prevent the tiling of loops $i_k, ..., i_n$.

The algorithm in Figure 5.6 constructs a transformation matrix that reduces the height of the reuse vectors, and creates opportunity for tiling. It will find the largest innermost loop nest by minimizing the *blocking level* k. A row s is calculated to reduce the height of the reuse vectors, and to make the loop i_s tilable. In the algorithm, a dependence vector will flow from D, the dependence matrix carried by loops $i_k, ...i_n$, to D_o, the dependence matrix carried by loops $i_k, .., i_{(s-1)}$, and then be deleted if it is carried by the loop $i_1, ..., i_{(k-1)}$. When the vector from the null space of P and R is computed, we need to make sure that it does not violate the data dependences in D. If this fails, then no such row s exists. To guarantee that the nest $i_k, ..., i_s$ can be tiled, we will not generate

Input: *A reuse matrix R, a dependence matrix D and dimension n.*
Output: *A legal transformation matrix T that maximizes data reuse.*
Algorithm WidthReduction(R, D, n) : Matrix
begin
  k = 1; $P^T = \emptyset$; s = 0;
  While ($s < n$) *do*
    $Q^T = union(P^T, R^T)$;
    While (rank(Q) = n) delete the lowest priority reuse vector from R
    solve $Q^T v = 0$, *let* $v = a_1 x_1 + ... a_t x_t$
    choose $a_1, ..., a_t$ *such that* $v^T D = a_1 x_1^T D + .. + a_t x_t^T D \geq 0$,
      and try to optimize for $v^T D_o \geq 0$.
    if success then
      For (($d \in D_o$) *and* ($v^T d < 0$)) k = *max(k, r+1), if loop* i_r *carries* d.
      For (($d \in D$) *and* ($v^T d > 0$)) $D = D - d$; $D_o = D_o + d$
      For (($d \in D_o$) *and d carried by loops* $i_1, .., i_{(k-1)}$) $D_o = D_o - d$
      $P_s^T = v^T$; $s = s + 1$;
    else delete the reuse vector with the lowest priority from R
  End-While
  return(P^T);
end

Figure 5.6: Width Reduction

negative entry in any dependence vector in D_o. If a negative entry is generated for a dependence vector, d, then the highest tilable level, k, is the loop below the loop carrying d. Therefore, if we have to generate negative entries, it is better to give up the dependence vectors carried by the outer loops in loops $i_k, .., i_{(s-1)}$.

5.4.1 Tile Size and Cache Interferences

Another important issue is the block size. A naive approach is to choose a block size large enough that the block of data fits in the cache. This has been shown to cause very poor performance on real machines, which usually have direct mapped cache, or have caches with very small set associativity (2 or 4) [LRW91]. Performance can be erratic when tile size changes, and it is difficult to predict the optimal tile size. The main reason for performance degradation is cache interference, i.e. two elements are mapped to the same cache location. Width reduction also helps eliminate the erratic cache behavior because height reduction is performed as a part of width reduction. Experimental results in

Section 5.5.5 show that performance changes dramatically if tiling is directly applied to Cholesky decomposition, while tiling produces good performance independent of tile size after height reduction.

5.5 Experiments on Uniprocessor with Caches: HP 9000

5.5.1 The Machine Architecture

We conducted our experiments on an HP 9000/720, which has 256KB data cache, and 32MB main memory. A word has 4 bytes. The cache latency is 1 cycle, and the memory latency is 15 cycles. Benchmark programs are first optimized by *Pnuma*, and then compiled with the native FORTRAN compiler with the -O option.

We first show the results of banded SYR2K and the NASA benchmark from the SPEC benchmark suite, and then study the effect of data sets and loop tiling.

5.5.2 Banded SYR2K Results

We use banded SYR2K to show that our techniques can handle quite general data access patterns. We include the original program in Figure 5.7(a) for convenience. First, we compute the reuse matrix, shown in Figure 5.7(c), using the algorithm in Section 5.2. The reuse vectors computed from the references *Z(i, j-i+1)* are the second and third columns in the reuse matrix, which include both temporal and spatial reuses (Note that the temporal reuses are a subset of the spatial reuse. Therefore, only spatial reuse vectors are computed.); the reuse vectors (both temporal and spatial) for *X(k, i-k+1)* (also *Y(k, i-k+b)*) are the fourth and fifth columns of the reuse matrix; and the last two columns are the temporal and spatial reuse vectors for *Y(k, j-k+b)* (also *X(k, j-k+b)*). The first column is the basis vector of the intersection of all reuses, i.e. the most important reuse dimension. We then call the height reduction algorithm to produce the transformation matrix, shown in Figure 5.7(d), from that reuse matrix. Given the transformation matrix, we use the completion algorithm to generate the new loop nest, shown in Figure 5.7(b). The generation of the new loop nest requires handling general loop bounds with *max*, *min*, and variables, which is provided by our loop transformation framework. In the transformed version, height reduction exploits the spatial locality of the reference C, A, and B, with respect to the innermost loop. In the final stage, the loop nest is tiled to improve the temporal locality.

The execution times for the different versions are shown in Figure 5.7(e). The problem size n is 500. The time for the original version is 101.3 seconds. After

```
for i = 1, n
  for j = i, min(i+2*b-2, n)
    for k = max(i-b+1, j-b+1, 1),
            min(i+b-1, j+b-1, n)
      Z(i, j-i+1) = Z(i, j-i+1)
        +X(k, i-k+b)*Y(k, j-k+b)
        +X(k, j-k+b)*Y(k, i-k+b)
```

(a) Source

```
for u = max(2-2*b, 2-n-b, 0),
        min(-2+2*b, -2+n+b, -2+2*n, -1+n)
  for v = u+2*max(-u+1-b, 1-b, 1-n, -u+1-n),
          u+2*min(-u-1+b, -1+b, -u-1+n), 2
    for w = max(1, (u+v)/2+1),
            min(n, (-u+v)/2+n, (u+v)/2+n)
      Z((-i-j)/2+k, i+1) = Z((-i-j)/2+k, i+1)
        +X(w, (-u-v)/2+b)*Y(w, (u-v)/2+b)
        +X(w, (u-v)/2+b)*Y(w, (-u-v)/2+b)
```

(b) Transformed

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

(c) Reuse Matrix

$$\begin{pmatrix} -1 & 1 & 0 \\ -1 & -1 & 2 \\ 0 & 0 & 1 \end{pmatrix}$$

(d) Transformation

Versions	Source	Transformation	Strength Reduction
Times (sec.)	101.3	14.36	14.26
Speedup		7.05	7.10

(e) Execution Times

Figure 5.7: Banded SYR2K

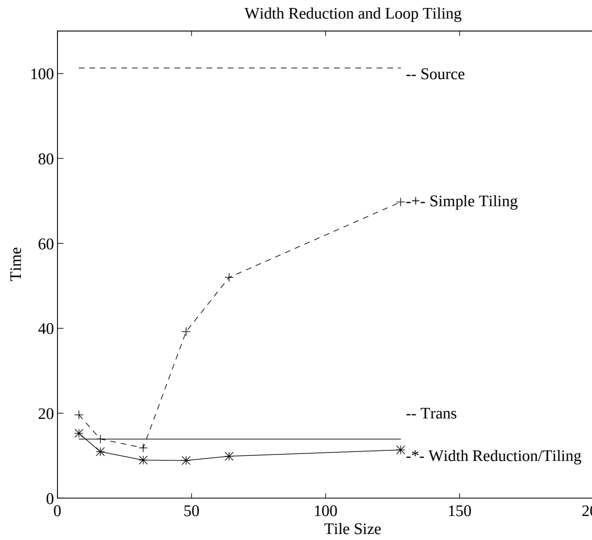


Figure 5.8: Width Reduction- banded rank-$2k$ Update

height reduction transformations, the time is reduced to 14.36 seconds, which has the speedup of 7.05. Strength reduction of the divisions in the program has not changed the performance.

We compare our results using the width reduction with the simple tiled version. The results are in Figure 5.8 in which *Source* represents the original version; *Simple Tiling* represents the simple tiling version; *Trans* represents the version with height reduction; and *Width Reduction/Tiling* represents the tiling version with width reduction. The performance of simple tiled version varies dramatically, when the tile size changes, while the performance of our version is quite stable, and always outpaces the simple tiling version.

5.5.3 NASA Benchmark Results

The NASA benchmark contains 7 floating point intensive programs written in Fortran 77. For each kernel, the program generates its own input data, performs the kernel and compares the result against an expected result.

A brief description of the programs and major data structures is list below.

- MXM: matrix multiply.

  The major data structures are three 2-dimensional arrays of sizes 256×128, 128×64, and 256×64.

- FFT: complex radix 2 FFT on 2D array.

  The major data structure is a 2-dimensional array of size 128×256.

- CHOLSKY: Cholesky decomposition on a set of input matrices. It also performs backward substitution.

  The major data structures are two 3-dimensional arrays of sizes $250\times4\times40$ and $3\times250\times40$.

- BTRIX: Block tridiagonal matrix solution along one dimension of a four dimensional array.

  The major data structures are a 4-dimensional array of sizes $30\times30\times30\times5$ and three 4-dimensional arrays of size $5\times5\times30\times30$.

- GMTRY: Sets up arrays for a vortex method solution and performs Gaussian elimination on the resulting arrays.

  The major data structures are one 2-dimensional array of size 500×500, and three 2-dimensional arrays of size 100×5.

- EMIT: Creates new vortices according to certain boundary conditions.

  The major data structures are one 2-dimensional array of size 500×500, and four 2-dimensional arrays of size 100×5.

- VPENTA: inverts 3 pentadiagonal matrices.

  The major data structures are two 3-dimensional arrays of size $128\times128\times3$, and seven 2-dimensional arrays of size 128×128.

We applied the height reduction algorithm to these programs, and Table 5.1 contains the performance results. No loop transformations have been applied to FFT, GMTRY and EMIT. There are a few reasons for this: there are data dependences that prevent loop transformation, the loops are imperfectly nested, or the loops are already in the right form for cache locality.

For MXM and CHOLSKY, transformations are applied, but no improvement are seen. This is because that the test data set is so small that they almost fit in the caches [2], and furthermore MXM has been hand-optimized for locality by unrolling the outermost loop 4 times. We will study the effect of the size of the data sets in Section 5.5.4.

For BTRIX and VPENTA, since both have arrays that are larger than the data cache, we see improvement from our transformations. If the data sets were even larger, we would expect to see much better improvement.

5.5.4 The Effect of Data Sets

In this section, we study matrix multiplication and Cholesky decomposition with larger data sets.

[2] Recall that the size of the data cache on the HP 9000/720 is 64K words.

Table 5.1: NASA benchmark on HP 9000/720

Prog	MXM	FFT	CHO	BTRIX	GMTRY	EMIT	VPENTA
Source	14.11	93.05	40.61	55.78	111.34	15.80	113.50
New	14.24	N/A	40.70	34.37	N/A	N/A	99.52
Speedup	1		1	1.64			1.12

A natural way of writing matrix multiplication is the simple ijk form in Figure 5.9(a). However, this version is not suitable for cache locality, because, for example, the spatial locality of neither C nor A is exploited, and there is only temporal locality of C and spatial locality of B. The NASA kernel chooses the kji form, and then unrolls the k-loop 4 times. Figure 5.9 gives the execution times for the problem size of 750.

Our height reduction algorithm will transform both versions to the new version in Figure 5.9. In the transformed version, the spatial locality of C is exploited by the innermost loop, the temporal locality is carried by the second loop. References A and B have spatial locality and temporal locality respectively. As expected, the new version performs significantly better, requiring only 68.65 seconds. Compared to the ijk version, the speedup is 5.88; compared to the kji version, the speedup is 10.

Another example is the Cholesky Decomposition, shown in Figure 5.10(a). We would like to study data sizes ranging from 250 to 2000. Since the cache size on the HP 9000/720 is 64K words, a two dimensional array of 250×250 is about the largest that can fit in the cache. The original code has poor locality, since the inner loop walks across the columns, while FORTRAN uses column major form for array layout. The cache lines (column segments) may be displaced before the other elements in the same cache line can be accessed. The transformation is a loop interchange, and the new version is shown in Figure 5.10(b).

The results are listed in Figure 5.10(c). When the problem size is 250, there is almost no improvement. This is expected, since the array is small enough to fit in the cache. When the size is 500, we see a moderate speedup of 2. The improvement is significant when the size is larger than 1000. We get speedup of 5 over the original version.

5.5.5 The Effect of Tiling

We use Cholesky decomposition as the example to illustrate the effect of tiling. If we apply tiling directly on the ij-nest in Figure 5.10(a), we get the results in Table 5.2 for a problem of size 1000.(The original version took 196.05 seconds.) The performance varies greatly, when the block size changes. Although tiling is an important optimization, it is very difficult to predict the optimal block

for i = 1, n
  for j = 1, n
    for k = 1, n
      *C[i, j] = C[i, j] + A[i, k]*B[k, j]*

(a) *ijk*

for k = 1, n
  for j = 1, n
    for i = 1, n
      *C[i, j] = C[i, j] + A[i, k]*B[k, j]*

(b) *kji*

for u = 1, n
  for v = 1, n
    for w = 1, n
      *C[w, u] = C[w, u] + A[w, v]*B[v, u]*

(c) new

Versions	*ijk*	*kji*
Source	379.69	708.37
Transformed	68.65	68.65
Speedup	5.88	10

(d) Execution Times (Size = 750)

Figure 5.9: Matrix Multiplication

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

(a) Source

```
for k = 1, n
  A(k,k) = sqrt(A(k,k))
  for i = k+1, n
    A(i,k) = A(i,k)/A(k,k)
  end for
  for u = k+1, n
    for v = max(u, k+1), n
      A(v,u) = A(v,u)-A(v,k)*A(u,k)
    end for
  end for
end for
```

(b) Transformed

Size	250	500	750	1000	1250	1500	1750	2000
Source	0.61	11.71	63.41	196.05	453.25	704.65	1161.74	1748.19
Trans	0.59	5.74	19.17	46.49	91.12	157.29	249.65	373.15
Speedup	1.03	2	3.33	4.35	5	4.55	4.55	4.55

(c) Execution Times

Figure 5.10: Cholesky Decomposition

Table 5.2: Loop Tiling (Size 1000)

Block Size	16	32	64	100	200
Times (sec.)	60.77	56.15	55.74	175.08	184.85

Table 5.3: Loop Tiling after Height Reduction (Size 1000)

Block Size	16	32	64	100	200
Times (sec.)	52.78	50.12	48.16	49.09	47.89

size [LRW91]. If a random block size is chosen, the performance may be very poor.

In our approach, we apply height reduction first. The reuse vector that has the most number of occurrences is $(1,0)^T$, which is computed from all of the four references. The execution time of the untiled transformed version is 46.49 seconds (Figure 5.10(c)), about more than 20% improvement over the tiled untransformed version. The actual percentage depends on the choice of the block size. If we apply tiling after height reduction, the block size is not a dramatic factor causing big performance differences. Therefore, this optimization is more stable, since it does not depend on the cache mapping, cache size and problem size. There is some performance degradation due to the extra overhead of tiling. The results are shown in Table 5.3.

5.6 Discussion and Related Work

In this section, we discuss related work.

Gannon, Jalby and Gallivan [GJG88] introduced the notion of *uniformly generated* data reuse. Porterfield [Por89] studied the problem of estimating the number of cache lines for uniprocessor machines, when the cache line size is 1. The techniques by Ferrante, Sarkar, and Thrash [FST91] estimate the the number of distinct cache lines used by a given loop in a loop nest. Given this estimate, they compute the number of cache misses for a loop nest.

Wolf and Lam [WL91a] focus on loop tiling of the innermost loops as a means of achieving cache locality. They try all possible subset of the loops in the loop nest, and then try to bring that subset into the innermost position. The subset that can be brought into the innermost position, and has the best objective function from the *reuse vector space* model is chosen to be tiled. The cost function computed using the reuse vector space model is a function of the *dimension* of the reuse vector space, i.e. *number* of loops that carry reuse.

Therefore, the order of loops is irrelevant as long as the loops carry reuse. If data dependences prevent a subset to become the innermost loop nest, unimodular transformations are applied so that the inner most loop nest may be tilable. For example, the reuse vector space for the ij-nest in the Cholesky decomposition in Figure 5.10(a) is the whole space that include both i and j dimensions, which has the maximum reuse (the maximum reuse dimension is 2, in the 2-dimensional space), according to the reuse vector space model, and the subset that includes loop i and loop j is tilable. Therefore, loop tiling is performed on the loop nest ij. The experimental results are reported in Table 5.2.

Our approach is based on reuse vector and reuse distance (we refer to this as the *reuse distance model*), rather than the dimension of the reuse space. Therefore, the order of loops does make a difference in our model. It takes the reduction of reuse vector height as the primary optimization. In the Cholesky example, by reducing the reuse vector $(1,0)^T$, which is the basis vector of the intersection of all reuses, we get the new version in Figure 5.10(b), which would have the same cost function as the original version under the reuse space cost model. Therefore, our *reuse distance model* with the reuse vectors and reuse distances improves upon the reuse vector space model, and is a more precise than the reuse space model. This allows us to develop algorithms to get better performance. The experimental results are shown in Table 5.3, which shows that our approach generates the better results. The second advantage is that since the transformations are operated directly on the reuse vectors, no exhaustive search is necessary. The third advantage is that the performance of the generated code is much less dependent on the tile size. Another advantage of the reuse distance model is that we can represent more complicated data reuses such as the data access patterns in the banded SYR2K example in Figure 5.7. For the reuse vector $(1,1,0)$, the reuse space model will make an approximation by having both loops carry reuse, but we can optimize on it directly without losing the precision.

Chapter 6

Improving Parallelism and Locality

6.1 Introduction

We have seen that loop transformations can be applied to improve memory locality and cache locality. In this chapter, we first show that our loop transformation framework can be used to improve parallelism as well. Then, we develop a unified algorithm for parallelism, memory locality, and cache locality.

Together with the locality-driven code generation scheme presented in Chapter 2, we have a complete parallelizing system for NUMA architectures.

Empirical results of Cholesky decomposition and SIMPLE, a hydrodynamics program will be presented. The experiments were conducted on the KSR1.

This chapter is organized as follows: first we define *dependence summary vector*, which summarizes the data dependences in the loop nest, and also serves as the objective function to be optimized; second, we show an algorithm that takes the data access matrix, and optimizes for parallelism with memory locality; third, we present the complete algorithm for parallelism, memory locality and cache locality. Finally experimental results are presented.

6.2 Dependence Summary Vector

We define *dependence summary vector* to summarize data dependences in a loop nest.

Definition 6.2.1 Given a dependence vector d, if the ith entry has the first non-zero, the dependence vector is said to be *carried* by the loop i.

A loop is a parallel loop, if it does not carry any data dependences.

Definition 6.2.2 Given a loop nest of depth n, a *dependence summary vector* is an integer vector, where the ith entry is the number of dependences carried by loop i.

Let pv_1 and pv_2 be two dependence summary vectors, then $pv_1 < pv_2$ if there exists an i such that $pv_1[1 : i-1] = pv_2[1 : i-1]$ and $pv_1[i] < pv_2[i]$.

Now we show that loop transformations can reduce the dependence summary vector. Consider parallelizing the following program for a MIMD machine.

for i = 4, 8
for j = 3, 8
A[i, j] = A[i-3, j-2] + 1;

The dependence matrix for this program is $D = \begin{pmatrix} 3 \\ 2 \end{pmatrix}$

The outermost loop is parallel if and only if it does not carry any dependences; that is, the first entry of every dependence vector is 0. In our example, the outermost loop is not a parallel loop, since iteration i depends on iteration $i-3$. We can transform the loop nest into one in which the outermost loop is parallel if we can find a transformation T such that every entry in the first row of TD is 0. Therefore, the condition that must be satisfied for transformation $T = \begin{pmatrix} t_{11} & t_{12} \\ t_{21} & t_{22} \end{pmatrix}$ is that $\begin{pmatrix} t_{11} & t_{12} \end{pmatrix} \begin{pmatrix} 3 \\ 2 \end{pmatrix} = 0$. The condition can be satisfied by choosing $t_{11} = 2$ and $t_{12} = -3$. This determines the first row of the transformation matrix, and now we must add additional row(s) to get a non-singular matrix that respects all the dependences of the loop nest.

In the nest section, we show how to increase coarse-grain parallelism with memory locality.

6.3 Transformation for Parallelism

The problem of improving coarse grain parallelism can be formulated as follows: we would like to generate as many parallel outermost loops as possible. If a loop has to carry dependences, it is better to carry less dependences in order to reduce synchronization.

For coarse grain parallelism, the problem is to minimize the dependence summary vector. A straightforward approach will be to find a partial transformation in the null space of D so that the outermost loops will be parallel. This is the approach taken by Banerjee [Ban90] and Wolf/Lam [WL91a]. to construct a unimodular matrix to increase coarse-grain parallelism. For NUMA architectures, data locality as well as parallelism is critical to high performance. Our approach is to take both into account.

We will construct our non-singular transformations from the data access matrix from Chapter 4. If a row from the data access matrix is in the null space

Input: *Candidate matrix C, dependence matrix D and dimension n.*
Output: *A partial transformation P for coarse grain parallelism.*
Algorithm Parallelism(C, D, n) : Matrix
begin
  $P^T = \emptyset$; $s = 0$; $L^T = \emptyset$; $t = 0$;
  For $i = 1$, *NumberOfRows(C)*
    c_i^T = *row* i *of* C;
    If $c_i^T D = 0$ *and* c_i *linearly independent of* P^T *then*
      $P_s^T = c_i^T$; $s = s + 1$;
    else
      $L_s^T = c_i^T$; $t = t + 1$;
    End-If
  End-For
  return(P^T, L^T);
end

Figure 6.1: Improving Parallelism

of D, then we can improve both parallelism and memory locality by making that row part of the partial transformation. The algorithm in Figure 6.1 is simple. Since the data access matrix is ordered with the most important rows at the beginning, we search from the beginning of the matrix.

6.4 A Unified Algorithm for Parallelism and Locality

In this section, we show that the techniques we have developed for memory locality, cache locality and parallelism can be unified into a single algorithm.

The techniques for memory locality developed in Chapter 4 can be extended to take a partial transformation, rather than starting from an empty transformation matrix. The function *MemoryLocality* takes a partial transformation, a data access matrix, and a dependence matrix, and returns a new partial transformation. Dependences carried by the input partial transformation are deleted from the dependence matrix. With the data access matrix and the updated dependence matrix, we can call the algorithms *BasisMatrix* and *LegalBasis* from Chapter 4 to generate a partial transformation. This partial transformation is then appended to the input partial transformation to generate a new transformation as the returned partial transformation of the function *MemoryLocality*.

Input: *Data access matrix, reuse matrix and dependence matrix.*
Output: *A legal transformation T for parallelism and locality*
Algorithm ParallelismAndLocality(X, R, D, n) : Matrix
begin

$(P^T, X) = Parallelism(X,\ D);$

$(P^T, D) = MemoryLocality(P^T,\ X,\ D);$

$(P^T, D) = CacheLocality(P^T,\ R,\ D)$

$T = Completion(P^T,\ D)$

end

Figure 6.2: Improving Parallelism and Locality

Similarly, the techniques for cache locality developed in Chapter 5 can also be extended to take a partial transformation as an input. The function *CacheLocality* takes a partial transformation, a reuse matrix and a dependence matrix, and returns a new partial transformation. The reuse vectors carried by the input partial transformation are deleted from the reuse matrix. The dependence vectors carried by the partial transformation are deleted from the dependence matrix as well. For the algorithm in Figure 5.4, we can simply start with the partial transformation.

If the transformation matrix generated by *Parallelism*, *MemoryLocality* and *CacheLocality* is still a partial transformation, then the completion algorithm in Section 4.6 will be called to generate a legal full transformation.

6.5 Experiments on Multiprocessor with Caches: KSR1

6.5.1 The Machine Architecture

KSR1 is a 64-bit shared memory multiprocessor. Each KSR1 processor is connected to a *subcache*, which is the first level cache. The subcache has 0.5MB divided into a 0.25 MB instruction subcache and a 0.25MB data subcache. The second level cache, called the *local cache*, has 32MB.

The processors are connected in a hierarchy of rings. Thirty two processors can be connected via a ring to form a *ring0*. At most 34 ring0's can be connected to form a *ring1*.

Table 6.1: KSR1 Memory Hierarchy

From:	Cycles	Size
Subcache	2	256KB
Local Cache	18	32MB
Ring 0	126	1GB
Ring 1	600	34GB

The memory system on the KSR1 is different from the traditional multiprocessor memory system: the mapping between the physical address space and the set of processors is not fixed. Local memory (*local cache*) also behaves like a cache [1]. The contents of the 64-bit *System Virtual Address* space are stored physically in the set of local caches. Data moves at the point of reference on demand. Data replication is allowed: multiple local caches can have a copy of a memory location. A *read* request will result in searching for the data in the subcache first. If it is not found in the subcache, then the local cache on the same processor is searched. If it is a not found there, the memory system will retrieve a copy from a processor that has the data, and return a copy to the processor that made the read request. Therefore, multiple copies of the same data may be created. A *write* request will invalidate all copies of the data, and makes the requesting processor the *owner* of the data. The memory system implements a *sequentially consistent* shared memory space programming model [Lam79].

As expected, it is much faster to access the subcache than local cache, which is faster to access than the local cache of some other processor from the same ring0, which is faster to access than the local cache of some processor from the same ring1 but not the same ring0. The latencies and memory capacities are listed in Table 6.1. The ratio of a ring1 memory access latency and a subcache access latency is 300.

From the programming point of view, it is critical to have the data locality in order to get high performance. We need both memory and cache locality as well as parallelism.

Pnuma takes programs in FORTRAN, and generates programs in FORTRAN with KSR parallel constructs. The native KSR FORTRAN compiler with the Presto runtime system is used to generate executable code. The native scalar optimizations are used. The timing is recorded using the KSR Performance Monitor, Pmon.

[1] *ALLCACHE* is the name of the memory system.

```
for k = 1, n
  A(k,k) = sqrt(A(k,k))
  TILE (i)
  for i = k+1, n
    A(i,k) = A(i,k)/A(k,k)
  end for
  END TILE
  TILE (i)
  for i = k+1, n
    for j = k+1, i
      A(i,j) = A(i,j)-A(i,k)*A(j,k)
    end for
  end for
  END TILE
end for
```

(a) KAP

```
for k = 1, n
  A(k,k) = sqrt(A(k,k))
  for i = k+1, n
    A(i,k) = A(i,k)/A(k,k)
  end for
  PARALLEL REGION
  (lb,ub,s) = get_bounds(k+1,n,1,p,P)
  for u = lb, ub, s
    for v = max(u, k+1), n
      A(v,u) = A(v,u)-A(v,k)*A(u,k)
    end for
  end for
  END PARALLEL REGION
end for
```

(b) Pnuma

Figure 6.3: Cholesky Decomposition on KSR

6.5.2 Cholesky Decomposition

We studied Cholesky decomposition and compared our result with KAP, a commercial parallelizing compiler on the KSR1. The original version is shown in Figure 5.10(a). The KAP-generated version is shown in Figure 6.3(a). KAP decided to tile the outer parallel loops. The two i-loops are parallelized. The Pnuma-generated version is shown in Figure 6.3(b). Pnuma assumes that the arrays are distributed by columns. Loop transformations are applied first. The new u-loop is parallelized, since it is the outermost parallel loop that carries data locality, using the code generation scheme from Chapter 2.

The problem size used is 1000. The original sequential version took 584.81 seconds. Table 6.2 has the execution times and speedups of both KAP and Pnuma versions. The Pnuma version had a speedup of 21.58 on 4 nodes! The row *Speedup1* in the table contains the speedups over the original version. The reason for the extraordinary speedup is that loop transformations helped exploit data reuse in subcaches and local caches. The ratio of accesses to local cache and subcache is 9. Therefore, there will be a factor of 9 speedup between a version with good locality and a version with poor locality. The access latency of remote accesses is even longer. To see the cache reuse effect on the sequential version, we also ran the transformed version on one node. The execution time of the transformed version is 98.5 seconds, a speedup of about 6 over the original

Table 6.2: Cholesky Decomposition on KSR1 (time unit = second)

Prog/Nodes	Nodes = 4	Nodes = 8	Nodes = 16	Nodes = 32
KAP	249.45	162.54	95.41	60.32
Speedup	2.34	3.6	6.14	9.6
Pnuma	27.09	14.17	8.05	5.11
Speedup1	21.58	41.27	72.6	113
Speedup2	3.63	6.95	10.88	19.27

Table 6.3: SIMPLE on KSR1 (Size=400)

Prog/Nodes	Nodes = 4	Nodes = 8	Nodes = 16	Nodes = 32
Pnuma	8.12	4.7	2.81	2.11
Speedup	3.9	6.8	11.5	15.29

version. Then we compute the speedups over the transformed one node version to get the row *Speedup2*.

6.5.3 SIMPLE Benchmark Results

SIMPLE is a hydrodynamics program from Lawrence Livermore Labs, which has about 2400 lines of FORTRAN. The program has about 20 two-dimensional arrays and a few one-dimensional arrays as work space. There are 25 subroutines and 3 functions.

A column distribution is assumed since the arrays in FORTRAN have the column major form. There was a slight change to the source code. The initial values for computing the inverse of a polynomial were dependent on the previous function call. This dependence was deleted to increase parallelism, since it didn't affect the program correctness. Variable privatization was not implemented, but done by hand. We ran one iteration of the problem on a grid of size 400×400. The sequential version took 32.28 seconds. Table 6.3 contains the results from KSR1.

6.6 Discussion and Related Work

In this chapter, we have put all of the techniques developed in this thesis together. The central issue is that, to get high performance, both parallelism and data locality have to be optimized. Program transformations were shown to be very effective in improving memory locality, cache locality and parallelism.

Chapter 7

Conclusions

We have built a system called *Pnuma* for programming NUMA machines. We make the following contributions:

First, we take sequential programs, and generate parallel code exploiting both parallelism and data locality.

Second, we have developed a framework based on *non-singular* matrices and integer lattice theory for the systematic development of loop transformations. It can be used in parallelizing compilers for both coarse-grain and fine-grain parallel architectures. We have implemented a loop restructuring tool-kit, called *Lambda*, based on this framework.

Third, using this loop transformation framework, we have developed algorithms for improving memory locality. The memory locality algorithm restructures loop nests to expose opportunities for parallel execution and for block transfers, while keeping data accesses local wherever possible.

Fourth, for cache locality, we have introduced a new simple cache model based on *reuse distances*, that is more precise than the existing *reuse vector space* model. We have developed a new loop transformation technique that optimizes directly on reuse distances, so that no exhaustive search is necessary.

Fifth, we have used our loop transformation framework to improve parallelism as well. We have developed a unified algorithm for parallelism, memory locality and cache locality.

Finally, system evaluations have been conducted on a multiprocessor machine without cache: BBN GP1000, a uniprocessor workstation with cache: HP 9000/720 and a multiprocessor machine with caches: KSR1, using programs from linear algebra, NASA benchmarks and SIMPLE hydrodynamics benchmark.

The techniques developed in this thesis can be applied, and extended in various ways.

- Imperfectly nested loops

So far, our matrix theory works on *perfectly nested* loops, or those that can be made perfectly nested by adding conditionals. In real applications, many loops are not perfectly nested. It is both intellectually challenging and practically important to extend the theory to handle general loop nests.

- Eliminating *false sharing*

  It is important to eliminate *false sharing* for coherent caches, or *shared virtual memory* [LH89]. For dense matrix problems, we may be able to apply the matrix approach to analyze the data access patterns, and lay out "unrelated" data in different cache lines or memory pages.

Appendix A

Pnuma Compiler

The Pnuma compiler is implemented in C. The front-end takes a FORTRAN program and generates an abstract syntax tree. Optimizations are performed on the abstract syntax tree. FORTRAN source code is generated. Loop transformation algorithms are implemented using the Lambda toolkit, which is described in Appendix B.

The compiler options are list below. DOALL loops are generated for comparison. Transformations for multiprocessors include both transformations for memory locality and cache locality. Transformations for uniprocessors include transformations for cache locality only. The statistics option outputs loop profiling information such as the number of loops with certain depth.

- Parallelization:
  - parallelizing DOALL loops
  - parallelizing for locality
- Optimizations:
  - transformations for multiprocessors
  - transformations for uniprocessor
- Statistics:
  - loop statistics
- Debugging:
  - demo/debugging mode
  - Lambda debugging mode

Appendix B

Lambda Transformation Toolkit

B.1 Introduction

In this chapter, we describe the *Lambda* toolkit, which is based on the framework developed in Chapter 3. The data structures and routines from the following modules are described:

- Data Dependence Module
- Transformation Module
- Code Restructuring Module
- Utility Module

The toolkit is independent of the intermediate representation used in the compiler. This enabled Bergmark and Presberg from Cornell Theory Center have integrated it with Parascope, a parallelizing environment from Rice University [BP93].

B.2 Data Dependences

B.2.1 Data Types

A data dependence can be specified with either *distance* or *direction*. A distance is represented by an integer constant, and a direction is represented by the following symbols.

```
typedef enum {
  dK,
  dLT,
  dLEQ,
```

```
  dEQ,
  dGEQ,
  dGT,
  dDOT,
  dLG,
  dSTAR,
  dSIZE
  } LA_DIR_T;
```

dK represents distance, where its value is stored in another variable. The data type for a dependence is a structure with two fields.

```
typedef struct {
  LA_DIR_T dir;
  int dist;
} LA_DEP_T;
```

A data dependence vector is a vector of dependences represented by a linked list.

```
typedef struct _vector {
  LA_DEP_T * vector;
  struct _vector *next;
} *LA_DEP_V_T, LA_DEP_V_T1;
```

And a dependence matrix is a set of dependence vectors.

```
typedef struct{
  LA_DEP_V_T vectors;
  int dim;
  int size;
} *LA_DEP_M_T, LA_DEP_M_T1;
```

B.2.2 Routines

There are routines provided to operate on the dependences, some of which are listed here. A legal dependence vector should be lexicographically positive. A data dependence analyzer may return a dependence vector with illegal components, i.e. *not* lexicographically positive. *la_dep_legal* deletes the illegal components. For example, the dependence vector $(*, 1)$ becomes $(<, 1)$ and $(=, 1)$. The illegal component $(>, 1)$ is deleted.

- Creates a dependence vector.

  ```
  LA_DEP_V_T la_dep_vec_new( int size );
  ```

- Frees a dependence vector.

  ```
  void la_dep_vec_free( LA_DEP_V_T d );
  ```

- Creates a dependence matrix.

  ```
  LA_DEP_M_T la_dep_matrix_new(int dim, int size);
  ```

- Deletes the illegal components from a dependence matrix.

  ```
  LA_DEP_M_T la_dep_legal(LA_DEP_M_T D);
  ```

- Eliminates redundant data dependences.

  ```
  LA_DEP_M_T  la_dep_no_redundant(LA_DEP_M_T D);
  ```

B.3 Constructing Transformations

B.3.1 Data Types

A matrix type is a two dimensional matrix. The field *denom* is used to represent a rational matrix, A/d, where A is an integer matrix, and d is an integer.

```
typedef struct {
  int ** matrix;
  int row_size;
  int col_size;
  int denom;
} *LA_MATRIX_T;
```

To create a matrix structure with *rowsize* and *colsize*, we can use the following routine.

- `LA_MATRIX_T la_matrix_new(int rowsize, int colsize);`

B.3.2 Nonsingularity

This set of routines are useful in the construction of a non-singular transformation matrix.

- Checks if the transformation matrix is nonsingular.

```
int la_is_nonsingular(LA_MATRIX_T T);
```

- Checks if the transformation matrix has full rank.

```
int  la_is_fullrank(LA_MATRIX_T pT);
```

- Computes the rank of a matrix.

```
int  la_rank(LA_MATRIX_T pT);
```

- Computes a base matrix.

```
LA_MATRIX_T la_base(LA_MATRIX_T pT);
```

- Extends a base matrix to a nonsingular matrix.

```
LA_MATRIX_T  la_padding(LA_MATRIX_T pT);
```

B.3.3 Data Dependences

This set of routines assists the construction of a legal transformation.

- Checks if a full transformation is legal with respect to the data dependences.

```
int  la_is_legal(LA_MATRIX_T T, LA_DEP_M_T D);
```

- Checks if a partial transformation is legal.

```
int  la_is_legal_par(LA_MATRIX_T pT, LA_DEP_M_T D);
```

- Computes a legal base matrix by deleting the rows that violate the dependences.

```
LA_MATRIX_T  la_base_legal(LA_MATRIX_T pT, LA_DEP_M_T D);
```

- Extends the legal base matrix with respect to the dependence matrix.

```
LA_MATRIX_T  la_padding_legal(LA_MATRIX_T pT, LA_DEP_M_T D);
```

B.4 Code Restructuring

A loop nest is represented by a list of loops, where each loop contains a lower bound, a upper bound, and a step size. Each bound is a list of linear expressions. A lower bound can be the maximum of a set of linear functions, and a upper bound can be the minimum of a set of linear functions.

Variables are allowed in the loop bounds as long as they are loop invariants. We call them *blobs*. In general, a loop bound can have a linear expression of blobs as a part of the bound.

B.4.1 Computing Loop Bounds

First, simple integer vectors and matrices are defined.

```
typedef int * la_vect;
typedef la_vect * la_matrix;
```

B.4.1.1 Linear Expressions

The linear expression type has a field for the coefficients of the loop index variables and the coefficients of the blobs.

```
typedef struct _la_expr{
  la_vect coef;
  int c0;
  la_vect blob_coef;
  int denom;
  struct _la_expr *next;
} *LA_EXPR_T;
```

For example, let i, j be the loop index variables. An example of a linear expression is $(i + j + 2 + x + y)/3$. The coefficient vector contains the linear coefficients ($coef = (1\ 1)$ for $i + j$). The integer constant is in *c0* ($c0 = 2$). The blob coefficients are (1, 1), since there are only two blobs and we consider x as blob 1, and y as blob 2. The denominator is 3.

We can allocate and free an expression with the dimension *dim* and total number of blobs *blobs* using the following routines.

- `LA_EXPR_T la_expr_new(int dim, int blobs);`

- `void la_expr_free( LA_EXPR_T expr );`

B.4.1.2 Single Loops

A loop has a lower bound, an upper bound, and a step size. The lower bound can be the maximum of a set of linear expressions defined in the previous section, and a linear expression may have an implicit ceiling function since only integer functions are allowed. An upper bound can be the minimum of a set of linear expressions. The floor function is implicit.

For example,

```
DO i = max( ceil((i+j)/2), ...), min( floor((2i)/3), ...), step 2
```

The loop type has a list of expressions for both lower and upper bounds. The offset expression is used in the new bounds.

```
typedef struct _la_loop{
  LA_EXPR_T low;
  LA_EXPR_T up;
  int step;
  LA_EXPR_T offset;
} *LA_LOOP_T;
```

The new loop bounds after a nonsingular transformation can be more complex. The bounds may have a linear offset, and an integer factor that equals to the step size. The offset and factor are the same for both lower and upper bound in the same loop.

For example,

```
DO v = u + 2*max( ceil((u+v)/2), ...), u + 2*min(...), step 2
```

These two routines allocate and free a loop structure.

- `LA_LOOP_T la_loop_new( void );`
- `void la_loop_free( LA_LOOP_T loop );`

B.4.1.3 Loop Nests

A loop nest is an ordered array of loops with first being the outermost loop and the last being the innermost. *depth* is the depth of the loop nest. *blobs* is the total number blobs.

```
typedef struct _la_loopnest{
  LA_LOOP_T *loops;
  int depth;
  int blobs;
} *LA_LOOPNEST_T;
```

These routines allocate and free a loop nest.

- `LA_LOOPNEST_T la_nest_new(int depth, int blobs );`
- `void la_nest_free( LA_LOOPNEST_T nest );`

la_nest takes a loop nest, and the transformation matrix and returns the new loop nest.

- `LA_LOOPNEST_T la_nest(LA_LOOPNEST_T nest, LA_MATRIX_T T);`

B.4.2 Computing Loop Body

The loop body can be considered as a set of *simple vectors*. Each vector can be transformed using *la_vector*.

A simple vector is a piece-wise linear function.

```
typedef struct {
  la_vect coef;
  int size;
  int denom;
} *LA_VECTOR_T;
```

The following routines allocate and free a simple vector.

- `LA_VECTOR_T la_vector_new( int size );`
- `void la_vector_free( LA_VECTOR_T v );`

la_vector computes the new expression in the transformed loop nest. The input is the inverse of the transformation.

- `LA_VECTOR_T la_vector(LA_MATRIX_T invT, LA_VECTOR_T v);`

B.5 Utility Routines

There are many other routines available in the toolkit. These routines include vector and matrix operations, algebraic operations on dependences, print routines for debugging, and so on.

Bibliography

[Aga91] A. Agarwal. Limits on interconnection network performance. *IEEE Transactions on Parallel and Distributed Systems*, 2(4):398–412, October 1991.

[AI91] C. Ancourt and F. Irigoin. Scanning polyhedra with DO loops. In *Third ACM Symposium on Principles and Practice of Parallel Programming*, pages 39–50, April 1991.

[AK87] R. Allen and K. Kennedy. Automatic translation of FORTRAN programs to vector form. *ACM Transactions on Progamming Languages and Systems*, 9(4):491–542, October 1987.

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

[Ban90] U. Banerjee. Unimodular transformations of double loops. In *Proceedings of the Workshop on Advances in Languages and Compilers for Parallel Processing*, pages 192–219, August 1990.

[BBN89] BBN Advanced Computers Inc. *Butterfly GP1000 Switch Tutorial*, 1989.

[BFKK90] V. Balasundaram, G. Fox, K. Kennedy, and U. Kremer. An interactive environment for data partitioning and distribution. In *Proc. 5th Distributed Memory Comput. Conf.*, April 1990.

[BL92] M. Barnett and C. Lengauer. Loop parallelization and unimodularity. Technical Report ECS-LFCS-92-197, University of Edinburgh, 1992.

[BP93] D. Bergmark and D. Presberg. Initial experiments in the integration of Parascope and Lambda. Technical Report CTC93TR136, Cornell Theory Center, 1993.

[Cas59] J. W. S. Cassels. *An introduction to the geometry of numbers.* Berlin, Springer, 1959.

[CK88] D. Callahan and K. Kennedy. Compiling programs for distributed memory multiprocessors. *The Journal of Supercomputing*, 2(2), October 1988.

[CVL88] T. Coleman and C. Van Loan. *Handbook for Matrix Computations*. SIAM Publication, Phil, 1988.

[DE73] G. B. Dantzig and B. C. Eaves. Fourier-motzkin elimination and its dual. *Journal of Combinatorial Theory(A)*, 14:288–297, 1973.

[Dow90] M. Dowling. Optimal code parallelization using unimodular transformations. *Parallel Computing*, 16:157–171, 1990.

[EA87] K. Ekanadham and Arvind. SIMPLE Part I: An exercise in future scientific programming. Technical Report RC12686, IBM, 1987.

[EZ92] D. L. Eager and J. Zahorjan. Adaptive guided self-scheduling. Technical Report 92-01-01, University of Washington, January 1992.

[FST91] J. Ferrante, V. Sarkar, and W. Thrash. On estimating and exchange cache effectiveness. In *Proceedings of the Fourth Workshop on Languages and Compilers for Parallel Computing*, August 1991.

[GJG88] D. Gannon, W. Jalby, and K. Gallivan. Strategies for cache and local memory management by global program transformaions. *Journal of Parallel and Distributed Computing*, 5:587–616, 1988.

[GVL89] G. H. Golub and C. F. Van Loan. *Matrix Computations*. Johns Hopkins University Press, 1989.

[HA90] D. Hudak and S. Abraham. Compiler techniques for data partitioning of sequentially iterated parallel loops. In *Proc. ACM Int. Conf. Supercomputing*, June 1990.

[HKT91] S. Hiranandani, K. Kennedy, and C. Tseng. Compiler optimizations for FORTRAN-D on MIMD distributed-memory machines. Technical Report TR91-156, Rice University, April 1991.

[HP90] J. Hennessy and D. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, 1990.

[HW79] G. H. Hardy and E. W. Wright. *An introduction to the theory of numbers*. Oxford University Press, 1979.

[IFKF90] K. Ikudome, G. Fox, A. Kolawa, and J. Flower. An automatic and symbolic parallelization system for distributed memory parallel computers. In *Proc. of the 5th Distributed Memory Comp. Conf.*, April 1990.

[Int91] Intel Corporation. *iPSC/i860 System Overview*, 1991.

[Ken91] Kendall Square Research Corporation, 170 Tracer Lane, Waltham, Ma 02154. *Parallel Programming Manual*, 1991.

[KLS90] K. Knobe, J. Lukas, and G. Steele. Data optimization: Allocation of arrays to reduce communication on SIMD machines. *Journal of Parallel and Distributed Computing*, 8:102–118, February 1990.

[KM91] C. Koelbel and P. Mehrotra. Compiling global namespace parallel loops for distributed execution. *IEEE Transactions on Parallel and Distributed Systems*, 2, October 1991.

[Lam74] L. Lamport. The parallel execution of do loops. *Communications of the ACM*, pages 83–93, February 1974.

[Lam79] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, pages 690–691, September 1979.

[LC89] J. Li and M. Chen. Index domain alignment: Minimizing cost of cross-referencing between distributed arrays. Technical report, Yale University, 1989.

[LC91] J. Li and M. Chen. Compiling communication efficient program for massively parallel machines. *IEEE Transactions on Parallel and Distributed Systems*, 2:361–376, July 1991.

[LH89] K. Li and P. Hudak. Memory coherence in shared virtual memory systems. *ACM Transactions on Computer Systems*, November 1989.

[LP92] W. Li and K. Pingali. Access Normalization: Loop restructuring for NUMA compilers. In *Proc. 5th International Conference on Architectural Support for Programming Languages and Operating Systems*, October 1992.

[LP93a] W. Li and K. Pingali. Access Normalization: Loop restructuring for NUMA compilers. *ACM Transactions on Computer Systems*, 1993.

[LP93b] W. Li and K. Pingali. A singular loop transformation framework based on non-singular matrices. *International Journal of Parallel Processing*, 1993.

[LRW91] M. S Lam, E. E. Rothberg, and M. E. Wolf. The cache performance and optimizations of blocked algorithms. In *Proc. 5th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 63–74, April 1991.

[Lu91] L. Lu. A unified framework for systematic loop transformations. In *3rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 28–38, April 1991.

[MFL+92] A. Mohamed, G. Fox, G. Laszewski, M. Parashar, T. Haupt, K. Mills, Y. Li, N. Lin, and N. Yeh. Applications benchmark set for Fortran-D and High Performance Fortran. Technical Report SCCS 327, Syracuse University, 1992.

[ML92] E. P. Markatos and T. J. LeBlanc. Shared-memory versus locality management in shared-memory multiprocessors. In *Proc. of '92 International Conference On Parallel Processing*, August 1992.

[MP87] S. P. Midkiff and D. A. Padua. Compiler algorithms for synchronization. *IEEE Transactions on computers*, C-36:1485–1495, December 1987.

[MSS+88] R. Mirchandaney, J. Saltz, R. Smith, D. Nicol, and K. Crowley. Principles of runtime support for parallel processors. In *Proc. of the 2nd Int. Conf. on Supercomputing*, July 1988.

[Pol88] C. D. Polychronopoulos. *Parallel Programming and Compilers*. Kluwer Academic Publishers, 1988.

[Pol89] C. Polychronopoulos. The Parafrase-2 restructurer. In *Proc. of the Second Workshop on Languages and Compilers for Parallel Computing*, August 1989.

[Por89] A. Porterfield. *Software Methords for Improvement of Cache Performance on Supercomputer Applications*. Ph.D. dissertation, Rice University, May 1989.

[PW86] D. Padua and M. Wolfe. Advanced compiler optimizations for supercomputers. *Communications of ACM*, 29(12):1184–1201, December 1986.

[Ram92] J. Ramanujam. Non-unimodular transformations of nested loops. In *Proc. of Supercomputing*, 1992.

[RP89] A. Rogers and K. Pingali. Process decomposition through locality of reference. In *Proc. of the 1989 SIGPLAN Conference on Programming Language Design and Implementation*, 1989.

[RS91] J. Ramanujam and P. Sadayappan. Compile-time techniques for data distribution in distributed memory machines. *IEEE Transactions on Parallel and Distributed Systems*, 2, October 1991.

[Rue] Roland Ruehl. personal communication.

[Sch86] A. Schrijver. *Theory of Linear and Integer Programming.* John Wiley & Sons, 1986.

[Tse89] P. Tseng. *A Parallelizing Compiler For Distributed Memory Parallel Computers.* Ph.D. dissertation, Carnegie Mellon University, 1989.

[TY86] P. Tang and P.-C. Yew. Processor self-scheduling for multiple nested parallel loops. In *Proc. of '86 International Conference On Parallel Processing*, August 1986.

[WL91a] M. Wolf and M. Lam. A data locality optimizing algorithm. In *Proc. ACM SIGPLAN 91 Conference on Programming Language Design and Implementation*, pages 30–44, June 1991.

[WL91b] M. Wolf and M. Lam. A loop transformation theory and an algorithm to maximize parallelism. *IEEE Transactions on Parallel and Distributed Systems*, October 1991.

[Wol89] M. Wolfe. *Optimizing Supercompilers for Supercomputers.* Pitman Publishing, London, 1989.

[ZC90] H. Zima and B. Chapman. *Supercompilers for Parallel and Vector Computers.* ACM Press Frontier Series, New York, New York, 1990.