Ph.D. Dissertation
Optimization within a Unified Transformation Framework

Wayne Anthony Kelly
Advisor: Professor William W. Pugh
Department of Computer Science
University of Maryland, College Park, MD 20742
wak@cs.umd.edu

December 8, 1996

Contents

1 Introduction **3**

2 The Abstractions **5**
- 2.1 Tuple Relations and Sets 5
- 2.2 Distributing Computation 6
  - 2.2.1 Traditional Abstractions 7
  - 2.2.2 My New Abstraction 8
- 2.3 Ordering Computation 9
  - 2.3.1 Traditional Abstractions 10
  - 2.3.2 My New Abstraction 14
- 2.4 Combining Distribution and Ordering of Computation 20

3 The Algorithms **24**
- 3.1 The Issues 24
- 3.2 Selecting Space Mappings 26
  - 3.2.1 Introduction 26
  - 3.2.2 Candidate Space Mappings 26
  - 3.2.3 Estimating Parallelism 27
  - 3.2.4 False Sharing on Shared Memory Architectures 34
  - 3.2.5 Estimating Communication 34
  - 3.2.6 The Search Problem 36
  - 3.2.7 Alignment 37
  - 3.2.8 An Example: adi 39
- 3.3 Selecting Time Mappings 39
  - 3.3.1 Introduction 39
  - 3.3.2 Estimating Data Locality 40
  - 3.3.3 Incompatible Time Mappings 41
  - 3.3.4 The Search Problem 42
  - 3.3.5 Constant Levels and Offsets 42
- 3.4 The Search Procedure 45
  - 3.4.1 The Simple Search Procedure 45
  - 3.4.2 Pruning Strategies 45
  - 3.4.3 Time-limited Searches 47
  - 3.4.4 Semi-automatic Systems 47
- 3.5 Code Generation 48
  - 3.5.1 Introduction 48
  - 3.5.2 Scanning Multiple Polyhedra 48
  - 3.5.3 Generating SPMD Code 52
  - 3.5.4 Inserting Synchronization 54
  - 3.5.5 Reduction Optimizations 56
  - 3.5.6 Examples 58

4 Experimental Results **61**
4.1 The Implementation 61
4.2 Efficiency 61
4.3 Effectiveness 64
4.4 Robustness 68

5 Related Work **74**
5.1 Distributing Computation 74
5.2 Ordering Computation 75
5.3 Generating Code 78

6 Future Work **80**
6.1 Distributed Memory 80
6.2 Arbitrary Control Flow 80
6.3 Multiple Procedures 80
6.4 Improved Space Mappings 80
6.5 Improved Time Mappings 81
6.6 My Ultimate Goal 81

7 Conclusions **82**

Chapter 1

Introduction

Supercomputers have dramatically increased the amount of computing power available to scientists. These performance improvements have made it feasible to realistically model a much larger class of scientific problems than has previously been possible. Examples of such problems include atmospheric modeling, molecular dynamics and fluid dynamics.

Two basic methods have been used to achieve the very high levels of performance of these new machines. The first method is to make the processor more powerful. This is primarily how the earliest supercomputers such as the Cray 1 achieved their high performance. The problem with this approach is that physical factors such as the speed of light and heat dissipation place an absolute upper bound on obtainable performance. The other method used to increase computing power is to use multiple processors that can work in parallel. Over the last five years, the trend has been towards massively parallel computers, where hundreds or even thousands of processors are used [Ken92].

The problem with using multiple processors is that it is very difficult to write programs that can attain the theoretical peak performance of these machines. The reasons for these difficulties include load balancing problems, memory contention, interprocessor communication bottlenecks, and lack of explicit parallelism. So, while massively parallel machines promise a great deal, the results obtained in practice have been disappointing.

Two approaches have been used to program massively parallel machines. The first approach requires the programmer to write explicitly parallel code, i.e all responsibility is placed on the programmer. The programmer must specify what computations should be performed on which processors, and what communications are required. This approach is clearly not acceptable in the long term, as it places too onerous a burden on the programmer.

The other approach is to have the programmer write normal sequential programs, and then have a compiler convert these sequential programs into explicitly parallel programs. This approach has the advantage that no additional responsibilities are placed on the programmer. The source programs used with this approach also tend to be more portable than explicitly parallel source programs. Of course, the problem with this approach is that converting sequential programs into efficient parallel programs is a very difficult task. So, not surprisingly, existing parallelizing compilers for massively parallel machines often fail to produce satisfactory results [Blu92, SH91].

There are two basic choices to be made when parallelizing a program. First, the computations of the program need to be distributed amongst the set of available processors. Second, the computations on each processor need to be ordered. Approaches taken to the second of these tasks have traditionally been very ad-hoc. Specifically, parallelizing compilers have attempted to parallelize programs and improve their performance by applying a sequence of source to source transformations, such as loop interchange, loop skewing and loop distribution [Wol89a]. Each of these transformations has its own legality checks and transformation rules. These checks and rules make it hard to analyze or predict the effects of compositions of these transformations, without actually performing the transformations and analyzing the resulting code.

I have developed a new framework that unifies the problems of distributing and ordering computation. This framework is based on a simple but powerful mathematical abstraction for representing these decisions. I have also developed algorithms for making these decisions within this framework. These algorithms are extremely extensible, in that the set of transformations considered and the performance estimators used to decide which transformation to apply, are not hard wired into the system. Users are able to modify or write their own performance estimators to reflect the factors which affect performance on their particular architecture. They can also modify the set of transformations considered, so as to obtain the trade-off between efficiency and effectiveness that best suits their individual needs. Conversely, many of the abstractions and algorithms I have developed have applications outside of the framework developed here.

Chapter 2

The Abstractions

There are two basic choices to be made when parallelizing a program. First, the computations of the program need to be distributed amongst the set of available processors. Second, the computations on each processor need to be ordered. In this chapter I describe the abstractions that have traditionally been used to represent these choices. I'll discuss the problems associated with these traditional abstractions and I'll propose new abstractions that are designed to solve some of these problems. Before discussing these new abstractions I will first describe an abstraction called tuple relations upon which the other new abstractions are based.

2.1 Tuple Relations and Sets

An integer k-tuple is simply a point in $\mathcal{Z}^k$. A *tuple relation* is a mapping from tuples to tuples. A single tuple may be mapped to zero, one or more tuples. All the relations I consider map from k-tuples to k'-tuples for some fixed k and k'. I introduce variables corresponding to each of the input and output positions. Relationships between these variables and those corresponding to symbolic constants are represented as logical formulas involving affine equality and inequality constraints. For example, the following relation maps all tuples $[i, j]$ to the tuple $[i, i]$ if i is even and j is between 1 and the symbolic constant n, or i is less than j:

$$\{[i, j] \rightarrow [i, i] \mid (\exists \alpha \text{ s.t. } i = 2\alpha \wedge 1 \leq j \leq n) \vee i < j\}$$

I have helped develop the Omega Library [KMP$^+$95, KPRS95] which is a set of `C++` classes for representing and manipulating tuple relations and sets. The Omega Library can represent any tuple relation or set that can be described using Presburger arithmetic. Presburger formulas are made up from logical quantifiers $(\forall, \exists)$; logical operations $(\vee, \wedge, \Rightarrow, \neg)$; and affine equality and inequality constraints on integer valued variables. Although Presburger arithmetic has extremely high theoretical worst case complexity, $O(2^{2^{2^n}})$, the Omega library has been extensively tuned to handle simple cases, and typically takes only a few milliseconds to produce a result.

Table 2.1 gives a brief description of some of the operations on integer tuple relations and sets that we have implemented. The following are some examples of operations on tuple relations and sets:

If $I = \{[i,j] \mid 1 \leq i \leq n \wedge 1 \leq j \leq i\}$
and $T = \{[i,j] \rightarrow [i+j,i]\}$
then $Range(Restrict\_Domain(T,I)) = \{[x,y] \mid y+1 \leq x \leq 2y \wedge y \leq n\}$

If $C_{22} = \{[i,j,k] \rightarrow [i',j',k'] \mid (i = i' \wedge j' = j+1 \wedge k = k') \vee i' = i+1\}$
$T_{12} = \{[i,j] \rightarrow [i,j,k]\}$
$T_{21} = \{[i,j,k] \rightarrow [i',j'] \mid i' = i+1 \wedge j' = j+1\}$
then $T_{21} \circ C_{22}^* \circ T_{12} = \{[i,j] \rightarrow [i',j'] \mid i < i'\}$

Operation	Definition
Composition	$x \rightarrow z \in F \circ G \Leftrightarrow \exists y$ s.t. $x \rightarrow y \in F \wedge y \rightarrow z \in G$
Inverse	$x \rightarrow y \in F^{-1} \Leftrightarrow y \rightarrow x \in F$
Union	$x \rightarrow y \in F \cup G \Leftrightarrow x \rightarrow y \in F \vee x \rightarrow y \in G$
Intersection	$x \rightarrow y \in F \cap G \Leftrightarrow x \rightarrow y \in F \wedge x \rightarrow y \in G$
Restrict Domain	$x \rightarrow y \in Restrict\_Domain(F,S) \Leftrightarrow x \rightarrow y \in F \wedge x \in S$
Restrict Range	$x \rightarrow y \in Restrict\_Range(F,S) \Leftrightarrow x \rightarrow y \in F \wedge y \in S$
Domain	$x \in Domain(F) \Leftrightarrow \exists y$ s.t. $x \rightarrow y \in F$
Range	$y \in Range(F) \Leftrightarrow \exists x$ s.t. $x \rightarrow y \in F$
Projection	$x \in \Pi_{1,\ldots,v}\ S \Leftrightarrow \lvert x \rvert = v \wedge \exists y$ s.t. $xy \in S$
Transitive Closure	$x \rightarrow y \in F^* \Leftrightarrow x \rightarrow y \in F \vee \exists z$ s.t. $x \rightarrow z \in F \wedge z \rightarrow y \in F^*$
Is satisfiable ?	$Is\_Satisfiable(S) \Leftrightarrow \exists x \in S$

Table 2.1: Operations on tuple sets and relations

Tuple relations and sets and their associated operations have an enormous number of applications other than those described in this dissertation. Some of the applications to which they have already been applied include:

- Data dependence analysis
- Program transformations
- Redundant synchronization removal
- Code generation
- Induction variable analysis
- Safety and Coverage analysis for finite state systems

2.2 Distributing Computation

The units of computation that are distributed amongst the processors of a multiprocessor machine may be either entire tasks or individual iterations of loops. Task-level parallelism works best on distributed systems such as networks of workstations because it generally involves a lesser amount of inter-processor communication, which is particularly expensive on such systems. Loop-level parallelism works best on tightly-coupled massively parallel machines because it is more likely to provide sufficient parallelism to make use of all available processors. Such parallelism is most likely to be found in programs for scientific applications which consist mainly of `for` loops and assignment statements involving arrays. The scientific community is consequentially the largest group of users of massively parallel computers today. This thesis deals exclusively with loop-level parallelism.

```
    doall i = 1 to n
1:      a(i) = i
    doall i = 1 to n
2:      b(i) = a(i-1)
```

Figure 2.1: Why `doall` loops are insufficient for NUMA machines

2.2.1 Traditional Abstractions

Traditionally, the distribution of computations is specified via either `doall` and `doacross` loops or via a data distribution for each array.

Doall and Doacross Loops

One way to specify the parallelism available in a program written in a sequential language is to annotate the do loops in that program as being either `doall`, `doacross` or `dosequential` [ZC91].

A `doall` loop indicates that all of the iterations of that loop are independent and therefore can be executed in any order (including in parallel). A `doall` loop does not indicate which iterations should be performed on which processors. The mapping of iterations to processors is determined either statically by some latter phase of the compilation/linking process or dynamically by the runtime system.

A `doacross` loop indicates that it is possible to partially overlap the execution of the loops iterations. There are generally some ordering constraints on the loop iterations, but not so many that the iterations must be executed entirely sequentially. These ordering constraints are enforced by explicit synchronization statements that must be inserted in the body of such loops.

A `dosequential` loop indicates that it is not possible to overlap the execution of any of the loop iterations. In practice, a loop could be marked as being `dosequential` even if it could be marked as being `doall` or `doacross`. This is generally done when it is estimated that the overhead associated with executing the loop in parallel is higher than the potential savings due to parallelism.

In the case of *Uniform Memory Access* (UMA) [AG94] machines, the fact that `doall` and `doacross` loops cannot control the mapping of iterations onto processors is of no consequence since the choice of mapping can not affect performance (provided some care is taken to ensure load balancing). However, in the case of *Non-Uniform Memory Access* (NUMA) machines [AG94], the choice of mapping can have a substantial affect on performance. For NUMA machines, the choice of mapping will determine which data accesses will be satisfied locally and which data accesses will require communication with other processors.

Consider for example the program shown in Figure 2.1. If for both loops, iteration i is assigned to processor i, then inter-processor communication will be required. If, however, iteration i of the first loop is assigned to processor i and iteration i of the second loop is assigned to processor $i-1$, then no inter-processor communication will be required. In most modern NUMA architectures, the time required to access data from another processor is orders of magnitudes longer than the time required to access data that is on processor.

Data Distributions

In order to control the mapping of iterations onto processors, a different method of representing parallelism in programs is required. A common approach is the so called *data parallel* paradigm. In this model, parallelism is achieved by performing the same operation on different data elements at the same time. The distribution of computation to processors is specified implicitly

```
PROCESSORS procs(32)
TEMPLATE T(1024)
ALIGN a(i) with T(i)
ALIGN b(i) with T(i-1)
DISTRIBUTE T(block) onto procs
```

Figure 2.2: Data distribution

```
do i = 1 to n
    s = 0.
    do j = 1 to i - 1
        s = s + a(n-i+1,n-j+1) * x(n-j+1)
    x(n-i+1) = (a(n-i+1,n+1) - s) / a(n-i+1,n-i+1)
```

Figure 2.3: Reduction example where a data distribution is not optimal

in such systems via data distribution annotations [HKT91, For92] and the owner-computes rule [CK88]. A data distribution is an affine mapping from the elements of an array to a virtual processor array. The owner-computes rule states that each iteration should be performed on the virtual processor that "owns" the array element being written. Virtual processors are folded onto physical processors in either a block, cyclic or block-cyclic manner. Data distributions may be either static or dynamic. A static distribution means that the same distribution is used throughout the entire execution of the program and a dynamic distribution means that different distributions will be used during various phases of the program's execution. Figure 2.2 shows an example of a data distribution (and corresponding alignment statement) for the program in Figure 2.1.

Data distributions are an appropriate abstraction for machines with logically distributed memories. Such machines require the programmer or compiler to explicitly manage the storage and transfer of data. On machines with logically shared memories, however, data distributions may not be the most appropriate abstraction. On such machines, data is automatically transferred from one processor's memory to another's as required. In this context, data distributions are simply an indirect way to specify a computation distribution and as such are unnecessarily restrictive. Data distributions, when used with the owner-computes rule, do not allow different iterations of the same statement that write to the same array element to be executed on different processors. Such computation distributions are often desirable, especially for statements that are performing a reduction. A reduction is a set of operations that compute a scalar value from an array[ZC91]. For example, in the program shown in Figure 2.3, it would be best to assign iteration $[i, j]$ to virtual processor j in order to perform the reduction in parallel.

The other drawback with data distributions is that either a static data distribution must be chosen, or the program must be divided into phases between which data redistributions will be performed. If the compiler is not going to consider restructuring the program in a major way then this is not a major problem. If, however, the compiler is trying to both automatically select data distributions and restructure the program, then deciding how to divide a program into phases can be particularly difficult as it also involves making decisions about the best order in which to perform the computations.

2.2.2 My New Abstraction

I use a relatively new abstraction called a space mapping [Fea94, AL93b] to directly map the computations of a program to a virtual processor array. I associate a tuple relation S_p, with

$S_1: \{[i] \to [i]\}$
$S_2: \{[i] \to [i-1]\}$

```
block_size = max(0,ceiling((n+1)/nprocs))
lb = 0+my_id*block_size
ub = min(n,lb+block_size-1)
do t = max(1,lb) to min(n,ub)
    a(t) = t
do t = max(0,lb) to min(n-1,ub)
    b(t+1) = a(t)
```

Figure 2.4: Space mappings and corresponding SPMD code.

each statement p. It specifies that each iteration i of statement p will be executed on virtual processor $S_p(i)$. The input arity of S_p is equal to the number of loops surrounding statement p and the output arity is equal to the dimensionality of the virtual processor array. I will only consider space mappings that are functions (i.e., each iteration i of statement p is mapped to a unique virtual processor $S_p(i)$). As with data distributions, the virtual processor array is folded onto the physical processor array in either a blocked, cyclic, or block-cyclic fashion.

By directly mapping iterations to virtual processors (rather than using data decompositions), I am able to represent a wider range of computation distributions and hence have more freedom in trying to minimize inter-processor communication. In addition, I can represent dynamic data distributions without having to partition the program into phases between which redistribution will occur. Figure 2.4 shows an example of space mappings that could be used for the program in Figure 2.1, together with the SPMD code that would result from applying these space mappings with a blocked distribution. In the (SPMD) Single Program Multiple Data programming model, all processors execute an identical copy of the program. The key to SPMD programming is that each processor can access a variable (commonly called `my_proc`) that allows the program running on that processor to know which processor it is running on. This makes it possible to explicitly control which iterations are performed on which processors.

A separate space mapping is associated with each statement, however, what constitutes a statement may vary. In this dissertation, I generally assume that each assignment statement in the original program is a separate statement. However, an entire basic block, or any other well nested section of code could be considered a single statement. At the other extreme, an assignment statement could be decomposed into the machine code instructions required to execute it, each of which could be considered a separate statement. Obviously, there is a trade-off between the flexibility of mapping finer-grained statements, and the extra time required to consider a greater number of possibilities.

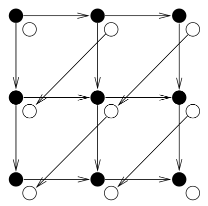
2.3 Ordering Computation

Program transformations can generally be classified into one of three classes:

Reordering transformations, where the same set of computations are performed but in a different order. Examples of this class include loop interchange, statement reordering and loop distribution.

Storage modifying transformations, where the same computations are performed in the same order but with the intermediate results being stored in different locations. Examples of this class include array and scalar expansion and privatization.

```
for i = 1,n do
  for j = 1,n do
    a(i,j)=a(i-1,j)+a(i,j-1)
    b(i,j)=b(i,j)+b(i-1,j+1)
```



Loop distribution

```
for i = 1,n do
  for j = 1,n do
    a(i,j)=a(i-1,j)+a(i,j-1)
for i = 1,n do
  for j = 1,n do
    b(i,j)=b(i,j)+b(i-1,j+1)
```

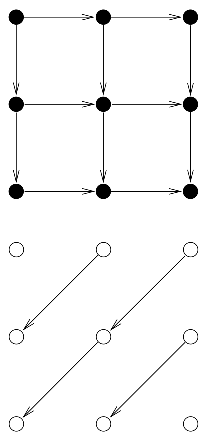


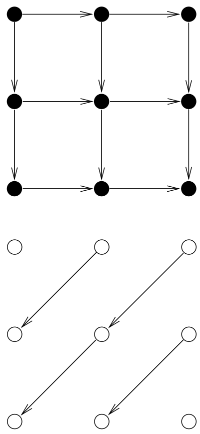
Figure 2.5: A sequence of traditional transformations

Algorithm modifying transformations, where a different set of computations are performed to compute the same final result. Examples of this class include induction variable replacement and reordering reduction operations.

In this thesis I deal only with reordering transformations; however, I assume that some other transformations such as induction variable replacement, and scalar/array expansion will have been performed before the transformations described here. I also assume that other transformations such as code hoisting and strength reduction will be performed after the transformations described here.

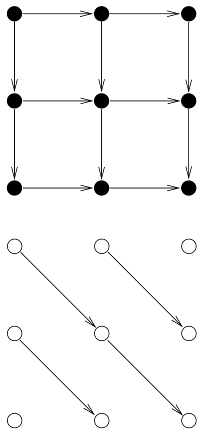
2.3.1 Traditional Abstractions

Optimizing compilers reorder iterations of statements to improve instruction scheduling, register use, cache utilization, and to expose parallelism. Many different reordering transformations have been developed and studied, such as loop interchange, loop distribution, skewing, tiling, index set splitting and statement reordering [AK87, Wol89c, Wol90, CK92]. Figures 2.5 through 2.8 show an example of such a sequence of transformations (black dots correspond to iterations of the first statement and white dots correspond to iterations of the second statement). Each of these transformations has its own special legality checks and transformation rules. These checks and rules make it hard to analyze or predict the effects of compositions of these transformations, without actually performing the transformations and analyzing the resulting code.



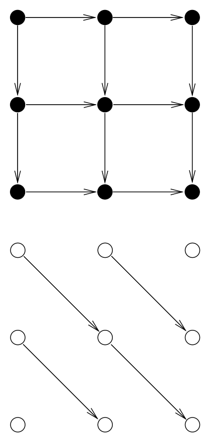
```
for i = 1,n do
  for j = 1,n do
    a(i,j)=a(i-1,j)+a(i,j-1)
for i = 1,n do
  for j = 1,n do
    b(i,j)=b(i,j)+b(i-1,j+1)
```

Loop reversal (second loop nest)



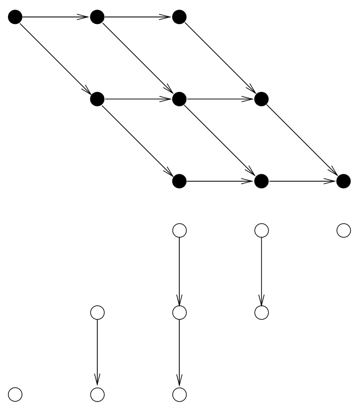
```
for i = 1,n do
  for j = 1,n do
    a(i,j)=a(i-1,j)+a(i,j-1)
for i = 1,n do
  for j = -n,-1 do
    b(i,-j)=b(i,-j)+b(i-1,-j+1)
```

Figure 2.6: A sequence of traditional transformations, continued



```
for i = 1,n do
  for j = 1,n do
    a(i,j)=a(i-1,j)+a(i,j-1)
for i = 1,n do
  for j = -n,-1 do
    b(i,-j)=b(i,-j)+b(i-1,-j+1)
```

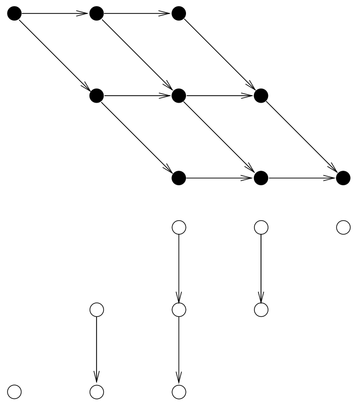
Loop skewing (both loop nests)



```
for i = 1,n do
  for j = 1+i,n+i do
    a(i,j-i)=a(i-1,j-i)+a(i,j-i-1)
for i = 1,n do
  for j = -n-i,-1-i do
    b(i,-j-i)=b(i,-j-i)+b(i-1,-j-i+1)
```

Figure 2.7: A sequence of traditional transformations, continued

```
for i = 1,n do
  for j = 1+i,n+i do
    a(i,j-i)=a(i-1,j-i)+a(i,j-i-1)
for i = 1,n do
  for j = -n-i,-1-i do
    b(i,-j-i)=b(i,-j-i)+b(i-1,-j-i+1)
```



Loop interchange (both loop nests)

```
for j = 2,2n do
  forall i = max(1,j-n),min(n,j-1) do
    a(i,j-i)=a(i-1,j-i)+a(i,j-i-1)
forall j = -2n,-2 do
   for i = max(1,-n-j),min(n,-1,-j) do
    b(i,-j-i)=b(i,-j-i)+b(i-1,-j-i+1)
```

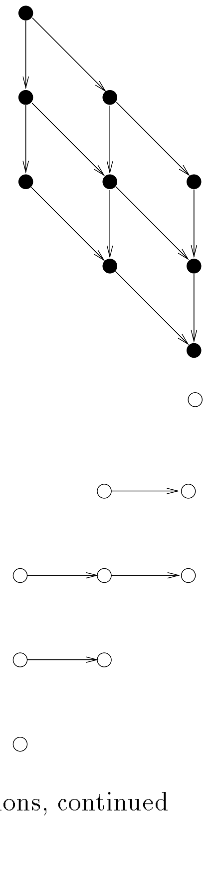


Figure 2.8: A sequence of traditional transformations, continued

Program:

```
      do i = 1 to n
1       s(i) = 0
        do j = 1 to i-1
2         s(i) = s(i) + a(j,i)*b(j)
3       b(i) = b(i) - s(i)
```

Iteration space:

$$\begin{array}{llll} I_1: & \{ & [\,i\,] & \mid 1 \le i \le n \;\} \\ I_2: & \{ & [\,i,\ j\,] & \mid 1 \le i \le n \wedge 1 \le j \le i-1 \;\} \\ I_3: & \{ & [\,i\,] & \mid 1 \le i \le n \;\} \end{array}$$

Figure 2.9: Program and associated iteration space

Unimodular transformations [Ban90, WL91] go some way towards solving this problem. Unimodular transformations are able to describe any transformation that can be obtained by composing loop interchange, loop skewing, and loop reversal. Such a transformation is described by a unimodular linear mapping from the original iteration space to a new iteration space. For example, loop interchange in a doubly nested loop maps iteration $[i, j]$ to iteration $[j, i]$. This transformation can be described using a unimodular matrix:

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

Unfortunately, unimodular transformations are limited in two ways: they can only be applied to perfectly nested loops, and all statements in the loop nest are transformed in the same way. They therefore cannot represent some important transformations such as loop fusion, loop distribution and statement reordering [Wol89c].

2.3.2 My New Abstraction

Before describing the abstraction I use to represent the ordering of computations, I first need to define some terms.

Iteration spaces

Each statement p has associated with it an iteration space I_p, which is a subspace of $\mathcal{Z}^{n_p}$ (where n_p is the number of loops nested around p). A statement's iteration space is the set of iterations for which that statement will be executed.

More formally $[x_1, \ldots, x_n] \in I_p$ if and only if:

$$\forall j,\ 1 \le j \le n \Rightarrow L_j(x_1, \ldots, x_{j-1}, \vec{s}) \le x_j \le U_j(x_1, \ldots, x_{j-1}, \vec{s})$$

where L_j and U_j are functions representing the lower and upper bounds respectively of the j^{th} loop around p, and $\vec{s}$ is a vector of symbolic constants.

For example, Figure 2.9 shows a program and its associated iteration space.

Time mappings

I have developed a new abstraction called time mappings for specifying the order in which iterations should be executed. I associate a separate time mapping, T_p, with each statement p. A time mapping is a tuple relation that maps each point (or iteration) in the original iteration space I_p to a unique point in a new iteration space I'. The set of time mappings as a whole, Time Mapping:

$$\begin{array}{lllllll} T_1: & \{[\,i\,] & \rightarrow [0, & i & & &]\,\} \\ T_2: & \{[\,i,\ j\,] & \rightarrow [1, & j, & 0, & i &]\,\} \\ T_3: & \{[\,i\,] & \rightarrow [1, & i-1, & 1 & &]\,\} \end{array}$$

Transformed program:

```
      do i = 1 to n
1       s(i) = 0
      do t = 1 to n-1
        do i = t+1 to n
2         s(i) = s(i) + a(t,i)*b(t)
3       b(t+1) = b(t+1) - s(t+1)
```

Figure 2.10: Time mapping and associated transformed program

describe a 1-1 mapping from the set of original iteration spaces to a single new iteration space. That is:

$$\forall p, q, i, j \ T_p(i) = T_q(j) \Rightarrow p = q \wedge i = j \tag{2.1}$$

The iterations will be executed in lexicographic order based on their coordinates in the new iteration space. So, specifying a time mapping is effectively specifying a reordering of the iterations.

Figure 2.10 shows an example of a set of time mappings for the program in Figure 2.9. This time mapping maps iteration $[5,7]$ in the original iteration space of statement 2 to iteration $[1,7,0,5]$ in the new iteration space and maps iteration $[6]$ in the original iteration space of statement 1 to iteration $[0,6]$. Iteration $[0,6]$ is lexicographically less than iteration $[1,7,0,5]$, so in the transformed code, the iteration originally referred to as iteration $[6]$ of statement 1 will be executed before the iteration originally referred to as iteration $[5,7]$ of statement 2. So, as this example demonstrates, by mapping to a single new iteration space, I can specify the relative order in which iterations should be executed, even for iterations belonging to different statements.

The transformed code will always contain the same elementary statements as the original code, but will contain different loop structures. The new loop structures execute in lexicographic order, all and only those iterations in the new iteration space. In Section 3.5 I will describe an algorithm I have developed for generating transformed code given a set of time mappings.

Time mappings can be considered a generalization of Unimodular transformations. Time mappings are more general in the following respects:

- I associate a separate time mapping with each statement, whereas unimodular transformations require a single unimodular matrix to be used to transform all statements in the body of a perfectly nested loop.
- Unimodular transformations only allow linear mapping to be specified. Time mappings allow any mapping to be specified that can be expressed in Presburger arithmetic.
  - In particular, time mappings can easily represent affine mappings. An affine expression is equivalent to a linear expression plus a constant term. For example, $i+j$ is linear, $i+j+1$ is affine. There is no advantage in using affine mappings rather than linear mappings if only one mapping is being used to transform all statements (as is the case for unimodular transformations), since a constant term will simply translate all points in the new iteration space and not affect their relative order. However, if more than one mapping is specified (as is the case for time mappings), then using different constant terms in different mappings can affect the relative order of the

Code adapted from OLDA in Perfect club (TI)

```
      do 20 mp = 1, np
        do 20 mq = 1, mp
          do 20 mi = 1, morb
10          xrsiq(mi,mq)+=xrspq((mp-1)*mp/2+mq)*v(mp,mi)
20          xrsiq(mi,mp)+=xrspq((mp-1)*mp/2+mq)*v(mq,mi)
```

Time Mapping (to expose parallelism)

$$
\begin{array}{llllllllll}
T_{10}: & \{\, [\, mp, & mq, & mi \,] & \rightarrow [\, mi, & mq, & mp, & 0] \} \\
T_{20}: & \{\, [\, mp, & mq, & mi \,] & \rightarrow [\, mi, & mp, & mq, & 1] \}
\end{array}
$$

Transformed code

```
     do 20 mi = 1,morb  /* parallel */
       do 20 t2 = 1,np  /* parallel */
         do 10 t3 = 1,t2-1
10         xrsiq(mi,t2)+=xrspq((t3-1)*t3/2+t2)*v(t3,mi)
         xrsiq(mi,t2)+=xrspq((t2-1)*t2/2+t2)*v(t2,mi)
         xrsiq(mi,t2)+=xrspq((t2-1)*t2/2+t2)*v(t2,mi)
         do 20 t3 = t2+1,np
20         xrsiq(mi,t2)+=xrspq((t2-1)*t2/2+t3)*v(t3,mi)
```

Figure 2.11: OLDA with time mappings, and resulting transformation

points in the new iteration space (as was demonstrated in the previous example). Affine mappings allow me to represent a number of traditional transformations in addition to those representable using unimodular transformations, including statement reordering, loop distribution, loop fusion, and loop alignment.

– Time mappings can also represent pseudo-affine mappings. A pseudo-affine expression is an expression that involves affine expressions and integer division and modulo operations (provided the denominator is a known integer constant). Pseudo-affine mappings allow me to represent a number of additional transformations including strip mining (or tiling).

– Time mappings allow different affine (or pseudo-affine) mappings to be specified for different parts of the iteration space of a single statement. This allows me to represent index set splitting transformations.

Time mappings can represent not only all of the traditional transformations mentioned above, but also any sequence of these traditional transformations. I have therefore simplified the problem of reordering the iterations of a program to finding a time mapping for each statement. This simple and elegant representation makes it much easier to reason about reordering transformations.

Some more examples of time mappings are given in Figures 2.11 through 2.14.

Legality

Not all time mappings correspond to legal transformations, so I need a way to distinguish between legal and illegal time mappings. A time mapping is legal if the transformation it describes preserves the semantics of the original code. This is true if the new ordering of the iterations respects all of the dependences in the original code.

I assume that the programs contain only static control flow, so control dependences do not have to be considered. Three types of data dependences can occur; flow, output and anti. It

LU Decomposition without pivoting

```
    do 20 k = 1, n
      do 10 i = k+1, n
10      a(i,k) = a(i,k) / a(k,k)
        do 20 j = k+1, n
20        a(i,j) = a(i,j) - a(i,k) * a(k,j)
```

Time Mapping (for locality)

$$
\begin{array}{l}
T_{10} : \{[k, i\ \ ] \rightarrow [64((k-1)\ div\ 64)+1, 64(i\ div\ 64), k, k, i]\} \\
T_{20} : \{[k, i, j] \rightarrow [64((k-1)\ div\ 64)+1, 64(i\ div\ 64), j, k, i]\}
\end{array}
$$

Transformed code

```
     do 30 kB = 1, n-1, 64
       do 30 iB = kB-1, n, 64
         do 20 kj = kB, min(kB+63, n)
           do 10 k = kB, kj-1
             do 10 i = max(k+1,iB), min(iB+63,n)
10             a(i,kj)=a(i,kj)-a(i,k)*a(k,kj)
           do 20 i = max(iB,kj+1), min(iB+63,n)
20           a(i,kj)=a(i,kj)/a(kj,kj)
         do 30 kj = kB+64, n
           do 30 k = kB to kB+64
             do 30 i = max(k+1,iB), min(iB+63,n)
30             a(i,kj)=a(i,kj)-a(i,k)*a(k,kj)
```

Figure 2.12: LU with time mappings, and resulting transformation

Code adapted from CHOSOL in the Perfect club (SD)

```
      do 30 i=2,n
10      sum(i) = 0.
        do 20 j=1,i-1
20        sum(i) = sum(i) + a(j,i)*b(j)
30      b(i) = b(i) - sum(i)
```

Time Mapping (to expose parallelism)

$$
\begin{array}{lllllll}
T_{10} : & \{ [\ i\ ] & & \rightarrow [\ 0, & i, & 0, & 0\ ]\ \} \\
T_{20} : & \{ [\ i, & j\ ] & \rightarrow [\ 1, & j, & 0, & i\ ]\ \} \\
T_{30} : & \{ [\ i\ ] & & \rightarrow [\ 1, & i-1, & 1, & 0\ ]\ \}
\end{array}
$$

Transformed code

```
     do 10 i = 2,n  /* parallel */
10     sum(i) = 0.
     do 30 t2 = 1, n-1
       do 20 i = t2+1,n  /* parallel */
20       sum(i) = sum(i) + a(t2,i)*b(t2)
30     b(t2+1) = b(t2+1) - sum(t2+1)
```

Figure 2.13: CHOSOL with time mappings, and resulting transformation

Banded SYR2K adapted from BLAS

```
      do 10 i = 1, n
       do 10 j = i, min(i+2*b-2,n)
        do 10 k = max(i-b+1,j-b+1,1),min(i+b-1,j+b-1,n)
10        C(i,j-i+1) = C(i,j-i+1) +
   $                   alpha*A(k,i-k+b)*B(k,j-k+b) +
   $                   alpha*A(k,j-k+b)*B(k,i-k+b)
```

Time Mapping (for locality and to expose parallelism)

$$T_{10}: \ \{ [\ i, \ \ j, \ \ k] \ \rightarrow [\ j-i+1, \ \ k-j, \ \ k\ ] \}$$

Transformed code

```
      do 10 t1 = 1, min(n,2*b-1) /* parallel */
        do 10 t2 = max(1-b,1-n), min(b-t1, n-t1)
          do 10 k = max(1,t1+t2), min(n+t2,n)  /* parallel */
10          C(-t1-t2+k+1,t1) = C(-t1-t2+k+1,t1) +
   $                           alpha*A(k,-t1-t2+b+1)*B(k,-t2+b) +
   $                           alpha*A(k,-t2+b)*B(k,-t1-t2+b+1)
```

Figure 2.14: SYR2K with time mappings, and resulting transformation

is generally possible (though not always desirable) to remove output and anti dependences by performing array expansion. Two possible approaches can be taken:

- Determine legality based on only flow dependences and then remove any output and anti dependences that are not respected, by performing array expansion on the arrays involved in those dependences.
- Determine a priori, which arrays will be expanded and then determine legality based on flow dependences and whichever output and anti dependences remain.

Most of the previous work on program transformations uses data dependence directions or distances to summarize dependences between array references. These abstractions are sufficient for simple transformations such as unimodular transformations, but they are not precise enough to determine the legality of loop fusion and a number of other transformations without actually applying the transformation and re-evaluating dependences. Since my framework includes loop fusion, they are not sufficient for my purposes either. I evaluate and represent dependences exactly using integer tuple relations. If there is a data dependence from $s_p[i]$ (i.e., iteration i of statement p) to $s_q[j]$ then the tuple relation D_{pq} representing the dependences from p to q will map tuple i to tuple j.

The legality requirement is then simply:

$$\forall i, j, p, q, Sym \quad i \rightarrow j \in D_{pq} \ \Rightarrow T_p(i) \prec T_q(j) \tag{2.2}$$

where $\prec$ means lexicographically precedes and Sym is the set of symbolic constants in D_{pq}. Intuitively, if there is a data dependence from $s_p[i]$ to $s_q[j]$ then iteration i must be executed before iteration j in the transformed program. To be well-formed, the time mappings must also be 1-1 (see Equation 2.1).

Representing Traditional Transformations

In this section I demonstrate how time mappings can be used to represent all transformations that can be obtained by applying any sequence of the following traditional transformations:

$\texttt{Orig}(\text{S}, [i_1, ..., i_k] \to [f_1, ..., f_a])$

    $\texttt{case}\ S\ \texttt{of}$

        "for $i_{k+1} = \ldots$ to $\ldots$ do S_1":

            $\texttt{return Orig}(S_1,\ [i_1, \ldots, i_k, i_{k+1}] \to [f_1, \ldots, f_a, i_{k+1}])$

        "$S_1; S_2; \ldots; S_m$":

            $\texttt{return} \bigcup_{p=1}^{m} \texttt{Orig}(S_p,\ [i_1, \ldots, i_k] \to [f_1, \ldots, f_a, p])$

        "assignment #p":

            $\texttt{return}\ T_p : \{\ [i_1, \ldots, i_k] \to [f_1, \ldots, f_a]\ \}$

Figure 2.15: Computes mapping that corresponds to the original execution order

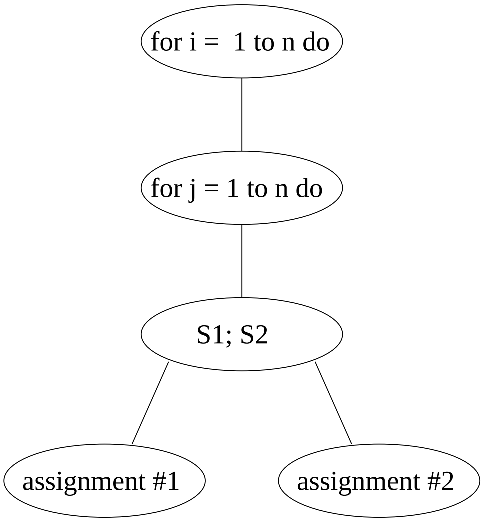


Figure 2.16: Abstract syntax tree

- Loop Distribution
- Statement Reordering
- Loop Fusion
- Loop Interchange
- Loop Skewing
- Loop Reversal.
- Strip-mining
- Index Set Splitting

I will describe how to construct time mappings to represent traditional transformations by describing how to modify time mappings that correspond to the original execution order of programs. The time mapping that corresponds to the original execution order of a program can be constructed by a recursive descent of the abstract syntax tree (AST). Nodes in the AST have three forms: loops, statement lists and assignment statements. The function `Orig` (see Figure 2.15), when called with arguments of S and $[\,] \to [\,]$, returns a time mapping that corresponds to the original execution order for each of the assignment statements in S.

For example, the program in Figure 2.5 has the AST shown in Figure 2.16 and the time mapping that corresponds to the original execution order is:

$$T_1 : \{[i,j] \rightarrow [i,j,0]\ \}$$
$$T_2 : \{[i,j] \rightarrow [i,j,1]\ \}$$

The general form of these time mappings is:

$$T_p : [i_p^1, \ldots, i_p^{m_p}] \rightarrow [f_p^1, \ldots, f_p^{n_p}]$$

The f_p^j expressions are referred to as mapping components and are classified as being either syntactic components (always an integer constant) or loop components (a linear function of the loop variables of that statement). $\texttt{syntactic}(f_p^j)$ is a boolean function which is true iff f_p^j is a syntactic component. $\texttt{loop}(f_p^j)$ is true iff f_p^j is a loop component. The common syntactic level of two statements s_p and s_q is defined as:

$$\texttt{csl}(s_p, s_q) \equiv \min\{j-1 \mid 1 \le j \wedge f_p^j \ne f_q^j \wedge \texttt{syntactic}(f_p^j) \wedge \texttt{syntactic}(f_q^j)\}$$

Intuitively, the common syntactic level of two statements is the deepest loop which surrounds both statements. Figures 2.17 and 2.18 describe how to construct time mappings to represent traditional transformations by modifying the time mappings I have just described. Since these rules can be applied repeatedly, I can represent not only standard transformations but also any sequence of standard transformations.

Figures 2.19 through 2.20 show how these rules are used to derive the time mapping:

$$\begin{array}{ll} T_1: & \{[i,\ \ j] \rightarrow [1,\ \ i+j,\ \ i,\ \ 1]\} \\ T_2: & \{[i,\ \ j] \rightarrow [2,\ \ i-j,\ \ i,\ \ 2]\} \end{array}$$

which is equivalent to the sequence of traditional transformations shown in Figures 2.5 through 2.8.

2.4 Combining Distribution and Ordering of Computation

The abstractions used for reordering computation can be used in isolation to specify a new total ordering of the iterations, or they can be used together with the abstractions for distributing computation to specify a new partial ordering of the iterations. When used with space mappings (or any other abstraction for specifying the distribution of computation), time mappings are used only to specify the relative execution order for iterations on the same physical processor. That is, if two iterations are executed on different physical processors, then the time mappings do not specify the relative order in which these iterations should be executed. If data dependences exist between iterations on different physical processors then synchronization statements will be used to ensure that those iterations are executed in the appropriate order. These ordering constraints together with the ordering constraints implied by the time mappings will generally only specify a partial ordering of the iterations. The absence of a total ordering is what allows parallelism to be exploited.

Distribution: distribute the loop at depth L over the set of statements D, with statement p going into ${r_p}^{\text{th}}$ loop.

Requirements: $\forall p, q \;\; p \in D \wedge q \in D \Rightarrow \texttt{loop}(f_p^L) \wedge L \leq \texttt{csl}(p, q)$

Transformation: $\forall p \in D$, replace T_p by $[f_p^1, \ldots, f_p^{(L-1)}, r_p, f_p^L, \ldots, f_p^n]$

Statement Reordering: reorder the set of statements D, at level L so that the new position of statement p is r_p.

Requirements: $\forall p, q \;\; p \in D \wedge q \in D \Rightarrow \texttt{syntactic}(f_p^L) \wedge L \leq csl(p, q) + 1 \wedge (L \leq csl(p, q) \Leftrightarrow r_p = r_q)$

Transformation: $\forall p \in D$, replace T_p by $[f_p^1, \ldots, f_p^{(L-1)}, r_p, f_p^{(L+1)}, \ldots, f_p^n]$

Fusion: fuse the loops at level L for the set of statements D, with statement p going into the ${r_p}^{\text{th}}$ loop.

Requirements: $\forall p, q \; p \in D \wedge q \in D \Rightarrow \texttt{syntactic}(f_p^{(L-1)}) \wedge \texttt{loop}(f_p^L) \wedge L - 2 \leq csl(p, q) + 2 \wedge (L - 2 < csl(p, q) + 2 \Rightarrow r_p = r_q)$

Transformation: $\forall p \in D$, replace T_p by
$[f_p^1, \ldots, f_p^{(L-2)}, r_p, f_p^{(L)}, f_p^{(L-1)}, f_p^{(L+1)}, \ldots, f_p^n]$

Unimodular Transformation: Apply a $k \times k$ unimodular transformation U to a perfectly nested loop containing the set of statements D, at depth L. Note: Unimodular transformations include loop interchange, skewing and reversal [Ban90, WL91].

Requirements: $\forall i, p, q \;\; p \in D \wedge q \in D \wedge L \leq i \leq L + k - 1 \Rightarrow \texttt{loop}(f_p^i) \wedge L + k - 1 \leq csl(p, q))$

Transformation: $\forall p \in D$, replace T_p by
$[f_p^1, \ldots, f_p^{(L-1)}, U[f_p^L, \ldots, f_p^{L+k-1}]^\top, f_p^{L+k}, \ldots, f_p^n]$

Strip-mining: strip-mine the level L loop for the set of statements D, with block size B

Requirements: $\forall p, q \;\; p \in D \wedge q \in D \Rightarrow \texttt{loop}(f_p^L) \wedge L \leq csl(p, q)) \wedge$ B is a known integer constant

Transformation: $\forall p \in D$, replace T_p by
$[f_p^1, \ldots, f_p^{(L-1)}, B(f_p^{(L)} \text{ div } B), f_p^{(L)}, \ldots, f_p^n]$

Figure 2.17: Representing traditional transformations

Index Set Splitting: split the iteration spaces of the set of statements D, using condition C

Requirements: C is affine expression of constants and indexes common to the set of statements D.

Transformation: $\forall p \in D$, replace T_p by $(T_p \mid C) \;\cup\; (T_p \mid \neg C)$

Figure 2.18: Representing traditional transformations, continued

Original Time Mappings:

$$
\begin{array}{l}
T_1 : \{[i, j] \rightarrow [i, j, 1]\} \\
T_2 : \{[i, j] \rightarrow [i, j, 2]\} \\
f_1^1 = i,\ f_1^2 = j,\ f_1^3 = 1,\ f_2^1 = i,\ f_2^2 = j,\ f_2^3 = 2 \\
\texttt{loop}(i) = True,\ \texttt{loop}(j) = True,\ \texttt{loop}(1) = False,\ \texttt{loop}(2) = False \\
csl(1, 2) = 2
\end{array}
$$

Loop distribution:

$$L = 1,\ D = \{1, 2\},\ r_1 = 1,\ r_2 = 2$$

Requirements:

$\forall p, q\ p \in \{1, 2\} \wedge q \in \{1, 2\} \Rightarrow \texttt{loop}(f_p^1) \wedge 1 \leq csl(p, q)$

Transformation:

$$
\begin{array}{lllllll}
T_1 : & \{[i, & j] \rightarrow [1, & i, & j, & 1]\} \\
T_2 : & \{[i, & j] \rightarrow [2, & i, & j, & 2]\}
\end{array}
$$

Loop reversal second loop (Unimodular Transformation):

$$k = 1,\ U = (-1),\ D = \{2\},\ L = 3$$

Requirements:

$\forall i, p, q\ p \in \{2\} \wedge q \in \{2\} \wedge 3 \leq i \leq 3 \Rightarrow \texttt{loop}(f_p^i) \wedge 3 \leq csl(p, q)$

Transformation:

$$
\begin{array}{lllllll}
T_1 : & \{[i, & j] \rightarrow [1, & i, & j, & 1]\} \\
T_2 : & \{[i, & j] \rightarrow [2, & i, & -j, & 2]\}
\end{array}
$$

Loop skewing first loop (Unimodular Transformation):

$$k = 2,\ U = \begin{pmatrix} 1 & 0 \\ 1 & 1 \end{pmatrix},\ D = \{1\},\ L = 2$$

Requirements:

$\forall i, p, q\ p \in \{1\} \wedge q \in \{1\} \wedge 2 \leq i \leq 3 \Rightarrow \texttt{loop}(f_p^i) \wedge 3 \leq csl(p, q)$

Transformation:

$$
\begin{array}{lllllll}
T_1 : & \{[i, & j] \rightarrow [1, & i, & i + j, & 1]\} \\
T_2 : & \{[i, & j] \rightarrow [2, & i, & -j, & 2]\}
\end{array}
$$

Figure 2.19: Deriving time mappings

Loop skewing second loop (Unimodular Transformation):

$$k = 2,\ U = \begin{pmatrix} 1 & 0 \\ 1 & 1 \end{pmatrix},\ D = \{2\},\ L = 2$$

Requirements:

$\forall i, p, q\ p \in \{2\} \wedge q \in \{2\} \wedge 2 \leq i \leq 3 \Rightarrow \texttt{loop}(f_p^i) \wedge 3 \leq csl(p, q)$

Transformation:

$$\begin{array}{lllllll} T_1: & \{[i, & j] \rightarrow [1, & i, & i+j, & 1]\} \\ T_2: & \{[i, & j] \rightarrow [2, & i, & i-j, & 2]\} \end{array}$$

Loop interchange first loop (Unimodular Transformation):

$$k = 2,\ U = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix},\ D = \{1\},\ L = 2$$

Requirements:

$\forall i, p, q\ p \in \{1\} \wedge q \in \{1\} \wedge 2 \leq i \leq 3 \Rightarrow \texttt{loop}(f_p^i) \wedge 3 \leq csl(p, q)$

Transformation:

$$\begin{array}{lllllll} T_1: & \{[i, & j] \rightarrow [1, & i+j, & i, & 1]\} \\ T_2: & \{[i, & j] \rightarrow [2, & i, & i-j, & 2]\} \end{array}$$

Loop interchange second loop (Unimodular Transformation):

$$k = 2,\ U = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix},\ D = \{2\},\ L = 2$$

Requirements:

$\forall i, p, q\ p \in \{2\} \wedge q \in \{2\} \wedge 2 \leq i \leq 3 \Rightarrow \texttt{loop}(f_p^i) \wedge 3 \leq csl(p, q)$

Transformation:

$$\begin{array}{lllllll} T_1: & \{[i, & j] \rightarrow [1, & i+j, & i, & 1]\} \\ T_2: & \{[i, & j] \rightarrow [2, & i-j, & i, & 2]\} \end{array}$$

Figure 2.20: Deriving time mappings continued

Chapter 3

The Algorithms

3.1 The Issues

As was stated earlier, the task of mapping a program written in a sequential programming language onto a multi-processor machine can be divided into two major subproblems: deciding how to distribute the computation amongst the available processors and deciding how to order the computations. Finding a close to optimal solution in a feasible amount of time for either of these problems in isolation is still an open problem; it is even more difficult to solve these problems simultaneously.

To simplify the problem, I first decide how to distribute the computations amongst the available processors, and then, based on that decision, decide how to order the computations. In making this simplification however, I am very mindful of the fact that the two problems are tightly coupled. As much as possible, I have tried to take this into account when devising methods to solve the first of these problems, namely distributing the computations.

In deciding how to distribute the computations amongst the available processors, I want to minimize the amount of communication between processors while at the same time preserving some degree of parallelism. Minimizing communication between processors can be accomplished without regard to the order of the computations. However, achieving sufficient parallelism does depend on the execution order.

Through a combination of scalar and array expansion or privatization, loop distribution, statement reordering and loop interchange, it is often possible to expose parallel loops that did not exist in the original program. Even if parallel loops exist in the original program, distributing the iterations of the newly exposed parallel loops rather than the original parallel loops might result in a higher granularity of parallelism or in lower inter-processor communication costs. In some cases, there may be no parallel loops to exploit, but the program may be able to be transformed to use doacross/pipelining techniques to allow computation and communication to be overlapped.

The phase that distributes the computations will be followed by a phase that reorders the computations so as to achieve maximal parallelism from the chosen distribution. So, it is important not to be influenced by the original computation order when evaluating how much parallelism could be achieved by distributing the iterations of a particular loop.

An overview of my entire optimization system is shown in Figure 3.1. Each of the sub-tasks shown in this overview will be explained in the remained of this chapter. The reader is advised to refer back to this overview from time to time, as they progress through this material.

```
Select Space Mappings:
    Calculate extended direction vectors for communication analysis
    Calculate transitive data dependences for parallelism analysis
    Analyze blocked distribution:
        Analyze comm. for each pair of candidate space mappings
        Analyze parallelism for each candidate space mapping
        Search for best combination of candidate space mappings
    if (any statement has an unbalanced workload)
        Analyze cyclic distribution:
            Analyze comm. for each pair of cand. space mappings
            Analyze parallelism for each cand. space mapping
            Search for best combination of cand. space mappings
        if (cost of cyclic solution < cost of blocked solution)
        then Unbalanced, so use cyclic distribution
        else Unbalanced, but use a blocked distribution
    else Balanced, so use a blocked distribution
    Form affine space mappings by adding constants
Select Time Mappings:
    Analyze locality for each candidate loop permutation
    Analyze compatibility for each pair of cand. loop permutations
    Search for best combination of candidate loop permutations
    Form time mappings by adding constant levels to best perm.
Generate Code:
    Generate new loop structures using time mappings
    Generate SPMD code using space mappings
    Insert synchronization for inter-processor dependences
```

Figure 3.1: Overview of the entire optimization process

3.2 Selecting Space Mappings

3.2.1 Introduction

The problem of automatically distributing computation has been addressed by a large number of authors [Gup92, Fea94, AL93a, BKK93, GAL95, SSP+95]. My work improves on most previous work in the following ways:

1. I am not influenced by the order of the computation of the original program. I use methods to determine the parallelism inherent in the program rather than the parallelism that can be obtained using the computation order in the original program.

2. When analyzing parallelism, I not only examine each loop to determine whether its iterations can be run entirely independently, but also whether its iterations can be pipelined (a lesser but still important form of parallelism). Exploiting this form of parallelism requires that a SPMD rather than a SIMD model be used.

3. I associate a space mapping with each statement, which allows me to represent dynamic data distributions without having to partition the program into phases, as well as allowing me to represent non-data distributions.

4. I obtain accurate indications of the relative volumes of different inter-processor communications by computing the dimensionality of value-based flow dependence relations[PW93] (an abstraction that precisely describes which iterations actually read values written by which other iterations). This allows me to analyze communication costs without knowing the order of the computations.

5. I solve the resulting graph search problem exactly, and have shown experimentally that I can normally do so in a feasible amount of time by using a number of very effective but safe pruning strategies. Other researchers use heuristic or greedy algorithms.

6. I simultaneously optimize for communication and parallelism, trading one off for the other where necessary to obtain an overall optimal solution.

Throughout this section, I make a number of simplifying assumptions, such as the assumption that all loops have an equal number of iterations (which I denote as n). Some of these assumptions could be eliminated at the cost of substantial complications to my framework. However, the point of my algorithm is not to identify that one decomposition is 10% better than another; my cost model is not sensitive or accurate enough to answer those kinds of questions. It is unclear if there is any way to answer those kinds of questions other than by performing time trials on the target machine. My methods are designed to find a distribution such that no significantly better distribution exists, and could be easily altered to generate a list of all such decompositions.

3.2.2 Candidate Space Mappings

Two basic approaches can be taken to selecting space mappings:

- First generate a finite set of candidate space mappings for each statement. This creates a finite search space where each solution corresponds to selecting one candidate space mapping for each statement. Exact or heuristic search procedures can then used to find an optimal or close to optimal solution from amongst this set of solutions.

- Use a method that directly synthesizes the optimal space mapping for each statement.

```
         for i = 1 to n
             for j = 1 to n
1:                a(i,j) = ...
             for k = 1 to n
2:                ...  = a(i,k+1)
```

Figure 3.2: Extended direction vectors

The disadvantage of the first approach is that if the optimal space mapping is not included in the set of candidate space mappings then it will not be selected. This is not a serious problem in practice because optimal space mappings tend to be very simple in realistic examples. It is therefore very easy to generate a small set of candidate space mappings that will likely contain the optimal space mapping.

The second approach has two major disadvantages. First, it is very difficult to accurately estimate complex performance properties such as communication latency and data locality without having actual space mappings in mind. Second, the synthesis process is inherently global rather than local (i.e., the space mapping chosen for one statement may affect the space mappings that should be chosen for all other statements). The size of the resulting optimization problem (which often can be formulated as an integer programming problem) will therefore be at least proportional to the number of statements in the program. The exponential nature of such problems means that this approach will generally scale poorly to larger programs.

I have therefore decided in favor of the first approach. In my current implementation, I only consider space mappings that map to one dimensional processor arrays. My set of candidate space mappings consists of each dimension in the original iteration space, plus zero (which corresponds to not distributing the computation). For example, the candidate space mappings for statement 1 in the program shown in Figure 3.2 are:

$$\{[i,j] \rightarrow [i]\}$$
$$\{[i,j] \rightarrow [j]\}$$
$$\{[i,j] \rightarrow [0]\}$$

It would be possible to extend my implementation to consider other candidates (including skewed mappings) whenever there is some reason to believe they might be desirable. In any case, the candidate space mappings will be linear (as opposed to affine). In Section 3.2.7, I will describe how to select constant offsets to add to these linear space mappings.

In order for my search procedure to select an optimal solution from amongst the set of all possible solutions there must be some metric for comparing solutions. In the next two sections, I will describe how to estimate the parallelism that will result from selecting each of the candidate space mappings and the amount of inter-processor communication that will result from selecting various pairs of candidate space mappings. The parallelism and communication estimates are combined to give an overall performance estimate for each solution.

3.2.3 Estimating Parallelism

In this section I describe my methods to determine the parallelism inherent a program. My first observation is that the most useful form of parallelism is between different iterations of the same statement, rather than between iterations of different statements. This implies that each statement should be examined separately to determine whether any of its iterations can be executed in parallel. In doing so, however, I want to ignore any constraints on parallelism imposed by the original loop order or by other statements that just happen to be in the same loop nest. On the other hand, it is clearly not sufficient to examine each statement in isolation. It turns out that what I need to consider are all direct and transitive self data dependences of

```
for r = 1 to n
    for p = 1 to n
        for q = 1 to n
            e_pq = e_pq ∨ (e_pr ∧ e_rq)
```

Figure 3.3: Floyd-Warshall algorithm

```
for i = 1 to n
    for j = 1 to n
        D'_ij = D_ij
for r = 1 to n
    for p = 1 to n
        for q = 1 to n
            D'_pq = D'_pq ∪ (D'_pr ∘ D'*_rr ∘ D'_rq)
```

Figure 3.4: Modified form of Floyd-Warshall algorithm

each statement. This takes into account constraints on parallelism imposed by other statements, but only those that can't be avoided. I consider all flow dependences, and whichever output and anti dependences remain after array expansion has been performed (deciding when to apply array expansion is outside the scope of this thesis, and was performed manually for the experiments presented in Section 4).

I have developed two methods for computing transitive self data dependences. Both methods use the same basic algorithm, which is a modified form of the Floyd-Warshall algorithm for computing the transitive closure of a graph (see Figure 3.4). The input to this algorithm is a set of variables D_{pq} representing all direct data dependences from statement p to statement q. The output is a set of variables D'_{pq} representing all transitive data dependences from statement p to statement q.

The original Floyd-Warshall algorithm (see Figure 3.3) can be used to determine the existence of a transitive dependence from any given statement to any other given statement (including itself); however, it cannot determine which iterations of those statements are dependent. To obtain this additional information, rather than using boolean valued variables, I use variables that describe which iterations are dependent on which other iterations. These values need to be combined using union and composition operations rather than boolean "`and`'' and "`or`'' operations. The other modification to the algorithm is somewhat subtle and involves the addition of the ${D'}^*_{rr}$ term. This transitive closure term is added because to find all transitive dependences from statement p to statement q, I need to consider dependence "chains" of the following form:

If there is a transitive dependence from some iteration i of statement p to some

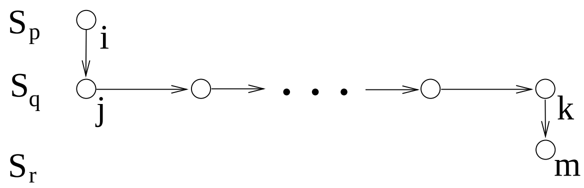


Figure 3.5: Transitive dependences

iteration j of statement q and a transitive self dependence from iteration j of statement q to iteration k of statement q and finally a dependence from iteration k of statement q to some iteration m of statement r, then there is a transitive dependence from iteration i of statement p to iteration m of statement r (see Figure 3.5).

Computing transitive data dependences is very useful for a number of applications other that parallelism analysis, for example they can also be used to detect redundant synchronization statements.

The only difference between the two methods that I have developed is the abstraction used to represent dependences.

An exact method

The first method uses tuple relations to represent data dependences, and uses the operators described in Table 2.1. This method will generally compute the set of transitive dependences exactly. The problem with this method, however, is that the size of the dependence relations generated, and hence the time required to generate them, tends to increase rapidly as the number of the statements is increased. I have found that this method is infeasible for analyzing large programs.

A faster but less accurate method

My other approach is to use extended direction vectors[WB87, Wol91a] to represent dependences. A dependence direction vector is a vector $(v_1, \ldots, v_m)$, where v_i is either '-', '0', or '+', indicating whether the difference between the level i index variable at the source of the dependence is less than, equal to, or greater than the level i index variable at the sink of the dependence. For a normal direction vector, the length of the vector is equal to the maximum common loop depth of the two statements. An extended direction vector has length equal to the minimum loop depth of the two statements. For example, for the flow dependence from statement 1 to statement 2 in Figure 3.2, the value of the level 1 index variable `i`, at the source of the dependence, is equal to the value of the level 1 index variable `i`, at the sink of the dependence. Also, the value of the level 2 index variable `j`, at the source of the dependence, is less than the value of the level 2 index variable `k`, at the sink of the dependence. So, the dependence is represented by the direction vector (0) and by the extended direction vector $(0, -)$.

When taking the union of two sets of extended direction vectors, vectors are combined if and only if doing so will not lead to the loss of information (i.e., if the entries are all identical but one). For example, $(0, +)$ and $(0, 0)$ can be combined to produce $(0, 0+)$; however, $(0, +)$ and $(+, 0)$ can't be combined to produce $(0+, 0+)$ (as that would imply that $(0, 0)$ and $(+, +)$ are possible direction vectors). When taking the union of sets of extended direction vectors with different lengths (which occurs when considering transitive dependences through statements which are not as deeply nested as the statement in question), the shorter vectors are padded with '*', indicating that '-', '0' and '+' are all possible. The composition of direction vectors is performed element-wise and is defined in the obvious manner (i.e. composing '+' and '0' produces '+', composing '+' and '-' produces '*', etc). For extended direction vectors, the transitive closure operation is simply a "no-op" (i.e., '+'* = '+', '0'* = '0' and '-'* = '-').

Loop Transformations

I wish to analyze the parallelism that would result from selecting each candidate space mappings without being influenced by the original loop order. My current implementation considers all legal loop permutations of the loops surrounding each statement (including all combinations of reversing the loops). It would also be possible to extend my implementation to consider

$$
\begin{array}{l}
\text{for } t_1 \\
\quad \ldots \\
\quad\quad \text{for } t_{z-1} \\
\quad\quad\quad \text{for } t_z^P \\
\quad\quad\quad\quad \text{for } t_z^B \\
\quad\quad\quad\quad\quad \text{for } t_z \\
\quad\quad\quad\quad\quad\quad \ldots \\
\quad\quad\quad\quad\quad\quad\quad \text{for } t_m \\
\quad\quad\quad\quad\quad\quad\quad\quad \text{stmt}
\end{array}
$$

Figure 3.6: Straight forward loop structure for block-cyclic

other unimodular loop transformations (including skewed transformations) when there is some reason to believe they might be desirable.

The candidate space mappings are classified according to the amount of synchronization they will require (and hence how much parallelism they permit) within each particular permutation. Even if all permutations of the loops are legal (a number which is exponential in the number of loops), only a very small amount of time will be required, since each statement is considered separately and statements are seldom nested within more than 4 or 5 loops. Each candidate space mapping is given an overall classification based on the amount of synchronization it will require using the best permutation for that particular candidate. The set of best permutations for the candidate space mapping that is ultimately selected will later be used as the starting point for selecting a time mapping for that statement.

Synchronization costs

To analyze the amount of synchronization that will be required for a particular candidate using a particular permutation, the structure of the loops that would ultimately be used must be considered. I analyze the general case of using a block-cyclic distribution rather than separately analyzing both the block and cyclic cases. If the loop of the candidate space mapping is at level z in the current permutation (and the statement is nested within m loops), then a straight forward implementation of block-cyclic distribution would lead to the loop structure shown in Figure 3.6.

The t_z^P loop iterates over the set of physical processors, the t_z^B loop iterates over the blocks and the t_z loop iterates over the iterations within each block. In a block distribution the t_z^B loop will be degenerate and in a cyclic distribution the t_z loop will be degenerate.

```
for t_1
  ...
    for t_x
      for t_z^B
        for t_{x+1}
          ...
            for t_{z-1}
              for t_z^P
                for t_z
                  ...
                    for t_m
                      stmt
```

Figure 3.7: Improved loop structure for block-cyclic

We define:

x as the deepest loop level that carries a transitive self-dependence with a negative dependence distance at the distributed loop level.

y as the deepest loop level that carries a transitive self-dependence with a positive dependence distance at the distributed loop level.

It is legal to move the t_z^B loop out to just inside the t_x loop, so a better loop structure is shown in Figure 3.7.

This can be converted to SPMD code as follows:

- Remove the t_z^P loop.
- Insert a barrier inside the t_z^B loop, if necessary, to enforce any dependences going from higher numbered physical processors to lower numbered physical processors.
- Insert post-and-wait style synchronization inside the t_y loop, if necessary (i.e. if $x < y$), to enforce any dependences going from lower numbered physical processors to higher numbered physical processors. The form of post-and-wait style synchronization that I use causes each processor (except the lowest numbered processor) to wait for the next lowest numbered processor.

My decision to use this style of post-and-wait synchronization and to use it only to synchronize dependences from lower numbered physical processors to higher numbered physical processors was made in order to simplify analysis, to allow multiple dependences to be satisfied by a single post-and-wait pair, and to avoid the possibility of deadlock. It is certainly not the only choice that could be made; for example, post-and-wait style synchronization could be used to synchronize only those dependences from higher numbered physical processors to lower numbered physical processors. If more sophisticated techniques were used to avoid deadlock or if other forms of post-and-wait style synchronization were used, then post-and-wait style synchronization could even be used to synchronize at least some dependences in both directions.

By moving the t_z^B loop out as far as possible, a minimal number of barriers will be executed. The placement of the t_z^B loop also implies that any dependences carried by loops t_{x+1} through t_{z-1} will be from a lower numbered physical processor to a higher numbered physical processor, so some form of parallelism (either pure or pipelined) will result within each iteration of the t_z^B loop.

On some architectures it is more efficient to synchronize using forms of synchronization other than barriers, for example using send and receive pairs on a distributed memory machine. The point I am trying to make is not that barriers should be used, but rather that at the places where I have said barriers should be inserted, some form of non-pipelined synchronization will be used. So from a purely cost point of view, it is reasonable to model this as a barrier.

The maximum amount, D, by which different processors can be expected to be out of lock-step, is computed as follows:

- If any dependences are carried by the distributed loop, the expected delay between the time processor p can start, and the time processor $p+1$ can start, will be $L + Bn^{m-y}$, where L is the inter-processor message latency, B is the block size, n is the number of iterations per loop, and P is the number of physical processors[1]. The wait from when the first processor reaches a barrier until the last reaches the barrier will be $P-1$ times the delay between successive processors. I simplify this slightly to $D = P(L + Bn^{m-y})$.
- If no dependences are carried by the distributed loop, but there are inter-processor dependences carried by loops t_{x+1} through t_{y-1}, then those dependences from processor p to processor $p+1$ may force processor $p+1$ to lag L behind processor p. I again simplify slightly and estimate $D = PL$.
- If no inter-processor dependences are carried by loops t_{x+1} through t_y, then the processors should remain synchronized to within $D = L$.

The number of barrier synchronizations performed will be $\frac{n^{x+1}}{BP}$. To perform a barrier synchronization, the processors must exchange messages (costing L) and synchronize (costing D). Since $D \geq L$, I simplify the cost per barrier to D. The total synchronization cost for each statement is therefore $\frac{n^{x+1}D}{BP}$.

Load balance

The loop bounds of each statement are examined to determine whether the amount of work in each iteration will be constant. If any statements have unbalanced loops, then a cyclic distribution is considered in addition to considering a block distribution. When evaluating block distributions for candidates with unbalanced workloads, an additional $\frac{n^m}{2}$ time is added to the overhead estimate. This heuristic is based on the fact that most unbalanced workloads are a result of triangular loops and the difference between the amount of work in a triangular loop and a rectangular loop is $\frac{n^m}{2}$. In Section 3.2.5 higher communication estimates will be used for some dependences when using a cyclic distribution.

Compatible candidates

After synchronization analysis has been performed, the minimum degree of synchronization required for each candidate will be known. For each candidate there will be a set of legal loop permutations that lead to this minimum degree of synchronization. For example, the candidate space mapping $\{[k,i] \rightarrow [k]\}$ for statement 1 in Figure 3.8 will produce parallel execution at loop depth 2, only if loop permutation (i,k) is used for statement 1. Similarly, the candidate space mapping $\{[k,i,j] \rightarrow [j]\}$ for statement 2 will produce parallel execution at loop depth 2, only if one of the following legal loop permutations are used for statement 2: $\{(k,-j,-i),(k,-j,i),(k,j,-i),(k,j,i)\}$.

Unfortunately, in this case, because of data dependences, the first statement's permutation: (i,k), can not be used with any of the second statement's permutations: $\{(k,-j,-i),(k,-j,i),(k,j,-i),(k,j,i)\}$. In other words, if candidate $\{[k,i] \rightarrow [k]\}$ is selected for the first statement and candidate $\{[i,j,k] \rightarrow [j]\}$ for the second statement, then parallelism will not be able to be achieved at loop depth 2 for both statements no matter how the iterations are reordered. Thus, analyzing parallelism for each statement in isolation can lead to overestimation of parallelism.

[1] In my current implementation I simply set L = 10, n = 100, P = 10 and B = 1 or 10, for cyclic and block distributions respectively

```
        for k = 1 to n
            for i = k+1 to n
1               a(i,k) = a(i,k) / a(k,k)
                for j = k+1 to i
2                   a(i,j) = a(i,j) - a(k,j)*a(i,k)
```

Figure 3.8: Gaussian elimination

To address this problem, I consider all pairs of statements (p, q), and determine which candidates of statement p are compatible with which candidates of statement q. Candidate C_p of statement p is compatible with candidate C_q of statement q if there exist permutations π_p and π_q for statements p and q respectively such that π_p produces the minimum degree of synchronization for C_p, π_q produces the minimum degree of synchronization for C_q, and π_p is compatible with π_q (see the next subsection on compatible permutations).

This compatibility information will be used when constructing the search problem described in Section 3.2.6 to try to ensure that parallelism is not overestimated as described above. The compatibility tests are only performed on each pair of statements in isolation. It is theoretically possible for each pair of selected candidates to be compatible but for the set of candidates as a whole to be incompatible. However, since transitive dependences are used in determining which permutations are compatible, it is very unlikely that this will occur. This problem has not arisen in any of the examples I have tried. If this ever did occur it would be very easy to detect, and it would be necessary to expand the search space in order to find a compatible solution.

Compatible permutations

The following test is used to determine whether permutation π_p for statement p is compatible with permutation π_q for statement q. First, a set of direction vectors is constructed that describe the order in which the iterations of statement q will be executed if loop permutation π_q is applied. These direction vectors do not correspond to actual data dependences, but rather to ordering constraints that will be satisfied if that permutation is used. For example, if permutation (k, i, j) is used for statement 2 in Figure 3.8 then the following set of direction vectors $\{(0, 0, +), (0, +, *), (+, *, *)\}$ would be constructed. In general, the set will be:

$$c_{qq} = \bigcup_{m \in \{0,\ldots,n-1\}} \left\{ \pi_q(\underbrace{0,\ldots,0}_{m}, +, \underbrace{*,\ldots,*}_{n-m-1}) \right\} \tag{3.1}$$

where $\pi_q(x_1, \ldots, x_n)$ means apply permutation π_q to the vector $(x_1, \ldots, x_n)$. Applying a permutation to a direction vector also involves reversing directions as indicated by the permutation. For example, if permutation $(k, j, -i)$ was used for statement 2 then the following set of direction vectors would be constructed $\{(0, +, 0), (0, *, -), (+, *, *)\}$.

Next, these ordering constraints are combined with the transitive dependences between statements p and q to infer new ordering constraints on statement p under the assumption that permutation π_q will be used for statement q. The new ordering constraints are:

$$c_{pp} = D'_{pq} \circ c_{qq} \circ D'_{qp} \tag{3.2}$$

This calculation can be performed using either tuple relations or extended direction vectors. Permutation π_p is compatible with permutation π_q if and only if π_p is legal with respect to the new set of ordering constraints c_{pp}.

For example, if in the example from Figure 3.8, $\pi_2 = (k, j, -i)$ then:

$$\begin{array}{lcll} c_{22} & = & \{(0,+,0),(0,*,-),(+,*,*)\} & \text{(from Eqn 3.1)} \\ D'_{12} & = & \{(0,0)\} & \text{(from Fig 3.4)} \\ D'_{21} & = & \{(+,0+)\} & \text{(from Fig 3.4)} \\ c_{11} & = & \{(+,*)\} & \text{(from Eqn 3.2)} \end{array}$$

So permutation (i, k) for statement 1 is incompatible with permutation $(k, j, -i)$ for statement 2 because (i, k) is not legal with respect to c_{11} (since the permuted direction vector $\pi_1(c_{11}) = \{(*,+)\}$ is not lexicographically positive).

3.2.4 False Sharing on Shared Memory Architectures

False sharing is a problem commonly encountered when trying to program machines with cache coherent shared memory architectures. It occurs when one processor performs a write, and soon after, some other processor accesses a different memory location belonging to the same cache line. The write performed by the first processor will cause an entire cache line on second processor to be invalidated, resulting in a cache miss for the second access, despite the fact that the memory location actually being referenced may be up to date.

False sharing may occur due to references belonging to different statements. To determine if false sharing will result between a given pair of references, it is generally necessary to know both the time and space mappings used for the statement(s) containing those references. It is therefore very difficult and expensive to accurately predict all cases of false sharing for the purpose of evaluating space mappings. I therefore use a rather crude model for predicting false sharing, that detects many common cases of false sharing, but misses some cases, and occasionally predict false sharing when it will not occur. Improving this model is a high priority for future work.

I assume that arrays are laid out according to the C programming language conventions (i.e., `a(i,j)` and `a(i,j+1)` occupy adjacent memory locations). The main simplifying assumption that I make, is that false sharing will only occur due to two different iterations of a single statement writing to the same cache line. So, if a block distribution is used and the candidate distributed loop's index variable occurs in at least one of the left side subscript expressions, then I assume that false sharing will not occur. Similarly, if a cyclic distribution is used and the candidate distributed loop's index variable occurs in at least one of the first $m - 1$ left side subscript expressions (where m is the dimensionality of the left side array), then I assume that false sharing will not occur. Otherwise, I assume pessimistically that false sharing will occur and assign a very high cost to that candidate.

False sharing can also be eliminated in some cases by transforming the layout of the arrays. Deciding when and how to do this is very difficult since a given array is normally accessed in more than one statement, each of which implies a preferred layout for the array.

3.2.5 Estimating Communication

My primary assumption is that communication will only be required between processors if one processor writes a value to a location and some other processor later reads that value from that location. Value-based flow dependence relations [PW93] are used to obtain accurate indications of the relative volumes of different inter-processor communications. I always use value based flow dependence relations in this section regardless of the abstraction chosen in the previous section. Value based dependence relations precisely describe which iterations actually read values written by which other iterations. For example, there is no value based flow dependence from statement 1 to statement 3 in Figure 3.9, since all memory locations written by statement 1 are overwritten by statement 2 before statement 3 can read them. The value based flow dependence from statement 2 to statement 3 would be represented by the dependence relation $\{[i,i] \rightarrow [i,i] \mid 1 \leq i \leq n\}$. From this information, it can be determined that only n values will

```
       for i = 1 to n
          for j = 1 to n
1:           a(i,j) = ...
             ...
2:           a(i,j) = ...
3:           ... = a(i,i)
```

Figure 3.9: Value based dependence example

```
    for t = 0 to ITERS
        for j = 0 , DIM-1
            for k = 1, DIM-1
1:              X[j, k] = ...
        for j = 0 , DIM-1
2:          ...  = X[j, DIM-1] + ...
```

$$d_{12} : \{[t, j, k] \rightarrow [t, j] \mid k = \mathrm{DIM} - 1 \wedge 0 \leq t \leq \texttt{ITERS} \wedge 0 \leq j < \texttt{DIM} \wedge 2 \leq \texttt{DIM}\}$$

Figure 3.10: Constant distance example

be communicated from statement 2 to statement 3, despite the fact that both statements have n^2 iterations.

I simplify matters by assuming that all loops have some unknown constant number of iterations "n" (even those with known constant loop bounds). This allows me to associate a dimensionality (or rank) with each value based flow dependence. For example, the dimensionality of the dependence relation given in Figure 3.9 is 1. The dimensionality of a relation is computed as the dimensionality of the relation's domain minus the number of equality constraints required to describe the domain.

Using dimensionality allows me to obtain accurate indications of the relative volumes of different inter-processor communications without having to resort to complex and expensive symbolic volume estimation algorithms [Pug94]. Dimensionality, however, only provides an asymptotic indication of the amount of communication that will result. It may, for example, indicate that $O(n^2)$ or $O(n^3)$ communication is required. It can not distinguish between, for example, $2n^2$ and $3n^2$ communication. So, there is no point in considering factors such as the aggregation of messages from two different writes, or the orientation of cache lines, since such factors can only effect the amount of communication by a constant factor.

For each value-based flow dependence, each combination of candidate space mappings for the two statements involved in the dependence is considered. The equality constraints in the dependence relations are examined and the difference between the virtual processor to which the first statement is mapped and the virtual processor to which the second statement is mapped is

		Stmt 2		
		0	t	j
Stmt 1	0	zero	not constant	not constant
	t	not constant	zero	not constant
	j	not constant	not constant	zero
	k	DIM-1	not constant	not constant

Table 3.1: Virtual processor differences

determined. The possible differences are: zero , a constant other than zero, and a non-constant difference. Table 3.1 shows each combination of candidate space mappings for statements 1 and 2 in Figure 3.10, with information about the difference in virtual processors for the value based flow dependence shown.

If the difference in virtual processors is zero then I estimate that no communication will occur. In the case of self dependences, this estimate is always exact. In the case of non-self dependences, this constitutes an optimistic assumption that the statements will be assigned identical constant offsets in Section 3.2.7. If the difference in virtual processors is a constant other than zero, then it is assumed that *nearest-neighbor communication* will occur. Again, in the case of self dependences, this is always true. In the case of non-self dependences, this constitutes a pessimistic assumption that the difference between the constant offsets assigned to the statements will not be the same as the non-zero constant difference between the virtual processors (if that were the case, then there would be no communication).

If a blocked distribution is being considered and communication is nearest-neighbor, then many of the dependences will be between different virtual processors that are mapped to the same physical processor. So, for a dependence of dimension n^d, I estimate that the amount of inter-processor communication will be n^d/B (where B is the number of virtual processors in each block). If the communication is not nearest-neighbor, or if a cyclic distribution is being considered, then a dependence with dimensionality n^d has a communication estimate of n^d.

3.2.6 The Search Problem

The space mapping selection problem is now represented as a weighted graph. With the exception of the candidate space mappings described in Section 3.2.6, the graph will contain a node corresponding to each candidate space mapping of each statement. The node weights will be the parallelism overheads as derived in Sections 3.2.3 and 3.2.4 and the edge weights will be the communication estimates as derived in Section 3.2.5. Parallelism overheads are first multiplied by a machine dependent constant that represents the ratio of computation speed to communication speed on the target machine. By varying this parameter, it can be determined whether or not a given solution is likely to be optimal across a wide variety of machines.

Incompatible candidates

If incompatible candidates exist (see Sub-section 3.2.3), the parallelism that can be achieved using some candidate space mapping may depend on which candidates are chosen for other statements. I therefore use two nodes to represent such candidates. For one node, it is optimistically assumed that the degree of parallelism estimated when considering the statement in isolation can be achieved, and for the other node, it is pessimistically assumed that choices made for other statements will force the use of a permutation that leads to the least possible degree of parallelism for this statement. The communication costs will be the same for both nodes, but the parallelism overhead estimate will be higher for the pessimistic version. If two candidates are incompatible then the optimistic version of both candidates should not be selected simultaneously. To ensure that does not occur, an edge with infinite cost is added between the optimistic versions of incompatible candidates.

It is possible for good solutions to be overlooked by this algorithm, since it is occasionally overly pessimistic about the amount of parallelism that can be achieved by using incompatible candidate space mappings. My intuition is that this is not a major problem in practice, however, investigating this problem and possibly improving the algorithm is also a high priority for future research.

The search procedure

The search problem is to select exactly one node for each statement, such that the sum of the node costs of selected nodes and sum of edge costs between selected nodes is minimized.

```
inout real a[1024,1024]
for k = 1, 1023 do
    for i = k+1, 1024 do
        a[i,k] = a[i,k]/a[k,k]
        for j = k+1, 1024 do
            a[i,j] = a[i,j]-a[k,j]*a[i,k]
        endfor
    endfor
endfor
```

Figure 3.11: Gaussian elimination example

Many previous approaches to automatically minimizing inter-processor communication solve similar formulations with heuristic or greedy algorithms. I instead solve the problem exactly. Admittedly, my approach is not guaranteed to find the optimal set of space mappings since the edge weights in the graph problem I am trying to solve are only estimates of actual cost. However, by using an exact algorithm to solve the search problem, I know that any imprecision is bounded by the imprecision of the performance estimates, rather than being unbounded as would be the case if I used a heuristic algorithm to solve the search problem. Within this framework, I can easily substitute different performance estimation algorithms until I find the one that best trades off precision for efficiency. The actual algorithm for performing the search is described in Section 3.4.

Figure 3.12 shows the graph that results from the Gaussian elimination program shown in Figure 3.11 when a cyclic distribution is being considered and a computation to communication ratio of 5 is used. The nodes labeled with primes are the pessimistic versions of their respective candidates. The search algorithm selects the i node for both statements, with an overall minimal cost of 101020.

3.2.7 Alignment

Adding a constant to the linear space mappings selected in the previous section can eliminate some nearest-neighbor communications. A constant offset will have no affect on communication between different iterations of the same statement. For dependences between different statements, however, if the dependent iterations map to virtual processors separated by a constant distance, then adding appropriate constants to the selected space mappings can map them to the same virtual processors. The alignment algorithm described here adds constants that are an affine function of the symbolic constants in the program. For example, for the program in Figure 3.10, the alignment algorithm might select "DIM-1" as the constant to add to the space mapping of statement 1.

In the previous section, I tentatively assumed that the same constant would be added to all space mappings, so nearest-neighbor communications would be eliminated only if the linear space mappings assigned to their respective statements in the previous section were equal. In this section I try to improve on this by adding potentially different constants to each space mapping. However, I regard any such improvements as a bonus, and certainly don't claim to solve the problem optimally (doing so would be too expensive). My alignment algorithm uses a greedy approach to decide which dependences with constant virtual processor distances will be made intra-processor. The alignment algorithm maintains a partitioning of the statements such that within each partition, the relative differences between the statements' constant parts are known. Initially all statements are in separate partitions. The algorithm processes all constant distance dependences in decreasing order based on their dimensionalities. If there is a dependence from statement p to statement q and p and q are in different partitions, then the

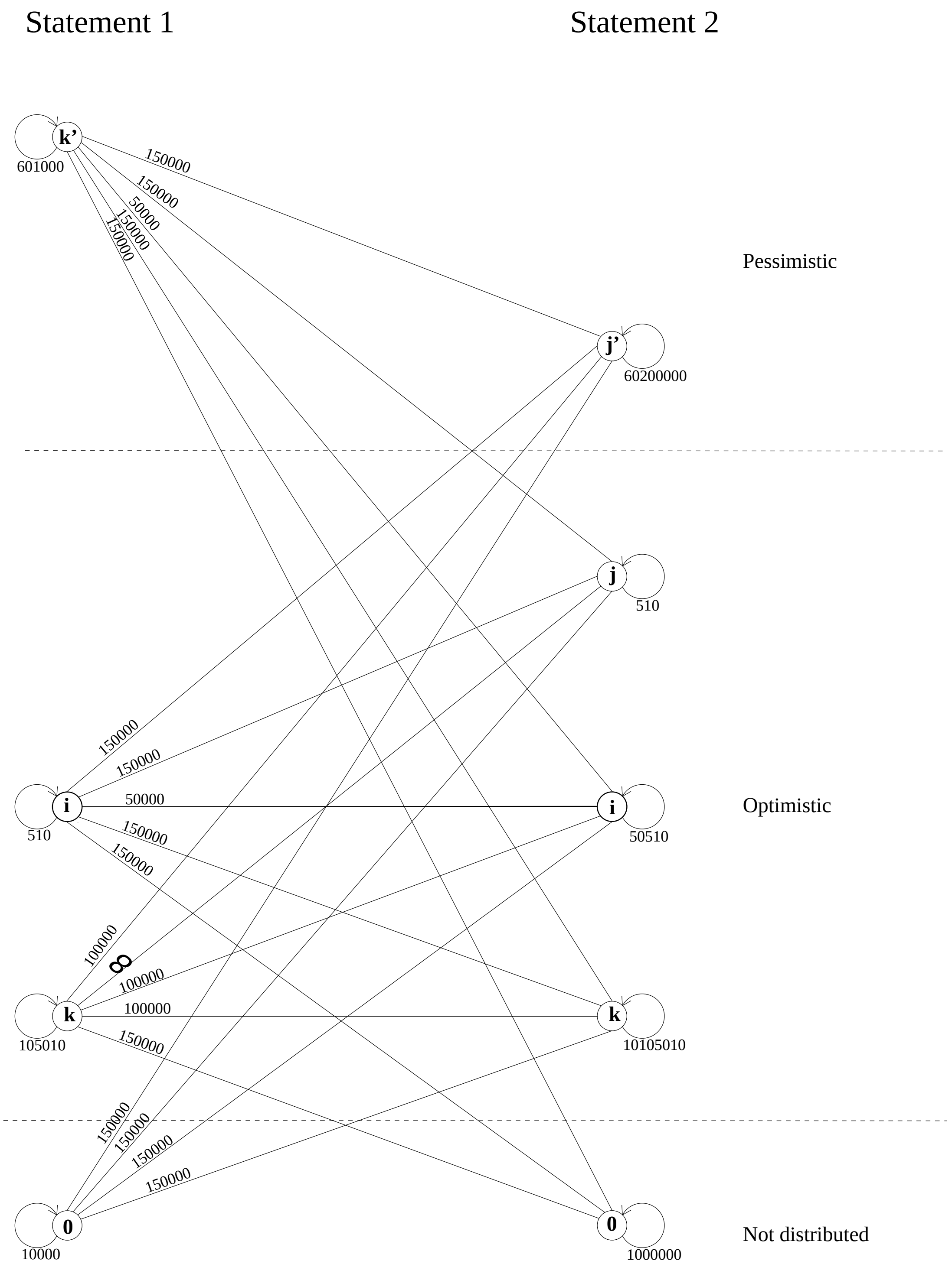


Figure 3.12: Search graph for Gaussian elimination

```
  for t = 0 to ITERS
    for j = 0 to DIM-1
      for k = 1 to DIM-1
1       X[j,k] = X[j,k]-X[j,k-1]*A[j,k]/B[j,k-1]
2       B[j,k] = B[j,k]-A[j,k]*A[j,k]/B[j,k-1]
    for j = 0 to DIM-1
3     X[j,DIM-1] = X[j,DIM-1]/B[j,DIM-1]
    for j = 0 to DIM-1
      for k = DIM-2 to 0 by -1
4       X[j,k] = (X[j,k]-A[j,k+1]*X[j,k+1])/B[j,k]
    for j = 1 to DIM-1
      for k = 0 to DIM-1
5       X[j,k] = X[j,k]-X[j-1,k]*A[j,k]/B[j-1,k]
6       B[j,k] = B[j,k]-A[j,k]*A[j,k]/B[j-1,k]
    for k = 0 to DIM
7     X[DIM-1,k] = X[DIM-1,k]/B[DIM-1,k]
    for j = DIM-2 to 0 by -1
      for k = 0 to -1+DIM
8       X[j,k] = (X[j,k]-A[j+1,k]*X[j+1,k])/B[j,k]
```

Figure 3.13: Adi

dependence can be made intra-processor. In this case, the partitions containing statements p and q are merged and the virtual processor difference between statements p and q is recorded. When all dependences have been processed or when only one partition remains, a constant is arbitrarily chosen for one statement in each partition and is used to compute constants for the other statements in the partition.

3.2.8 An Example: adi

Figure 3.13 shows a program fragment called `adi` that is used in alternating direction implicit integration. My parallelism analysis phase produces the results shown in Table 3.2. If I set the communication to computation ratio parameter to a low value such as 0.01, then the space mappings shown in Figure 3.14(a) are obtained. These space mappings result in all statements being executed in parallel; however, they also result in 4 n^3 and 12 n^2 inter-processor communications. If I set the communication to computation ratio parameter to a more realistic value such as 5.0, then the space mappings shown in Figure 3.14(b) are obtained. These space mappings result in the first four statements being parallelized and three of the last four statements being pipelined, but they result in only 4 n^3 and 4 n^2 inter-processor communications. If I set the communication to computation ratio parameter to an even higher value such as 100.0 (as might be the case for a network of workstations), then the space mappings shown in Figure 3.14(c) are obtained. These space mappings result in all statements being executed sequentially, with no inter-processor communication.

3.3 Selecting Time Mappings

3.3.1 Introduction

After having chosen a space mapping for each statement as described in the previous section, I now need to select time mappings that will achieve the degrees of parallelism that I estimated

Stmts	Transitive Dependences	Candidate space mappings		
		t	j	k
		barriers $\times$ cost	barriers $\times$ cost	barriers $\times$ cost
1,2,4	(+,*,*) (0,0,+)	$1 \times P(L + Bn^2)$	$n \times L$	$n \times P(L + B)$
3	(+,*)	$1 \times P(L + Bn)$	$n \times L$	
5,6,8	(+,*,*) (0,+,0)	$1 \times P(L + Bn^2)$	$n \times P(L + B)$	$n \times L$
7	(+,*)	$1 \times P(L + Bn)$		$n \times L$

Table 3.2: Synchronization costs for adi

$1 : \{[t, j, k] \rightarrow [j]\}$	$1 : \{[t, j, k] \rightarrow [j]\}$	$1 : \{[t, j, k] \rightarrow [0]\}$
$2 : \{[t, j, k] \rightarrow [j]\}$	$2 : \{[t, j, k] \rightarrow [j]\}$	$2 : \{[t, j, k] \rightarrow [0]\}$
$3 : \{[t, j] \rightarrow [j]\}$	$3 : \{[t, j] \rightarrow [j]\}$	$3 : \{[t, j] \rightarrow [0]\}$
$4 : \{[t, j, k] \rightarrow [j]\}$	$4 : \{[t, j, k] \rightarrow [j]\}$	$4 : \{[t, j, k] \rightarrow [0]\}$
$5 : \{[t, j, k] \rightarrow [k]\}$	$5 : \{[t, j, k] \rightarrow [j]\}$	$5 : \{[t, j, k] \rightarrow [0]\}$
$6 : \{[t, j, k] \rightarrow [k]\}$	$6 : \{[t, j, k] \rightarrow [j]\}$	$6 : \{[t, j, k] \rightarrow [0]\}$
$7 : \{[t, k] \rightarrow [k]\}$	$7 : \{[t, k] \rightarrow [DIM - 1]\}$	$7 : \{[t, k] \rightarrow [0]\}$
$8 : \{[t, j, k] \rightarrow [k]\}$	$8 : \{[t, j, k] \rightarrow [j]\}$	$8 : \{[t, j, k] \rightarrow [0]\}$
$(a) : r = 0.01$	$(b) : r = 5.0$	$(c) : r = 100.0$

Figure 3.14: Selected space mappings for adi

could be achieved. To do so, I reuse much of the analysis performed to select the space mappings. In particular, the loop permutations considered in order to analyze parallelism, will become the candidate time mappings. In general, for each candidate space mapping, there will exist a set of loop permutations that achieve the minimal degree of synchronization possible for that candidate. If the space mapping selection algorithm selects the optimistic version of a given space mapping, then I use the sets of loop permutations that achieve the minimal degree of synchronization possible for that space mapping as the candidate loop permutations for that statement. If the pessimistic version is selected, or if the zero space mapping is selected, then all legal loop permutations are considered candidates for that statement. These sets of candidate loop permutations are actually computed during parallelism analysis for selecting space mappings, so no further work is required at this stage. These candidates are evaluated according to a number of criteria including the degree of cache reuse they will cause. A search process similar to that used in Section 3.2.6 is used to select the best overall set of loop permutations. These loop permutations are then formed into time mappings by adding constant offsets and constant levels as described in Section 3.3.5. Constant offsets and constant levels allow loop alignment and loop fission/fusion decisions to be represented respectively.

3.3.2 Estimating Data Locality

My estimate of data locality is an estimate of the number of cache misses that will occur. To simplify the problem I only take into account self reuse, i.e., reusing a cache line used by a previous iteration of the same reference. This simplification is justifiable because self reuse can reduce the number of cache misses by a factor of $\Omega(n)$, whereas group reuse can only reduce the number of cache misses by a factor of $O(1)$. I do, however, take into account both temporal and spatial locality [WL91].

I say that an array reference (with dimensionality m):

- is completely pinned at depth k if and only if at least one of the subscript expressions involves the index variable at depth k in the candidate loop permutation, but no subscript expressions involve index variables deeper than depth k in the candidate loop permutation. Intuitively, if a reference is completely pinned at depth k then all iterations of loops inside the k^{th} loop will access the same memory location.

- is partially pinned at depth k if and only if at least one of the first $m-1$ subscript expressions involves the index variable at depth k in the candidate loop permutation, but none of the first $m-1$ subscript expressions involve index variables deeper than depth k in the candidate loop permutation. Intuitively, if a reference is partially pinned at depth k then all iterations of loops inside the k^{th} loop will access the same row of the array (and hence, often the same cache line).

For example, the array reference $a[2{*}j{+}1][j{+}k]$ is partially pinned at depth 1 and completely pinned at depth 2 by the candidate loop permutation (j, k, i).

I make two basic assumptions regarding whether an array element is expected to remain in the cache from one use to the next:

- If a particular cache line is being accessed every iteration during any interval of time, then I assume that that cache line will remain in the cache for that interval of time.
- Otherwise, I assume that it will be flushed from the cache by intervening references.

If a reference becomes completely pinned at depth k and was already partially pinned at depth $k' < k$, then successive iterations of the k^{th} loop will access array elements that map to the same cache line approximately $C - 1/C$ of the time (where C is the cache line size). So, given the above assumption, the expected number of cache misses is n^k/C.

If a reference becomes completely pinned at depth k, but was not already partially pinned, then successive iterations of the loops inside the k loop will access the same array element. So, given the above assumption, the expected number of cache misses is n^k.

The cache misses incurred by the entire statement is the sum the cache misses incurred for all array references in the statement.

Although relatively simple, I have found that this model for data locality produces adequate results and could easily be adapted for use in other transformation systems.

3.3.3 Incompatible Time Mappings

In Section 3.2.3, I described how to determine which permutations are compatible with which other permutations. It is not possible to generate semantically correct code using incompatible permutations, so I do not want to select incompatible permutations. I take care of this by introducing an edge with infinite weight between such permutations.

The incompatibility tested for in Section 3.2.3 dealt only with legality, that is, whether or not code could be generated that respected all data dependences in the original program. There are other ways in which permutations can be incompatible. A different form of incompatibility that I will deal with here, is whether or not the permutations allow the maximal amount of parallelism to be exploited for both statements. For example, consider the case where after applying permutations π_1 and π_2 to statements 1 and 2 respectively, statement 1 is distributed at depth 1 and is completely parallel and statement 2 is distributed at depth 2 and is completely sequential. If the inter-statement dependences are such that it is not possible to place statements 1 and 2 in different loops at depth 1 (i.e. apply loop distribution), then π_1 and π_2 would be considered incompatible because they do not allow the maximal amount of parallelism to be exploited for statement 1.

To determine which permutations are incompatible in this new way, I first determine the minimum depth at which each pair of statements can be separated (i.e., placed in separate loops), using each of the candidate loop permutations. I again use c_{pp} as defined in Equation 3.2, but now, rather than determining whether π_p is legal with respect to c_{pp}, I determine the maximum depth (according to π_p), at which any of these dependences are carried. This gives the minimum depth at which the statements can be separated.

For example, consider the Gaussian elimination program shown in Figure 3.11. If $\pi_1 = (k, i)$ and $\pi_2 = (k, i, j)$ then

$$\begin{array}{lcll} c_{22} & = & \{(+,*,*),(0,+,*),(0,0,+)\} & \text{(from Eqn 3.1)} \\ D'_{12} & = & \{(0,0)\} & \text{(from Fig 3.4)} \\ D'_{21} & = & \{(+,0)\} & \text{(from Fig 3.4)} \\ c_{11} & = & \{(+,*)\} & \text{(from Eqn 3.2)} \end{array}$$

$\pi_1(c_{11}) = \{(+,*)\}$, so the minimum depth at which the statements can be separated is 1. If $\pi_1 = (i, k)$ and $\pi_2 = (i, k, j)$ then

$$\begin{array}{lcll} c_{22} & = & \{(*,+,*),(+,0,*),(0,0,+)\} & \text{(from Eqn 3.1)} \\ D'_{12} & = & \{(0,0)\} & \text{(from Fig 3.4)} \\ D'_{21} & = & \{(+,0)\} & \text{(from Fig 3.4)} \\ c_{11} & = & \{(+,0),(*,+)\} & \text{(from Eqn 3.2)} \end{array}$$

$\pi_1(c_{11}) = \{(0,+),(+,*)\}$, so the minimum depth at which the statements can be separated is 2.

If both statements are completely sequential (i.e., their synchronization costs are at least n^d, where d is the depth of the loop nest), then their permutations are not considered incompatible. If only one of the statements is completely sequential, then the permutations will be considered compatible if and only if the depth at which the distributed loop of the parallel statement occurs, is greater than the minimum depth at which the statements can be separated from one another. If neither statement is completely sequential, then the permutations are considered compatible if and only if:

- both distributed loops occur at depths greater than the minimum depth at which the statements can be separated, or
- both distributed loops occur at the same depth and have the same synchronization costs.

3.3.4 The Search Problem

The time mapping selection problem can now be represented as a weighted graph. The graph will contain a node corresponding to each candidate loop permutation of each statement. The candidate loop permutations are those that achieve the minimal degree of synchronization possible for the space mappings chosen in the previous section. If the zero space mapping, or the pessimistic version of a candidate space mapping is chosen, then all legal loop permutations become candidates. The node weights will be the data locality costs as derived in Section 3.3.2 and the edge weights will be zero or infinity, depending on whether or not the candidate permutations are incompatible as defined in Section 3.2.5. The same algorithm is used to solve this graph problem as is used to solve the space mapping selection problem (see Section 3.4).

3.3.5 Constant Levels and Offsets

Given a loop permutation for each statement, I now need to create a time mapping for each statement. In addition to representing the loop permutations, the time mappings must also represent the loop level at which statements should be separated from one another, and how they should align with one another in the loops they do share.

Given a permutation, π_p, for statement p, I generate a time mapping of the following form:

$$T_p : \{[i_1, \ldots, i_m] \rightarrow [c_p^0, i_{\pi_p^1} \pm d_p^1, c_p^1, i_{\pi_p^2} \pm d_p^2, \ldots, i_{\pi_p^m} \pm d_p^m, c_p^m]\}$$

where π_p^j is the number of the loop in position j according to the given permutation π_p, and $c_p^0, \ldots c_p^m$, $d_p^1, \ldots, d_p^m$ are integer constants. If I want statements p and q to be in the same loops up to loop depth d and for statement p to come before statement q, then I will select constants such that $c_p^0 = c_q^0, \ldots, c_p^{d-1} = c_q^{d-1}$ and $c_p^d < c_q^d$ (these constants correspond to loop fusion/fission and statement reordering transformations). Similarly, if I want iteration x of statement p to occur in the same iteration of the level d loop as iteration $x + \delta$ of statement q, then I will select constants such that $d_p^d = d_q^d + \delta$ (these constants correspond to loop alignment transformations [ACK87]).

My current implementation separates statements at the minimum loop depth possible. This allows the maximal degrees of parallelism to be exploited for all statements. In some cases it may be possible and in fact preferable to separate statements at a depth greater that the minimal allowable. Whether or not this is preferable will depend on, amongst other things, the way the statements interact with one another via the cache. In some cases, separating statements at a deeper level will improve cache reuse, but in others it will degrade cache reuse. Since it is difficult to predict when separating at a deeper level will improve performance, and since the potential gain from doing so is small, I decided to always separate at the minimum depth possible. This is not a limitation of the overall framework, which easily could be altered to select another depth at which to separate.

The actual algorithm for selecting constants is as follows. I maintain a data dependence graph with each node corresponding to a statement and store a tuple relation, D_{pq}, with each directed edge from p to q, representing the data dependences from statement p to statement q. A topological sort is performed on reduced graph and a constant c_p^0 is assigned to each statement p according to the position of the connected component that it belongs to. The dependence graph is then updated so that only dependences between statements in the same connected component remain:

$$D_{pq} = D_{pq} \cap \{i \rightarrow j | c_p^0 = c_q^0\}$$

I then move on to selecting the first set of constant offsets. If:

$$S_p = \{[i_1, \ldots, i_m] \rightarrow [i_{\pi_p^1} + \delta]\}$$

then d_p^1 is set equal to δ, otherwise d_p^1 is set equal to 0. This selection of constant offsets makes code generation easier because the index variable of the new distributed loop will be synonymous with the virtual processor number.

It is possible that this selection of constant offsets will not result in a legal set of time mappings. To determine if this is the case, I test for each pair of statements p and q, whether:

$$\pi_p^1(i) + d_p^1 \leq \pi_q^1(j) + d_q^1 \tag{3.3}$$

for all $i \rightarrow j$ remaining in D_{pq}.

This test has never failed in any of the examples I have looked at; I have therefore been able to avoid handling that situation in my current implementation. This test usually succeeds, however, only because I separate statements (and hence remove as many dependences as possible) as early as possible. If I changed this policy, then alignment would become a much bigger problem. To solve the problem completely, it would be necessary to implement an algorithm to select constant offsets that properly align the statements, such as the algorithm I developed in [KP93]. The problem with that approach is that it may be too slow to use for large programs. Another approach is to backtrack in the search procedure to find a different set of space mappings or time mappings that don't need to be specially aligned. Such a set of time and space mappings always exists since the time mappings corresponding to the original program's execution order do not require any alignment.

If the given selection of constant offsets does satisfy condition 3.3, then I again update the dependence graph by removing any dependences that are now guaranteed to be satisfied by this choice of loop permutations and constant offsets:

```
    out real a[1024,1024]
    for i = 1, 1024 do
        for j = 1, 1024 do
1           a[i,j] = ...
        endfor
    endfor
    for i = 1, 1024 do
2       a[i,i] = ...
    endfor
    for k = 1, 1024 do
3       a[k,k] = sqrt(a[k,k])
        for i = k+1, 1024 do
4           a[i,k] = a[i,k]/a[k,k]
            for j = k+1, i do
5               a[i,j] = a[i,j]-a[i,k]*a[j,k]
            endfor
        endfor
    endfor
```

Figure 3.15: Cholesky decomposition

$$D_{pq} = D_{pq} \cap \{i \to j | \pi_p^1(i) + d_p^1 = \pi_q^1(j) + d_q^1\}$$

This process continues, alternating between selection of the constant levels using topological sort and selection of constant offsets using the rules described above.

For example, consider the Cholesky decomposition program shown in Figure 3.15. The selected space mappings and loop permutations are:

$$\begin{array}{llllll}
S_1: & \{[i,j] \to & [i]\} & \qquad & \pi_1: & (i) \\
S_2: & \{[i] \to & [i]\} & & \pi_2: & (i) \\
S_3: & \{[k] \to & [k]\} & & \pi_3: & (k) \\
S_4: & \{[k,i] \to & [i]\} & & \pi_4: & (k,i) \\
S_5: & \{[k,i,j] \to & [i]\} & & \pi_5: & (j,i,k)
\end{array}$$

The initial dependence graph is shown in Figure 3.16(a). Applying topological sort to this graph produces the following constants: $c_1^0 = 1,\ c_2^0 = 2,\ c_3^0 = 3,\ c_4^0 = 3,\ c_5^0 = 3$. The updated dependence graph is shown in Figure 3.16(b).

For statements 1,2 and 3, $S_p = \{[i_1, \ldots, i_m] \to [i_{\pi_p^1} + \delta]\}$, with $\delta = 0$, so $d_1^1 = 0$, $d_2^1 = 0$ and $d_3^1 = 0$. This is not true for the other statements, so $d_4^1 = 0$ and $d_5^1 = 0$. The updated dependence graph is shown in Figure 3.16(c).

Applying topological sort to each connected component in this graph produces the following constants: $c_1^1 = 1,\ c_2^1 = 1,\ c_3^1 = 2,\ c_4^1 = 3,\ c_5^1 = 1$. The updated dependence graph is shown in Figure 3.16(d). The time mappings for statements 2 and 3 are now complete.

For statements 4 and 5, $S_p = \{[i_1, \ldots, i_m] \to [i_{\pi_p^2} + \delta]\}$, with $\delta = 0$, so $d_4^2 = 0$ and $d_5^2 = 0$. This is not true for statement 1, so $d_1^2 = 0$. The updated dependence graph is shown in Figure 3.16(e).

Applying topological sort to each connected component in this graph produces the following constants: $c_1^2 = 1,\ c_4^2 = 1,\ c_5^2 = 1$. The updated dependence graph is shown in Figure 3.16(f). The time mappings for statements 1 and 4 are now complete.

It is not the case that $S_p = \{[i_1, \ldots, i_m] \to [i_{\pi_p^3} + \delta]\}$, for statement 5, so $d_5^3 = 0$. The updated dependence graph is shown in Figure 3.16(g). Applying topological sort to each connected component in this graph produces the following constants: $c_5^2 = 1$.

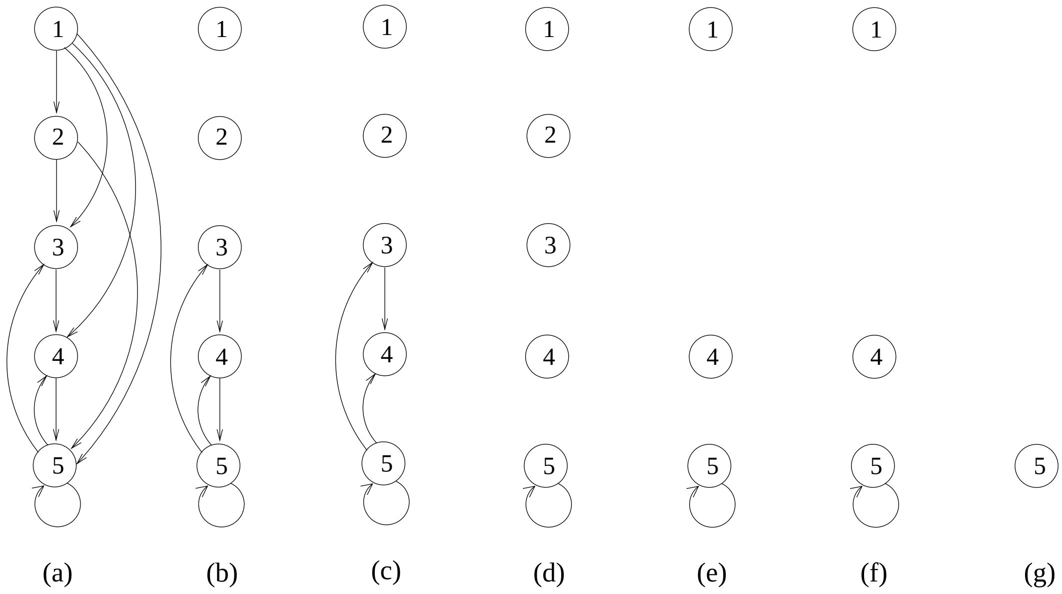


Figure 3.16: Dependence graph

The final time mappings are:

$$\begin{array}{llllllllllll}
T_1: & \{[& i, & j & &] & \rightarrow [& 1, & i, & 1, & j, & 1 & & &]\} \\
T_2: & \{[& i & & &] & \rightarrow [& 2, & i, & 1 & & & & &]\} \\
T_3: & \{[& k & & &] & \rightarrow [& 3, & k, & 2 & & & & &]\} \\
T_4: & \{[& k, & i & &] & \rightarrow [& 3, & k, & 3, & i, & 1 & & &]\} \\
T_5: & \{[& k, & i, & j &] & \rightarrow [& 3, & j, & 1, & i, & 1, & k, & 1 &]\}
\end{array}$$

3.4 The Search Procedure

3.4.1 The Simple Search Procedure

The basic approach to solving the search problem is simple; an exhaustive search is performed through all possible selections of nodes and the one with the lowest overall cost is chosen. In order to solve the problem in a feasible amount of time, I have developed a number of effective but optimality-preserving, branch and bound style pruning strategies. Figure 3.17 shows a simplified version (without any optimizations) of the recursive depth-first search algorithm. C_k is the list of nodes for statement k, N is the number of statements, $v_i(s)$ is the node cost for candidate s of statement i, and $e_{ij}(s_i, s_j)$ is the edge cost between statements i and j using candidates s_i and s_j respectively.

3.4.2 Pruning Strategies

Once the cost of at least one solution has been determined, its cost can be used to prune the search space. Suppose a solution with cost c has been found and I am currently considering a partial solution defined by a function S from some subset of the statements to the candidates currently being considered for those statements. I define:

$$P(S) = \sum_{i \in domain(S)} v_i(S[i]) + \sum_{j \in domain(S) \wedge j \leq i} e_{ij}(S[i], S[j])$$

```
search (statement k)
    foreach candidate s ∈ C_k
        S[k] = s
        if (k < N)
            search(k + 1)
        else
            cost = Σ_{i=1}^{N} (v_i(S[i]) + Σ_{j=1}^{i} e_ij(S[i], S[j]))
            if (cost < best_cost)
                best_cost = cost
                record {S[1], ..., S[N]} as best

Start by calling search(1)
```

Figure 3.17: Simplified search algorithm

If $P(S) \geq c$, then consideration of the partial solution can be terminated, since its total cost cannot be better than that of the solution already found.

Further pruning can be performed if I can determine a lower bound on the cost that will be contributed by those statements for which a candidate has not yet been chosen (i.e., without actually considering all combinations of selections and choosing the best one). The lower bound that I use is:

$$lb(S) = \sum_{i \notin domain(S)} \min_{s \in C_i} (P(S \cup \{i \rightarrow s\}) - P(S))$$

That is, $lb(S)$ is the sum of the edge costs from all statements for which a candidate has been chosen, to the best candidates of each of the statements for which a candidate has not been chosen, plus the node costs of each of these candidates. If $P(S) + lb(S) \geq c$ then consideration of the partial solution can be terminated.

Given that low-cost solutions allows me to prune more than high-cost solutions, it is advantageous to find low cost solutions early in the search process. The candidates of each statement can be considered in any order, so I choose an order that is most likely to lead to a complete low cost solution as early as possible. For each unselected statement i, the candidates s are ordered according to $P(S \cup \{i \rightarrow s\})$.

The statements can also be considered in any order. The best candidate for each statement will usually have to be considered with many other combinations of candidates for the remaining statements. It is highly desirable, however, if the candidates other than the best candidate, are not considered with many other combinations of candidates for the remaining statements. If selecting the second-best candidate of a statement will cause the total cost to rise substantially then only a few more statements (if any) will have to be considered before the total cost rises to a point where the partial solution can be pruned. So, when selecting the statement to explore next, I choose the one whose second best candidate will add the most to the total cost; that is, the statement i whose second best candidate s is most expensive according to:

$$P(S \cup \{i \rightarrow s\}) + lb(S \cup \{i \rightarrow s\})$$

Note that there is no fixed order in which the statements are considered. At each stage, the statement considered next will depend on the current context.

According to the above formula, deciding which statement to consider next would take approximately $O(N^4M^2)$ time, where M is the average number of candidates per statement. By storing partial sums, this can be reduced to $O(NFM^2)$, where F is the average number of statements that have value-based flow dependences reaching a statement. While this is still

Program	Nr Stmt	Max Nest	Un-optimized Calls	Un-optimized Time	Optimized Calls	Optimized Time
ge	2	3	4	0.00	2	0.00
ch	3	3	9	0.00	2	0.00
relax	1	3	1	0.00	1	0.00
jacobi	3	3	6	0.00	3	0.00
burg2	11	2	17839	0.99	11	0.01
lczos	23	3	7132375	563.93	23	0.02
cholsky	14	4	458656	69.04	14	0.02
mxm	2	3	4	0.00	2	0.02
adi	17	3	2.3×10^7	$\sim$ 20 minutes	17	0.02
intbal	41	2	7.1×10^7	$\sim$ 1 hour	41	0.03
eflux	27	3	2.9×10^{11}	$\sim$ 6 months	27	0.04
vpenta	53	2	3.7×10^{16}	$\sim 10^5$ years	53	0.17
shallow	65	2	4.6×10^{14}	$\sim$ 700 years	90	0.18
erle	60	3	3.6×10^{21}	$\sim 10^{10}$ years	105	0.30

(All times are in seconds unless otherwise specified)

Table 3.3: Space mapping search times for various benchmark programs

relatively expensive, it more than pays for itself by substantially increasing the amount of pruning. In my experiments I have not found the cost prohibitive and the theoretical exponential worst-case behavior has not been seen in practice.

Table 3.3 shows the number of recursive calls to the search procedure, and the overall execution time of the search procedure when used to select space mappings for a variety of benchmark programs. Data is shown for both the un-optimized version of the search algorithm (Figure 3.17) and the fully optimized pruning search algorithm. Times marked with a $\sim$ are projected times. The execution times for the un-optimized search algorithm clearly grows at an exponential rate, however, the fact that most practical programs don't have an exponential number of "interesting" candidate solutions, means that the heuristic search algorithm's execution times grow at a much more acceptable rate.

3.4.3 Time-limited Searches

While the optimized execution times shown in Figure 3.3 are relatively low, there is no guarantee that they will be low for all programs. There are, however, ways to modify the algorithm to place an arbitrary constant upper bound on the time spent performing the search, at the expense of no longer being guaranteed an optimal solution. The entire algorithm has exponential worst case performance; however, a complete (but not necessarily optimal) solution can trivially be found in a linear amount of time. In the case of searching for space mappings, the algorithm can be terminated at any point after a complete solution has been found, with the best solution found so far being returned. In the case of searching for time mappings, the algorithm can be terminated at any point after a complete solution with a non-infinite cost (indicating that the solution is legal) has been found. This kind of termination could be triggered whenever the number of recursive calls to the search procedure increases past some fixed limit or when the system clock indicates that more than a certain amount of cpu time has been spent in the search procedure.

3.4.4 Semi-automatic Systems

The search algorithm as it has been described here is designed to automatically select a space/time mapping for each statement. It is very easy, however, to modify the algorithm for use in an environment where a human user or some outside system makes some or all of the selections of the space mappings and/or time mappings. This is useful in an interactive environment where the user may for example want to experiment with various permutations and/or distributions for the most deeply nested statements. In such a setting, they may not care, or be bothered to determine the best permutations and/or distributions for the other statements, but the system can automatically select these.

3.5 Code Generation

3.5.1 Introduction

Given a space mapping and time mapping for each statement, it is now necessary to generate the transformed code that will realize these mappings. I will be generating SPMD code written in C. This C code must then be compiled to produce executable code for the target machine.

I first generate code that takes into account the time mappings but not the space mappings; that is, the transformed code executes all iterations in lexicographic order based on their coordinates in the transformed iteration space. I then modify this code to produce SPMD code by restricting the iterations to those that belong to a particular physical processor. Finally, I insert synchronization statements to enforce any inter-processor data dependences.

3.5.2 Scanning Multiple Polyhedra

A one-to-one and onto time mapping applied to a convex iteration space will result in a convex iteration space. The problem of generating perfectly nested loops to iterate over all and only those points in such a convex region has been studied by a number of researchers starting with the seminal work of Ancourt and Irigoin [AI91].

If the original iteration space is non-convex (as a consequence of non-unit loop steps), or if the time mapping applied is not onto, then the transformed iteration space may be non-convex. In these cases it is still possible to generate suitable perfectly nested loops; however, some of the loop steps will be non-unit. Techniques for handling this case are described by Li and Pingali [LP92].

The algorithm I describe in this section addresses the more general case, where a potentially different time mapping is used for each statement. The corresponding transformed iteration space can be "very" non-convex; that is, there is no set of perfectly nested loops without conditionals, even with non-unit steps, that can scan the space.

In earlier work Pugh, Rosser and I [KPR95] developed an effective code generation algorithm for this most general class of time mappings. That work included techniques to generate non-unit steps to iterate over non-convex regions and to split iteration spaces to handle the overlap of the transformed iteration spaces caused by multiple time mappings. It also included a framework to trade-off control overhead for code duplication. This algorithm and framework for reducing control overhead is useful for generating code even for transformation systems that aren't based on time mappings.

In this section I will describe a simpler, more efficient algorithm that is designed specifically for the form of time mappings selected by the algorithms described in the previous sections. The first thing to note about the form of these time mappings is that every odd level consists entirely of constants. These constant levels will lead to sequences of statements at various depths.

Consider the program and time mappings shown in Figure 3.18. The first constant level will produce a sequence of two compound statements at the outermost level, the first containing statement 1 and the second containing statements 2 and 3 (see Figure 3.19).

Code is generated for each compound statement in turn. Compound statements will generally consist of a `for` loop that iterates over a continuous range of values, containing all values

```
    do i = 1 to n
1       a[i] = 1
2       b[i] = a[i]
    do t = 0 to m
        do j = t to n
3           c[t][j] = a[t] * b[j]
```

$$\begin{array}{llllllll} T_1 : \{[i] & \rightarrow [& 0, & i, & & & 0]\} \\ T_2 : \{[i] & \rightarrow [& 1, & i-1, & & & 0]\} \\ T_3 : \{[t,j] & \rightarrow [& 1, & j, & 1, & t, & 0]\} \end{array}$$

Figure 3.18: Program and time mappings

```
for (t2=1; t2<=n; t2++)
    a[t2] = 1
for (t2=0; t2<=n; t2++)
    {
    if (t2 < n)
        b[t2+1] = a[t2+1];
    for (t4=t2; t4<=n; t4++)
        c[t4][t2] = a[t4] * b[t2];
    }
```

Figure 3.19: Transformed code

that could be taken on by any of the expressions at the next level of the time mappings for statements involved in that compound statement. That `for` loop, will contain, in turn, a sequence of compound statements based on the next constant level of those statements.

For example the first compound statement contains only statement 1 which has the expression i at the second level of its time mapping. So, the first compound statement will be a `for` loop which iterates from 1 to n. The second compound statement contains statements 2 and 3 which have the expressions $i-1$ and j respectively at the second level of their time mappings. The expression $i-1$ can take on the values 0 through $n-1$ and the expression j can take on the values 0 through n (when t = 0). So, the second compound statement will be a `for` loop which iterates from 0 to n.

The body of this second `for` loop will in turn be a sequence of two compound statements, the first containing statement 2 and the second containing statement 3. The loop bounds for the loop in the second of these compound statements are derived from the expression t in the time mapping for statement 3. In this case, I want the loop to iterate over the values j through n rather than 0 through n, because in this context I am generating code inside the j loop, and so the value of j can be considered a constant. In fact, the new outermost loop of statement 3 (the one that iterates form 0 to n), may not use the variable j as its index variable, since the same loop must also be used to iterate over all possible values of the expression $i-1$ for statement 2. For this reason, I instead use tL as the index variable for level L of the transformed iteration space.

The guard $(t2 < n)$ is placed around statement 2 because it should not be executed for all values of $t2$ (representing $i-1$) in the range 0 through n.

The actual code generation algorithm is given in Figures 3.20 and 3.21. The mutually recursive procedures gen_constant_level and gen_variable_level are responsible for generating code corresponding to the odd and even levels of the new iteration space respectively. The

```
procedure gen_code()
    gen_constant_level(1, {1, ..., n}, True)

procedure gen_constant_level(level L, set_of_statements active,
                                        tuple_set known)
    print("{")
    for posn = min(c_0^L, ..., c_n^L) to max(c_0^L, ..., c_n^L)
        new_active = {p | p ∈ active ∧ c_p^L = posn}
        new_known = known ∩(t_L = posn)∩
                    gen_guard(L, new_active, known)
        if (L < max{output_dimension(T_p) | p ∈ new_active})
            gen_variable_level(L+1, new_active, new_known)
        else
            gen_assignment(new_active)
    print("}")

procedure gen_variable_level(level L, set_of_statements active,
                                        tuple_set known)
    index_set = combine_new_IS(L, active, known)
    new_known = known ∩ gen_for_loop(L, index_set)
    gen_constant_level(L+1, active, new_known)

function gen_guard (level L, set_of_statements active,
                                tuple_set known) : tuple_set
    S = combine_new_IS(L-1, active, known)
    if (S is not a tautology)
        print("if (")
        for each constraint c in S
            print(c)
            if (c is not last constraint in S) print("&&")
        print(")")
    return S
```

Figure 3.20: Code generation algorithm

```
function gen_for_loop(level L, tuple_set index_set) : tuple_set
    new_known = False
    print("for (t" L " = max (")
    for (each lower bound constraint (expr ≤ m t_L) in index_set)
        print("(" expr "+" m-1 ")/" m)
        new_known = new_known ∩ (expr ≤ m t_L)
    print("); t" L "<=min(")
    for (each upper bound constraint (m t_L ≤ expr) in index_set)
        print("(" expr ")/" m)
        new_known = new_known ∩ (m t_L ≤ expr)
    print("; t" L "++)")
    return new_known
function combine_new_IS(level L, set_of_statements active,
                               tuple_set known) : tuple_set
    foreach statement p ∈ active
        new_IS[p] = Range(Restrict_Domain(T_p, I_p))
        proj_new_IS[p] = project(new_IS, {t_1, ..., t_L})
    return gist(hull({proj_new_IS[p] | p ∈ active}), known)

procedure gen_assignment(set_of_statements active)
    p = only active statement
    for each IndexPosition i
        replace i_j by t_2x − d_p^2x in original statement p, where π_p^x = i_j
    print(modified original statement p)
```

Figure 3.21: Code generation algorithm continued

actual code generation procedure, `gen_code,` simply calls `gen_constant_level` to generate a sequence of compound statements at the outermost level and the bodies of those statements recursively.

A set of statements, called the `active` set, is maintained that represents the set of statements contained in the current compound statement. Initially, all n statements are in this set.

The current context, or `known` information is also maintained. This represents constraints known about index variables of the outer levels, and is used to simplify loop bounds and guards generated at inner levels. The known information is usually initialized to $True$, although any user or system generated assertions about global variables could be included if desired.

When generating loop bounds and guards, I first determine the new iteration spaces of the statements inside that loop or conditional. The new iteration space of statement p is:

`Range(Restrict_Domain(`T_p`, `I_p`))`

where T_p is the time mapping selected for statement p, I_p is the original iteration space of statement p and `Range` and `Restrict_Domain` are relational operations as defined in Section 2.1. The constraints in I_p must be a complete description of the iteration space for statement p. This implies that all loop bounds and conditionals in the original program must be affine functions of outer level index variables and symbolic constants. Before constructing these tuple sets, I normalize all loops in the original program to have a step of 1. This coupled with the fact that all coefficients used in the time mappings are unary, implies that the transformed program will not contain non-unit steps which in turn substantially simplifies the algorithm.

When generating code at level L, I want the loop bounds or guards to be affine functions of symbolic constants and index variables from earlier levels. The function `combine_new_IS` therefore projects the new iteration spaces onto $(t_1, \ldots, t_{L-1})$. Since I want the loops and guards to include all of the iterations of all of the statements in the current active set, the `hull` operation is used to combine their iteration spaces. Hull is a relational operation that takes as arguments a finite set of tuple sets and returns a tuple set that includes all tuples in these sets. The input tuple sets must be *convex* (as my iteration spaces always are) and the result (by definition of the `hull` operation) will also be convex. A tuple set is convex if it can be represented as a conjunction of affine equality and inequality constraints.

The `gist` operator [PW92, Won95] is used to simplify a set of constraints given that some other set of constraints is known to be true. The only guaranteed property of the `gist` operator is:

$$\text{gist}(A,\ B) \wedge B \Leftrightarrow A \wedge B$$

However, the techniques used to compute `gist` are designed to produce simple (often minimal) sets of constraints for which this condition holds. For example $\texttt{gist}(1 \leq j \leq n \wedge j \leq i,\ 1 \leq i \leq n)$ is $1 \leq j \leq i$.

The projection operation has a number of applications in code generation other than those described here, for example it can be used to generate the set of array elements that need to be sent from one processor to another. The gist operation also has many code generation related applications other than its use in `combine_new_IS`.

Finally, the actual assignment statements have to be generated. The assignment statements have the same form as in the original code, except that the original index variables are replaced by expressions involving the new index variables. For example, if the time mapping is $\{[i,j] \rightarrow [0, j+1, 0, i, 0]\}$ then i is replaced by t_4 and j is replaced by $t_2 - 1$.

3.5.3 Generating SPMD Code

I will now describe how to modify the code produced in the previous section to produce SPMD code by restricting the iterations to those that belong to a particular physical processor. The space mapping selection algorithm described in Section 3.2.3 determines whether it is better to use a block or cyclic distribution of virtual processors to physical processors.

Blocked Distributions

If a block distribution is used then each physical processor will first need to compute two quantities, the lowest and highest numbered virtual processors that map to that physical processor, represented by variables `lb` and `ub`. To compute these quantities, it is first necessary to compute the lowest and highest numbered virtual processors that are used in the entire computation. The set of virtual processors used by statement p is given by:

$$V_p = \texttt{Range}(\texttt{Restrict\_Domain}(S_p,\ I_p))$$

The lowest and highest numbered virtual processors used in the entire computation are therefore computed as:

$$\begin{array}{l} global\_lb = min\{i \mid \exists p \text{ s.t. } i \in V_p\} \\ global\_ub = max\{i \mid \exists p \text{ s.t. } i \in V_p\} \end{array}$$

For example, if:

$$\begin{array}{l} I_1 : \{[i] \mid 1, m \leq i \leq 10, n\} \\ I_2 : \{[i,j] \mid 1, m \leq i \leq 10, n \wedge 1 \leq j \leq p\} \\ S_1 : \{[i] \rightarrow [i]\} \\ S_2 : \{[i,j] \rightarrow [j-1]\} \end{array}$$

then

$$\begin{array}{l} V_1 : \{[v] \mid 1, m \leq v \leq 10, n\} \\ V_2 : \{[v] \mid 0 \leq v \leq p-1\} \end{array}$$

and the computation would be:

```
global_lb = 0;
global_ub = max(min(n,10),p-1);
```

Each processor's lower and upper bound on virtual processors can then be easily computed as:

```
block_size = max(0,(global_ub-global_lb+1+(nprocs-1))/nprocs);
lb = global_lb+my_id*block_size;
ub = min(global_ub,lb+block_size-1);
```

where `nprocs` and `my_id` are variables initialized by the runtime system to be the total number of physical processors and the number of the physical processor that is executing the current thread respectively.

In Section 3.3.5, I conveniently selected the constant offset for the time mapping expression that contains the distributed loop to be the same as the constant offset for the space mapping of that statement. This means that the index variable at the level at which the statement is distributed can be used as the virtual processor number. So, if statement p is distributed at level L then the code needs to be modified to enforce the constraint that $lb \leq t_L \leq ub$.

If all of the other statements in the same loop as statement p at level L are also distributed at level L then the level L loop itself can be directly modified. I replace a loop of the form:

```
for (tL=max(a_1,...a_r); tL<=min(b_1,...b_s); tL++)
```

by:

```
for (tL=max(a_1,...a_r,lb); tL<=min(b_1,...b_s,ub); tL++)
```

Otherwise, the representation of this constraint will have to be delayed to the minimum level at which all other statements in p's compound statement at that level are all distributed at the same level (such a level will always exist because every statement is eventually in a compound statement by itself). At this level, a guard of the following form is inserted:

```
if (lb <= tL && tL <= ub)
```

Cyclic Distributions

Quantities analogous to `lb` and `ub` are not required when generating cyclic code. In the cyclic case, if statement p is distributed at level L then the code needs to be modified to enforce the constraint that $(tL \bmod nprocs) = my\_id$. The method used to determine the level at which this constraint can be enforced is identical to the method used in the blocked case. If the constraint can be enforced by modifying the level L loop itself, then it is replaced by:

```
for (tL = adjust(max(a_1,...,a_r));
     tL <= min(b_1,...,b_s);
     tL += nprocs)
```

where `adjust(s)` is defined as `max(s, (s/nprocs)*nprocs + my_id)`. Otherwise, the following guard is inserted at a later level:

```
if (tL % nprocs == my_id)
```

For example, consider generating code for the Cholesky decomposition program in Figure 3.15 using the following space and time mappings:

$$
\begin{array}{lll}
S_1: & \{[i,j] \rightarrow & [i]\} \\
S_2: & \{[i] \rightarrow & [i]\} \\
S_3: & \{[k] \rightarrow & [k]\} \\
S_4: & \{[k,i] \rightarrow & [i]\} \\
S_5: & \{[k,i,j] \rightarrow & [i]\}
\end{array}
$$

```
for (t2 = 1; t2 <= 1024; t2++)
    {
    if (2 <= t2)
        for (t4 = adjust(t2); t4 <= 1024; t4 += _my_nprocs)
            for (t6 = 1; t6 <= t2-1; t6++)
                a[t4][t2] = a[t4][t2]-a[t4][t6]*a[t2][t6];
    if (t2 % _my_nprocs == _my_id)
        a[t2][t2] = sqrt(a[t2][t2]);
    ...
    }
```

Figure 3.22: Sample SPMD code

$$
\begin{array}{lllllllllll}
T_1: & \{[\ i, & j &] & \rightarrow [\ 1, & i, & 1, & j, & 1 & &]\} \\
T_2: & \{[\ i & &] & \rightarrow [\ 2, & i, & 1 & & & &]\} \\
T_3: & \{[\ k & &] & \rightarrow [\ 3, & k, & 2 & & & &]\} \\
T_4: & \{[\ k, & i &] & \rightarrow [\ 3, & k, & 3, & i, & 1 & &]\} \\
T_5: & \{[\ k, & i, \ j &] & \rightarrow [\ 3, & j, & 1, & i, & 1, & k, \ 1 &]\}
\end{array}
$$

Statement 3 is distributed at level 2; statement 5 is in the same loop as statement 3 at level 2, but is not distributed until level 4. So, the constraint for statement 3 will not be able to be enforced until after level 3 at which point all statements in the same loop as statement 3 at that level are distributed. The code will have the form shown in Figure 3.22.

3.5.4 Inserting Synchronization

I will now describe how to insert synchronization statements to enforce any inter-processor data dependences. I use two forms of synchronization: barriers and post-wait pairs. I try to use post and wait pairs rather than barriers whenever possible, since they impose weaker constraints on the execution order, and hence allow more computation to be performed in parallel. The form of post-wait statements that I use only allow me to synchronize data dependences from lower numbered to higher numbered physical processors, so barriers are often needed as well.

To insert synchronization, I first examine all inter-processor data dependences and classify them according to:

1. Whether they go forward or backward with respect to the physical processor number to which their sources and destinations map.
2. The deepest level in the transformed iteration space at which they are carried.

The deepest level at which a dependence is carried in the transformed iteration space is the shallowest and therefore most efficient level at which to insert synchronization to enforce that dependence (the same basic idea has been used in the past to decide the outermost level at which communication statements can be legally placed [HKT91]). Synchronization inserted at deeper levels can make synchronization inserted at shallower levels redundant, so I first insert synchronization for dependences carried at deeper levels. Similarly, barrier synchronization can make post-and-wait synchronization inserted at the same level redundant, so at each level, I insert barriers and then post-and-wait synchronization.

Inserting Barriers

All backward dependences (with respect to physical processor number) carried at levels $2L$ and $2L+1$ will have to be enforced by barriers placed inside the body of loops created for level

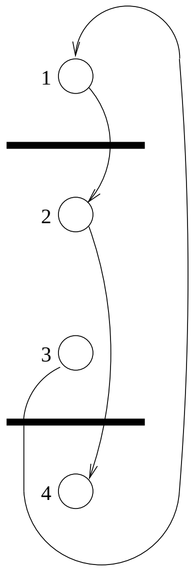


Figure 3.23: Barrier Example

$2L$ (recall that all odd levels are constant levels and will produce a sequence of statements, while even levels will generally produce loops). If a backward dependence from statement p to statement q is carried at level $2L+1$ then it must be the case that $c_p^{2L+1} < c_q^{2L+1}$ and a barrier will have to be inserted somewhere in that range of positions. If a backward dependence from statement p to statement q is carried at level $2L$ then:

- if $c_p^{2L+1} > c_q^{2L+1}$, a barrier will have to be inserted either between position c_p^{2L+1} and the end of the loop, or between the start of the loop and position c_q^{2L+1}.
- Otherwise, a barrier inserted anywhere in the loop will suffice.

It is possible for one barrier to enforce more than one dependence. In order to minimize the total number of barriers it is necessary to consider all backward dependences carried at levels $2L$ and $2L+1$ at the same time. Consider the situation illustrated in Figure 3.23. There are dependences from position 1 to position 2 and from position 2 to position 4 which are carried at level $2L+1$, and a dependence from position 3 to position 1 carried at level $2L$. If I examined the dependence from 2 to 4 in isolation, I might decide to place a barrier between statements 2 and 3. However, that would result in a total of 3 barriers being placed, whereas by examining all dependences simultaneously, it can be determined that only 2 barriers are actually required (as shown by the heavy bars in Figure 3.23).

My barrier placement algorithm works as follows. First, any dependences that are already enforced by barriers placed at deeper levels are removed from consideration. A barrier placed inside a deeper loop can only make a barrier at a higher level redundant if it can be proved that the loop always executes at least one iteration (otherwise the barrier won't be executed).

If any barriers are required inside the current loop, then an arbitrary position is tentatively selected at which to insert the first barrier. The effect of placing this first barrier is that a total order can now be assigned to the statements in the loop. Statements can be numbered in order starting immediately following the barrier. All dependences that remain to be enforced go from lower numbered to higher numbered statements. If the first barrier is placed at position b_0, then the total order is defined as $p \prec q$ if and only if:

$$(b_0 \le c_p^{2L+1} < c_q^{2L+1}) \vee (c_p^{2L+1} < c_q^{2L+1} < b_0) \vee (c_q^{2L+1} < b_0 \le c_p^{2L+1})$$

(A barrier placed at position b is actually placed between the assignment statements at positions $b-1$ and b). Intuitively, placing this first barrier splits a "ring" of statements into a "chain" of statements.

It is then easy to place the remaining barriers optimally. This problem reduces to finding a minimal clique cover for an interval graph and is known to be solvable in polynomial time [Gav72]. All ordering of statements and positions will now implicitly be with respect to the above total order. Let b_i be the position of the i^{th} barrier after b_0. In order to insert the minimal number of barriers, I need to place the i^{th} barrier as far after the $i-1^{st}$ barrier as possible. So, b_i is calculated as the maximum position such that any dependence whose source is between b_{i-1} and b_i, has a destination after b_i.

This process is repeated until all remaining dependences are enforced. At that point, the number of barriers required, given the arbitrary placement of the initial barrier, will be known. By repeating this entire process for all possible placements of the initial barrier, the overall minimal number of barriers can be determined. Only positions that enforce at least one data dependence need to be considered as initial placements. A sub-optimal placement of the initial barrier can only alter the total number of barriers inserted in a given loop by at most 1, so, the optimal solution will be known as soon as two initial placements produce a different number of barriers.

Let's see how this algorithm works for the example in Figure 3.23:

$$\begin{array}{lll} b_0 = 1 & \Rightarrow b_1 = 2,\ b_2 = 4 & : \text{3 barriers required} \\ b_0 = 2 & \Rightarrow b_1 = 4 & : \text{2 barriers required} \end{array}$$

The number of barriers differs by 1, so the second one must be an optimal solution.

The basic idea used in this algorithm can also be used for applications other than inserting barriers, for example it could be used to determine the optimal placement of blocking communication statements for distributed memory machines. See [CGC96] for a discussion of this problem.

Inserting Post-Wait Pairs

Post-wait pairs are used to enforce any forward dependences that haven't already been enforced by barriers. Any dependences that are already enforced by post-wait pairs at deeper levels are removed from consideration.

A dependence from statement p to statement q carried at level $2L+1$ will normally be enforced by inserting a post statement immediately after statement p and inserting a wait statement immediately before statement q. If, however, there is another dependence, from some statement r to some statement s, carried at the same level, such that $c_p^{2L+1} \leq c_r^{2L+1} \wedge c_s^{2L+1} \leq c_q^{2L+1}$, then enforcing the dependence from p to q would be redundant.

If there are any dependences carried at level $2L$, between the statements in a given loop, then a single pair of post and wait statements will be inserted to enforce those dependences. By default the wait statement will be inserted at the start of the loop body and the post statement will be inserted at the end of the loop body. If, however, the iterations of the level $2L$ loop are distributed in a blocked fashion then the wait statement can be hoisted to before the loop and the post statement can be hoisted to after the loop.

3.5.5 Reduction Optimizations

If the user is willing to accept inaccuracies that may result from treating machine arithmetic operations such as addition and multiplication as if they are commutative and associative operations, then a number of additional optimizations are possible. Any assignment statement of the form:

```
var_expr = var_expr ? expr
```

```
double s[m];
for (t2 = adjust(1); t4 <= n; t4 += _my_nprocs)
    for (t4 = 1; t4 <= n; t4++)
        s[t2+t4] = s[t2+t4] + a[t2][t4]
```

Table 3.4: Before reduction optimization

```
double s[m];
private double _s1[m];
init_sum_double(&_s1, m);
for (t2 = adjust(1); t4 <= n; t4 += _my_nprocs)
    for (t4 = 1; t4 <= n; t4++)
        _s1[t2+t4] += a[t2][t4]
reduction_lock(0);
reduce_sum_double(&s, &_s1, m);
reduction_unlock(0);
```

Table 3.5: After reduction optimization

where ? is a commutative and associative operator is referred to as an *update operation*. Consider two update operations, p and q (possibly the same), that update overlapping regions of the same array:

```
p: var[expr1] = var[expr1] ? ...
q: var[expr2] = var[expr2] ? ...
```

Rather than creating flow, output and anti dependences between these references to **var**, I instead create a new type of data dependence called a *reduction dependence* between these references [Wol82, PW94b]. Reduction dependences have a different semantics from other types of data dependences. Rather than specifying that one operation must be performed after some other operation, they specify that a set of operations can be performed in any order.

Reduction operations are ignored when inserting synchronization. It is important, however, to make sure that if there is a reduction dependence between two iterations executing on different processors, that they don't simultaneously write to the same location. If there are inter-processor reduction dependences between statements in a given distributed loop, but no other type of inter-processor dependences between those statements (that would cause synchronization to be inserted), then private versions of the variables being updated are created for each processor. Each processor then updates its private version of the variable rather than the original, hence avoiding different processors simultaneously writing to the same location. These private versions are initialized to the identity element for the operation in question prior to the start of the loop. After the loop each processor takes its turn to "add" its private version of the variable to the global variable.

For example, the program in Figure 3.4 is converted into the program in Figure 3.5. The function `init_sum_double(double *`v`, size_t `n`)` initializes the first n elements of array v to the value 0 and `reduce_sum_double(double *`g`,double *`l`,size_t `n`)` adds the first n elements of array l to the corresponding elements of array g.

Creating these private variables changes the amount of interprocessor communication, so the communication volume estimates used to select space mappings (see Section 3.2.5), need to be changed whenever these optimizations are going to be applied. Each self reduction dependence is given a revised volume estimate of Pn^d where P is the number of physical processors and d is the dimensionality of the array being updated. This is the amount of communication

```
          const n = 1024
          out real a(n,n)
          do i = 1, n {
            do j = 1, n {
0             a[i,j] = 1
            }
          }
          do k = 1, n {
            do i = k+1, n {
1             a(i,k) = a(i,k)/a(k,k)
              do j = k+1, n {
2               a[i,j] = a[i,j] - a[k,j]*a[i,k]
              }
            }
          }

Unbalanced, use Cyclic Distribution
S0: {[i,j] -> [i] }
S1: {[k,i] -> [i] }
S2: {[k,i,j] -> [i] }
T0: {[i,j] -> [0,i,0,j,0] }
T1: {[k,i] -> [1,k,0,i,0] }
T2: {[k,i,j] -> [1,k,1,i,0,j,0] }
```

Figure 3.24: Gaussian elimination (ge.t) with time and space mappings

that occurs when each processor takes its turn to "add" its private version of the variable to the global variable. Reduction dependences between different statements are given a revised volume estimate of 0, since both updates are to local versions of the variable. These reduction optimizations also prevent false sharing from occurring.

3.5.6 Examples

Figures 3.24, 3.25, 3.26 and 3.27 show a number of complete examples with output as generated by my current implementation.

```
static void doall_body(int _my_id) {
  int _my_nprocs = num_total_ids();
  for (int t2 = max(1,_my_id); t2 <= 1024; t2 += _my_nprocs)
    for (int t4 = 1; t4 <= 1024; t4++)
      a[t2][t4] = 1;
  for (int t2 = 1; t2 <= 1023; t2++) {
    global_barrier(0);
    for (int t4 = adjust(t2+1); t4 <= 1024; t4 += _my_nprocs)
      a[t4][t2] *= 1/a[t2][t2];
    for (t4 = adjust(t2+1); t4 <= 1024; t4 += _my_nprocs)
      for (int t6 = t2+1; t6 <= 1024; t6++)
        a[t4][t6] += -a[t2][t6]*a[t4][t2];
  }
}
```

Figure 3.25: Transformed code for Gaussian elimination (ge.c)

```
      const n = 512
      const l = 512
      out real a(n,n)
      do j = 1, n {
        do k = 1, n {
0         a[j,k] = 1
        }
      }
      do i = 1, l {
        do j = 2, n - 1 {
          do k = 2, n - 1 {
1           a[j,k] = (a[j,k-1]+a[j,k+1]+a[j-1,k]+a[j+1,k]) * .25
          }
        }
      }

Balanced, use Block Distribution
S0: {[j,k] -> [j] }
S1: {[i,j,k] -> [j] }
T0: {[j,k] -> [0,j,0,k,0] }
T1: {[i,j,k] -> [1,i,0,k,0,j,0] }
```

Figure 3.26: Red-black relaxation (relax.t) with time and space mappings

```
static void doall_body(int _my_id) {
  int _counter0 = 1;
  int _my_nprocs = num_total_ids();
  int global_lb = 1;
  int global_ub = 512;
  int block_size =
      max(0,(global_ub-global_lb+1+(_my_nprocs-1))/_my_nprocs);
  int lb = global_lb+_my_id*block_size;
  int ub = min(global_ub,lb+block_size-1);
  for (int t2 = max(1,lb); t2 <= min(512,ub); t2++)
    for (int t4 = 1; t4 <= 512; t4++)
      a[t2][t4] = 1;
  for (int t2 = 1; t2 <= 512; t2++) {
    global_barrier(0);
    for (int t4 = 2; t4 <= 511; t4++) {
      if (_my_id > 0) counter_wait(_my_id-1,0,_counter0++);
      for (int t6 = max(2,lb); t6 <= min(511,ub); t6++)
        a[t6][t4]=
            (a[t6][t4-1]+a[t6][t4+1]+a[t6-1][t4]+a[t6+1][t4])*.25;
      counter_incr(_my_id, 0);
    }
  }
}
```

Figure 3.27: Transformed code for Red-black relaxation (relax.c)

Chapter 4

Experimental Results

4.1 The Implementation

The algorithms described in the previous chapter have been implemented and are now part of the `PETIT` analysis and transformation system. `PETIT` is based on Michael Wolfe's `Tiny tool[Wol91b]` for loop restructuring research and has been extended over the last five years by myself and other members of the Omega Project (see `http://www.cs.umd.edu/projects/omega` for more information). `PETIT` accepts as input, programs written in a toy language called `Tiny`. `Tiny` programs consist of a single procedure containing `for` loops and assignment statements involving array expressions. In addition to the transformation system described here, `PETIT` has been extended to include array expansion, induction variable recognition and advanced dependence analysis using the Omega Test [Pug92, Won95]. All of these extensions have been implemented using the Omega Library [KMP+95] which is set of C++ classes for representing and manipulating tuple relations and sets.

`PETIT` and the Omega Library are freely available via anonymous ftp from `ftp://ftp.cs.umd.edu/pub/omega/omega_system`. The automatic parallelization system is invoked by using the $-W$ flag on the `PETIT` command line and produces as output the file `aux.out` which contains the transformed SPMD C code.

The rest of this chapter gives a number of experimental results obtained using this system that demonstrate that it is both efficient and effective in parallelizing small to medium sized numeric kernels.

4.2 Efficiency

This section gives experimental results to show that my system normally executes in a reasonable amount of time. Table 4.1 gives a breakdown of my system's execution times for a variety of benchmark programs. Tables 4.2, 4.3 and 4.4 give a further breakdown of these times. The times listed are in seconds and are as measured by Quantify[1] on a SPARCstation 10/51. These benchmark programs and the results that we obtain for them are available from `ftp://ftp.cs.umd.edu/pub/omega/results_KP95`.

For most programs, the largest amount of time is spent in the final code generation phase. Most of this code generation time is spent performing synchronization analysis, which is currently performed very precisely and involves many complex tuple relation operations. By sacrificing some of this precision, it would be possible to substantially speed up that part of the algorithm.

[1] Registered trademark of Pure Software Inc.

Program name	Number of statements	Max loop nest	Select space mappings	Select time mappings	Code generation	Total
mxm1	3	3	0.02	0.02	0.26	0.33
ge	3	3	0.08	0.04	0.25	0.40
relax	2	3	0.11	0.04	0.26	0.45
ch	4	3	0.10	0.07	0.40	0.62
jacobi	4	3	0.16	0.05	0.39	0.66
burg2	12	2	0.39	0.31	1.28	2.06
lczos	24	3	0.26	0.27	1.79	2.41
intba1	42	2	0.34	0.23	1.82	2.48
cholsky2	15	4	1.12	0.28	1.48	2.96
eflux	28	3	0.86	0.43	2.78	4.21
adi	17	3	1.56	0.78	2.67	5.15
shallow	66	2	1.59	0.71	4.41	6.90
erle	61	3	3.16	1.41	7.36	12.20
vpenta	54	2	3.44	1.41	9.57	14.89

(All times are in seconds)

Table 4.1: Breakdown of compilation times for various benchmark programs

Program name	Parallelism analysis	Communication analysis	Search	Alignment	Total
mxm1	0.01	0.01	0.00	0.00	0.02
ge	0.06	0.01	0.00	0.00	0.08
ch	0.08	0.01	0.00	0.00	0.10
relax	0.08	0.02	0.00	0.00	0.11
jacobi	0.12	0.03	0.00	0.00	0.16
lczos	0.11	0.11	0.02	0.01	0.26
intba1	0.11	0.17	0.02	0.02	0.34
burg2	0.25	0.08	0.02	0.02	0.39
eflux	0.34	0.44	0.04	0.02	0.86
cholsky2	0.99	0.10	0.02	0.01	1.12
adi	1.28	0.24	0.01	0.01	1.56
shallow	0.47	0.81	0.20	0.07	1.59
erle	1.32	1.24	0.49	0.08	3.16
vpenta	0.99	2.10	0.23	0.10	3.44

(All times are in seconds)

Table 4.2: Breakdown of select space mappings times

Program name	Locality analysis	Search	Alignment	Total
mxm1	0.00	0.00	0.02	0.02
relax	0.00	0.00	0.04	0.04
ge	0.00	0.00	0.04	0.04
jacobi	0.00	0.00	0.05	0.05
ch	0.00	0.00	0.07	0.07
intba1	0.00	0.01	0.21	0.23
lczos	0.00	0.01	0.26	0.27
cholsky2	0.00	0.01	0.26	0.28
burg2	0.00	0.00	0.30	0.31
eflux	0.00	0.04	0.32	0.43
shallow	0.00	0.09	0.56	0.71
adi	0.00	0.00	0.76	0.78
erle	0.01	0.20	0.90	1.41
vpenta	0.00	0.09	1.24	1.41

(All times are in seconds)

Table 4.3: Breakdown of select time mappings times

Program name	Synchronization analysis	Reduction analysis	Generate code	Total
ge	0.11	0.00	0.13	0.25
mxm1	0.06	0.00	0.19	0.26
relax	0.15	0.00	0.11	0.26
jacobi	0.22	0.00	0.16	0.39
ch	0.16	0.00	0.23	0.40
burg2	0.73	0.00	0.53	1.28
cholsky2	0.49	0.00	0.97	1.48
intba1	0.57	0.00	1.21	1.82
lczos	0.60	0.01	1.16	1.79
adi	1.75	0.00	0.90	2.67
eflux	1.00	0.00	1.74	2.78
shallow	1.90	0.01	2.33	4.41
erle	3.59	0.02	3.52	7.36
vpenta	5.80	0.01	3.61	9.57

(All times are in seconds)

Table 4.4: Breakdown of code generation times

Program name	Problem size	Number of processors 1	2	4	8	12	16
mxm1	512 x 512	8.4	4.3	2.2	1.5	1.2	1.2
ge	1024 x 1024	33.6	30.1	20.3	18.2	19.8	15.9
relax	512 x 512	26.8	14.5	8.3	5.8	4.8	8.3
ch	1024 x 1024	27.7	20.2	11.1	6.1	4.5	3.6
jacobi	512 x 512	15.4	8.3	6.0	5.1	4.8	4.1
burg2	16384	50.7	35.4	20.1	12.3	9.9	8.8
cholsky	256 x 256 x 64	12.4	12.4	12.3	5.1	2.5	1.9
intbal	1024 x 1024	8.2	4.5	2.7	2.2	1.7	1.8
lczos	512 x 512	8.4	5.1	2.8	5.0	7.2	15.4
adi	1024 x 1024	133.1	64.6	28.2	12.8	8.9	7.2
eflux	512 x 512	9.4	4.0	1.9	1.4	1.5	1.7
shallow	1024 x 1024	5.1	3.0	2.0	1.7	1.7	1.7
vpenta	1024 x 1024	6.6	4.1	3.1	3.2	5.5	58.9
erle	256 x 256 x 256	42.4	23.9	15.7	11.1	10.0	9.5

(All times are in seconds)

Table 4.5: Execution times using my parallelization system

4.3 Effectiveness

The machine that I used for my experiments was a 16 processor SGI POWER CHALLENGE. It has a shared-memory multiprocessor architecture based on the MIPS super-scalar RISC R8000 chip. The cache system consists of a 16 kilobyte direct-mapped on-chip integer data cache, and a 4 megabyte four-way set associative external cache. The processors communicate via a cache-coherent shared-bus interconnect with a bandwidth of 1.2 gigabytes per second. See `http://www.ncsa.uiuc.edu/Pubs/UserGuides/Power` for more details.

Each of the benchmark programs listed in Table 4.1 were parallelized using both my system and the parallelizing C compiler native to the SGI POWER CHALLENGE (based on Kuck and Associates' parallelizing compiler). All programs were then compiled using the native C compiler "cc" with the following options:

–O3 turns on aggressive optimization including standard software pipelining
–32 generate 32 bit object code for compatibility with SUIF runtime library assembly code
–sopt invoke extra scalar optimization pass which performs loop unrolling and other miscellaneous optimizations
–r=3 sets tolerable roundoff error to maximum
–o=5 sets general optimization level to maximum
–so=5sets serial optimization level to maximum

The resulting executables were executed using 1, 2, 4, 8, 12 and 16 processors. The wall clock execution times measured on a dedicated machine are shown in Tables 4.5 and 4.6 respectively. The speedups for some of these programs are graphed in Figures 4.1 through 4.9. The speedups are based on the minimum of the execution times obtained by my system and the native parallelizing compiler when executed on a single processor.

The native parallelizing C compiler produces `doall` loops and so cannot express pipeline style parallelism and is unable to control communication between parallel loops nests. My system generally performs more dramatic iteration reordering transformations than the native parallelizing compiler and made more use of array privatization.

Program name	Problem size	Number of processors 1	2	4	8	12	16
mxm1	512 x 512	9.8	5.5	3.3	2.7	4.4	3.8
ge	1024 x 1024	39.5	17.7	12.0	8.8	6.5	7.2
relax	512 x 512	49.3	46.7	47.4	47.9	48.3	48.5
ch	1024 x 1024	26.8	20.9	14.5	13.0	9.6	16.1
jacobi	512 x 512	23.9	13.0	8.4	5.8	5.1	6.5
burg2	16384	27.9	20.1	12.3	8.2	8.8	15.9
cholsky	256 x 256 x 64	27.6	44.3	52.9	56.6	70.8	190.6
intba1	1024 x 1024	7.9	7.5	7.7	9.3	9.3	9.5
lczos	512 x 512	6.1	4.1	3.4	3.8	3.9	5.7
adi	1024 x 1024	77.1	44.2	25.7	19.1	13.0	16.9
eflux	512 x 512	4.6	3.1	2.3	2.9	8.7	5.9
shallow	1024 x 1024	5.7	4.7	4.4	4.5	4.5	4.7
vpenta	1024 x 1024	3.7	3.0	2.9	3.3	4.6	4.7
erle	256 x 256 x 256	38.4	26.2	20.3	15.9	15.4	15.1

(All times are in seconds)

Table 4.6: Execution times using native parallelizing C compiler

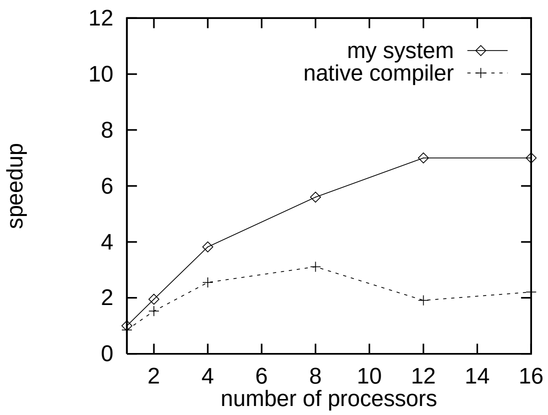


Figure 4.1: Speedup graph for mxm1

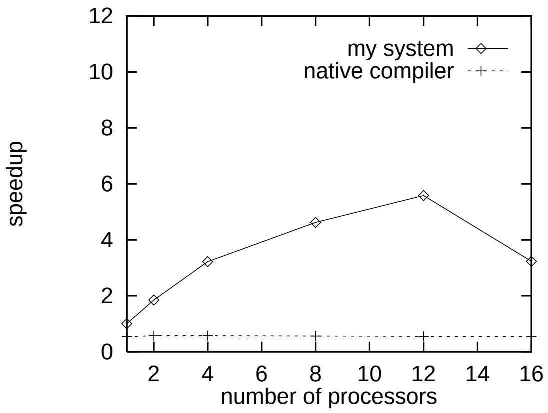


Figure 4.2: Speedup graph for relax

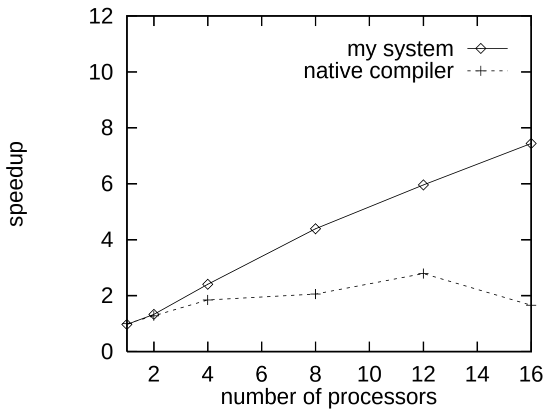


Figure 4.3: Speedup graph for ch

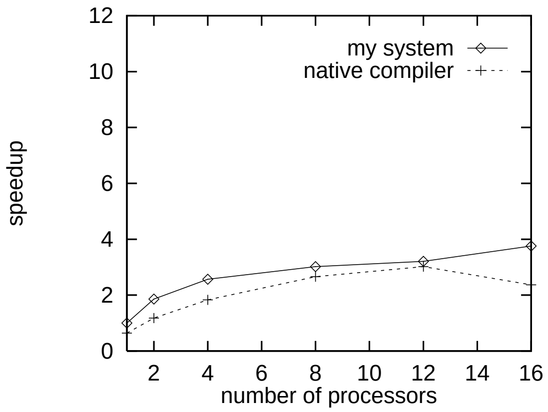


Figure 4.4: Speedup graph for jacobi

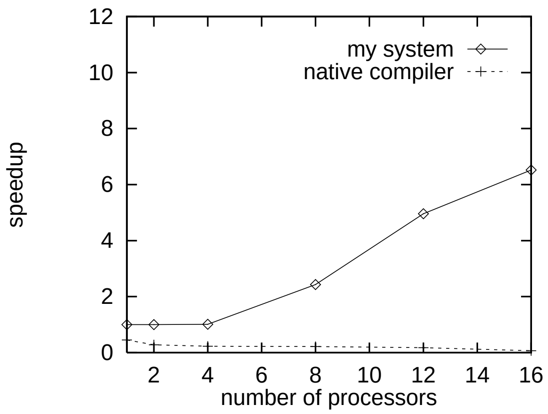


Figure 4.5: Speedup graph for cholsky

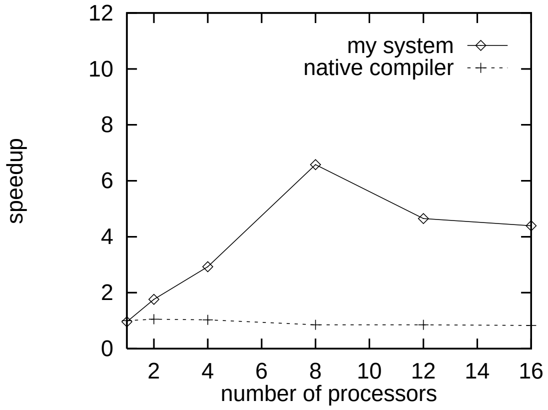


Figure 4.6: Speedup graph for intba1

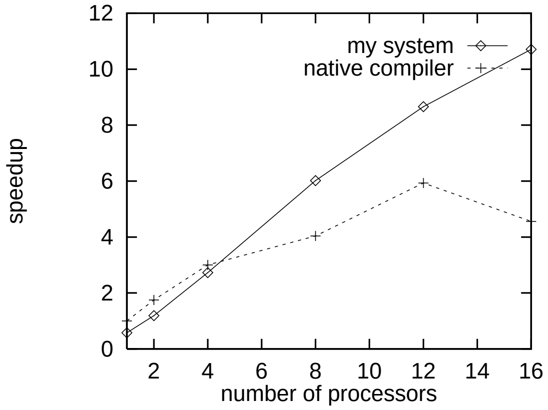


Figure 4.7: Speedup graph for adi

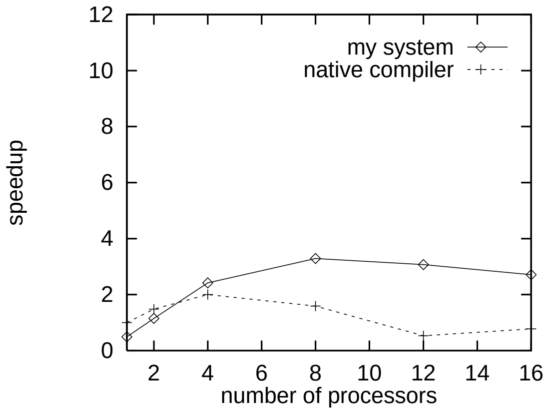


Figure 4.8: Speedup graph for eflux

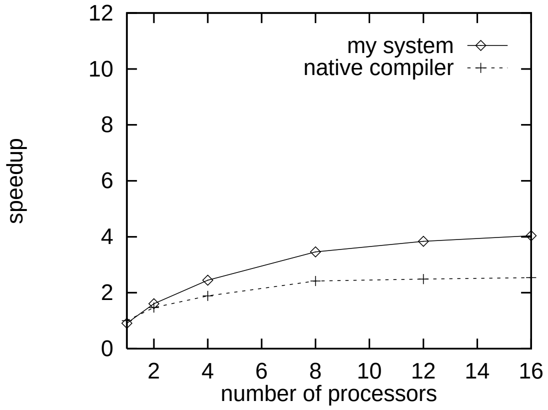


Figure 4.9: Speedup graph for erle

4.4 Robustness

Many papers have been written about automatic data decomposition and iteration reordering. Most contain examples to show the performance of their respective algorithms. Whilst the papers themselves contain impressive results, I have found that in the few implementations that I have been able to experiment with, most of these algorithms are very fragile. That is, the programs as given in these papers can be compiled very efficiently, but minor, semantic-preserving changes to these programs (such as performing loop interchange, loop fusion or statement reordering), often result in completely different and often far from optimal programs.

My aim was a system that produces the same result (hopefully an optimal result) regardless of the form in which the program is originally presented. In Figures 4.10 through 4.15, I demonstrate this aspect of my system by showing the space and time mappings selected by my system for all six legal loop permutations for Cholesky decomposition. For this example, identical code is produced by my system, for all six permutations (see Figure 4.16). I have not found any other system that is able to reproduce these results. I also produce identical code for all 6 legal permutations of Gaussian elimination, and for various loop restructurings of adi. In fact, my system is *guaranteed* to produce the same space mappings if the transitive dependences can be calculated exactly. If extended direction vectors are used, then the calculations may not be exact, although I have observed this phenomenon only in the presence of imperfectly nested loops. The system may not always produce the same time mappings since the selection of constant levels and alignments contains heuristics; however, the same loop permutations should always be selected.

Table 4.7 shows the execution times and Figures 4.17 through 4.22 show the speedups obtained for each permutation of Cholesky decomposition using both my system and the native parallelizing C compiler. As the figures show, the native compiler produces good speedups for some permutations but terrible slowdowns for others, while my system produces good speedups for all permutations. Table 4.8 shows similar results for all size legal permutations of Gaussian elimination.

```
    for ki = 1 to 1024
        if (2 <= ki) then
            for kj = 1 to ki
                for k = 1 to kj-1
3                   a[ki][kj] += -a[ki][k]*a[kj][k]
                if (kj <= ki-1) then
2                   a[ki][kj] *= 1/a[kj][kj]
1       a[ki][ki] = sqrt(a[ki][ki])
```

$$\begin{array}{llll} S_1 : \{[ki] & \rightarrow [ki]\} & T_1 : \{[ki] & \rightarrow [0, ki, 0]\} \\ S_2 : \{[ki, kj] & \rightarrow [1]\} & T_2 : \{[ki, kj] & \rightarrow [0, kj, 1, ki, 0]\} \\ S_3 : \{[ki, kj, k] & \rightarrow [kj]\} & T_3 : \{[ki, kj, k] & \rightarrow [0, k, 2, kj, 0, ki, 0]\} \end{array}$$

Figure 4.10: IJK permutation of Cholesky decomposition

```
    for ki = 1 to 1024
        for k = 1 to ki-1
2           a[ki][k] *= 1/a[k][k]
            for j = k+1 to ki
3               a[ki][j] += -a[ki][k]*a[j][k]
1       a[ki][ki] = sqrt(a[ki][ki])
```

$$\begin{array}{llll} S_1 : \{[ki] & \rightarrow [ki]\} & T_1 : \{[ki] & \rightarrow [0, ki, 0]\} \\ S_2 : \{[ki, k] & \rightarrow [1]\} & T_2 : \{[ki, k] & \rightarrow [0, k, 1, ki, 0]\} \\ S_3 : \{[ki, k, j] & \rightarrow [j]\} & T_3 : \{[ki, k, j] & \rightarrow [0, k, 2, j, 0, ki, 0]\} \end{array}$$

Figure 4.11: IKJ permutation of Cholesky decomposition

```
    for kj = 1 to 1024
        for i = kj to 1024
            for k = 1 to kj-1
3               a[i][kj] += -a[i][k]*a[kj][k]
1       a[kj][kj] = sqrt(a[kj][kj])
        for i = kj+1 to 1024
2           a[i][kj] *= 1/a[kj][kj]
```

$$\begin{array}{llll} S_1 : \{[kj] & \rightarrow [kj]\} & T_1 : \{[kj] & \rightarrow [0, kj, 0]\} \\ S_2 : \{[kj, i] & \rightarrow [1]\} & T_2 : \{[kj, i] & \rightarrow [0, kj, 1, i, 0]\} \\ S_3 : \{[kj, i, k] & \rightarrow [kj]\} & T_3 : \{[kj, i, k] & \rightarrow [0, k, 2, kj, 0, i, 0]\} \end{array}$$

Figure 4.12: JIK permutation of Cholesky decomposition

```
    for kj = 1 to 1024
        for k = 1 to kj-1
            for i = kj to 1024
3               a[i][kj] += -a[i][k]*a[kj][k]
1       a[kj][kj] = sqrt(a[kj][kj])
        for i = kj+1 to 1024
2           a[i][kj] *= 1/a[kj][kj]
```

$$
\begin{array}{llll}
S_1 : \{[kj] & \rightarrow [kj]\} & T_1 : \{[kj] & \rightarrow [0, kj, 0]\} \\
S_2 : \{[kj, i] & \rightarrow [1]\} & T_2 : \{[kj, i] & \rightarrow [0, kj, 1, i, 0]\} \\
S_3 : \{[kj, k, i] & \rightarrow [kj]\} & T_3 : \{[kj, k, i] & \rightarrow [0, k, 2, kj, 0, i, 0]\}
\end{array}
$$

Figure 4.13: JKI permutation of Cholesky decomposition

```
    for k = 1 to 1024
1       a[k][k] = sqrt(a[k][k])
        for i = k+1 to 1024
2           a[i][k] *= 1/a[k][k]
        for i = k+1 to 1024
            for j = k+1 to i
3               a[i][j] += -a[i][k]*a[j][k]
```

$$
\begin{array}{llll}
S_1 : \{[k] & \rightarrow [k]\} & T_1 : \{[k] & \rightarrow [0, k, 0]\} \\
S_2 : \{[k, i] & \rightarrow [1]\} & T_2 : \{[k, i] & \rightarrow [0, k, 1, i, 0]\} \\
S_3 : \{[k, i, j] & \rightarrow [j]\} & T_3 : \{[k, i, j] & \rightarrow [0, k, 2, j, 0, i, 0]\}
\end{array}
$$

Figure 4.14: KIJ permutation of Cholesky decomposition

```
    for k = 1 to 1024
1       a[k][k] = sqrt(a[k][k])
        for i = k+1 to 1024
2           a[i][k] *= 1/a[k][k]
        for j = k+1 to 1024
            for i = j to 1024
3               a[i][j] += -a[i][k]*a[j][k]
```

$$
\begin{array}{llll}
S_1 : \{[k] & \rightarrow [k]\} & T_1 : \{[k] & \rightarrow [0, k, 0]\} \\
S_2 : \{[k, i] & \rightarrow [1]\} & T_2 : \{[k, i] & \rightarrow [0, k, 1, i, 0]\} \\
S_3 : \{[k, j, i] & \rightarrow [j]\} & T_3 : \{[k, j, i] & \rightarrow [0, k, 2, j, 0, i, 0]\}
\end{array}
$$

Figure 4.15: KJI permutation of Cholesky decomposition

```
for (t2 = max(1,_my_id); t2 <= 1024; t2 += _my_nprocs)
    for (t4 = 1; t4 <= 1024; t4++)
        a[t2][t4] = 0;
for (t2 = max(1,_my_id); t2 <= 1024; t2 += _my_nprocs)
    a[t2][t2] = 1;
for (t2 = 1; t2 <= 1024; t2++)
    {
    if (t2 % _my_nprocs == _my_id)
        a[t2][t2] = sqrt(a[t2][t2]);
    global_barrier(0);
    if (t2 <= 1023 && 1% _my_nprocs == _my_id)
        for (t4 = t2+1; t4 <= 1024; t4++)
            a[t2][t4] *= 1/a[t2][t2];
    global_barrier(0);
    if (t2 <= 1023)
        for (t4 = adjust(t2+1); t4 <= 1024; t4 += _my_nprocs)
            for (t6 = t4; t6 <= 1024; t6++)
                a[t4][t6] += -a[t2][t6]*a[t2][t4];
    }
```

Figure 4.16: Transformed code for all permutations of Cholesky decomposition

Compiler	Permutation	Number of processors					
		1	2	4	8	12	16
Native parallelizing compiler	IJK	18.2	22.3	21.4	27.5	26.4	66.7
	IKJ	22.1	26.8	27.1	56.1	43.6	96.5
	JIK	25.8	18.8	13.5	17.2	14.4	46.8
	JKI	14.9	10.5	8.6	11.1	10.5	32.9
	KIJ	26.8	20.9	14.5	13.0	9.6	16.1
	KJI	15.3	11.3	8.3	7.1	5.3	15.4
My system	Any	27.7	20.2	11.1	6.1	4.5	3.6

(All times are in seconds)

Table 4.7: Execution times for all six permutations of Cholesky decomposition

Compiler	Permutation	Number of processors					
		1	2	4	8	12	16
Native parallelizing compiler	IJK	92.8	88.4	88.7	88.1	87.4	87.9
	IKJ	38.5	36.5	33.4	30.7	50.9	100.8
	JIK	95.9	93.5	97.7	92.2	92.1	106.5
	JKI	50.7	36.6	37.9	64.2	54.6	103.7
	KIJ	39.5	17.7	12.0	8.8	6.5	7.2
	KJI	37.5	17.6	10.8	8.9	7.2	7.9
My system	Any	33.6	30.1	20.3	18.2	19.8	15.9

(All times are in seconds)

Table 4.8: Execution times for all six permutations of Guassian elimination

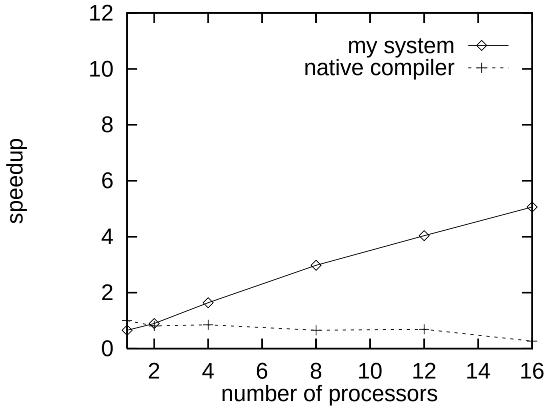


Figure 4.17: Speedup graph for IJK permutation of Cholesky decomposition

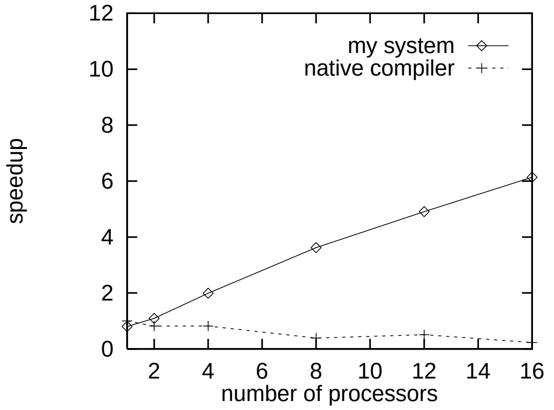


Figure 4.18: Speedup graph for IKJ permutation of Cholesky decomposition

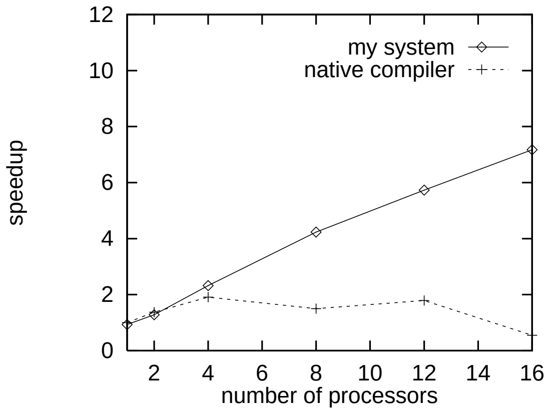


Figure 4.19: Speedup graph for JIK permutation of Cholesky decomposition

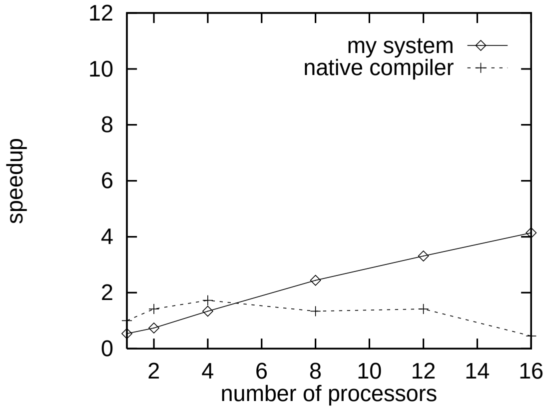


Figure 4.20: Speedup graph for JKI permutation of Cholesky decomposition

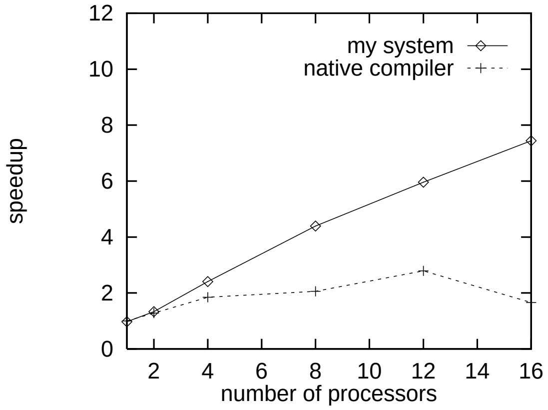


Figure 4.21: Speedup graph for KIJ permutation of Cholesky decomposition

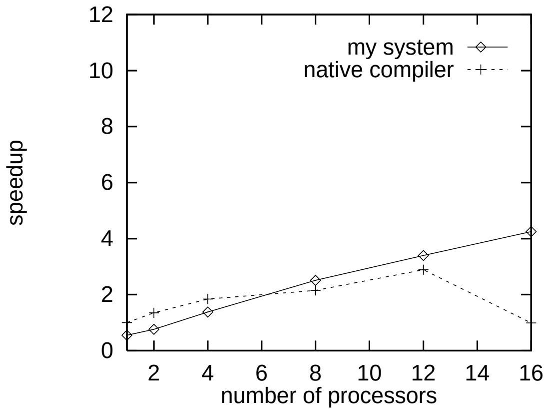


Figure 4.22: Speedup graph for KJI permutation of Cholesky decomposition

Chapter 5

Related Work

5.1 Distributing Computation

My work is most distinguished from all other related work by the fact that I am not influenced by the order of the computation in the original program. Most related works estimate parallelism or partition the program into phases based on the original loop structure.

Anderson and Lam [AL93b] have developed methods to automatically find computation and data decompositions that optimize both parallelism and locality. They first perform loop distribution and unimodular transformations to produce a canonical form consisting of fully permutable loop nests. Each loop nest is optimized in isolation to determine the computation and data decompositions that maximize parallel and data locality for that particular loop nest. Extra degrees of parallelism are traded off to eliminate non-nearest neighbor communication. If eliminating all non-nearest neighbor communication reduces the degree of parallelism to zero, then blocked distributions are used instead to produce pipeline style parallelism. A greedy approach is then used to possibly reduce communication between loop nests. The most expensive communications are considered first. If it is estimated that the cost of redistributing data between one loop nest and another is higher than the parallelism and locality benefits that would result from using different distributions for the two loop nests then the two loop nests are merged with respect to selecting distributions.

Feautrier's approach [Fea94] is to find a schedule for executing the program with maximum parallelism, ignoring locality and latency. Then, he uses a greedy algorithm, based on the dimensionality of value-based flow dependences, to select a computation distribution. It minimizes the volume of communications but doesn't place on the same virtual processor any two computations that could be run in parallel. The problem with this approach is that it is not possible to sacrifice some parallelism in a particular statement (by instead pipelining it or not distributing it), in order to reduce overall communication costs.

Although some other systems [BKK93, GAL95] also use exact rather than greedy heuristic algorithms, the size of the problems and the methods used are very different. I consider a list of candidate distributions for each statement, whereas these systems consider a list of candidate distributions for each array in each phase. My search spaces will therefore tend to be much larger. These systems use 0-1 integer programming formulations, whereas I have developed my own graph search algorithm. The performance numbers given in [GAL95] (which uses a commercial 0-1 integer programming system called LINGO) tend to suggest that the search algorithms are comparable in speed.

In Kremer's system [BKK93], an admittedly arbitrary scheme is used to identify a sequential loop nest that contains a series of phases, which are executed atomically (i.e., without overlap). Parallelism is exploited within each phase but not between them. Using techniques not described in their 1993 paper [BKK93], a set of candidate distributions are generated for each phase, and the system determines the cost of executing each phase in each distribution and the cost of re-mapping variables between each transition. The system is very dependent on obtaining a good partitioning of the program into phases and on having a good method to generate and evaluate distributions for each phase.

The system described by Garcia et.al. [GAL95] uses a static data decomposition for the entire program. They minimize communication volume and try to insure that the program can be executed in parallel simply by making some of the loops in the original program `doall` loops. They do not consider transformations such as loop distribution or interchange and do not consider pipelined parallelism nor the differences in synchronization costs between different candidate loops.

5.2 Ordering Computation

The framework of Unimodular transformations [Ban90, WL91, ST92, KKB92] has the same goal as my work, in that it attempts to provide a unified framework for describing loop transformations. It is limited by the facts that it can only be applied to perfectly nested loops, and that all statements in the loop nest are transformed in the same way. It therefore cannot represent some important transformations such as loop fusion, loop distribution and statement reordering.

Unimodular transformations are generalized by Li and Pingali in [LP92]. to include mappings that are invertible but not unimodular. This allows the resulting programs to have steps in their loops, which can be useful for optimizing locality. Li and Pingali use a technique called access normalization to improve data locality and reduce communication on non-uniform memory access computers.

Unimodular transformations are combined with blocking in [WL91, ST92]. A similar approach, although not using a unimodular framework, is described in [Wol89b].

McKinley [McK92] and Wolf [Wol92] have both developed models to estimate data locality and parallelism. Both papers highlight the importance of the tradeoff between parallelism and locality. Wolf's system is based on unimodular transformations while McKinley's system uses traditional transformations.

McKinley determines the preferred loop order for each loop nest based on data locality. If this loop order is not legal then a close to optimal loop ordering is used instead. Parallelism is then introduced by finding the outermost loop that has sufficient parallelism or an inner loop that becomes parallel when moved to an outer position. Parallelism granularity is maximized and data locality is maintained by strip mining the parallel loop – moving the outer loop as far out as possible and leaving the inner loop in the position preferred for data locality. A `doall` style of parallelism is used, so inter-processor communication cannot be controlled and pipeline parallelism cannot be used.

Wolf determines a unimodular transformation that produces the largest outermost fully permutable loop nest. A fully permutable loop nest is a set of adjacent loops such that all permutations of those loops are legal. The loops in this outermost fully permutable loop nest are then tiled so that the tiles can executed in parallel in pipelined fashion, unless of course one of these loops can be made a `doall` loop, in which case pipelining is not required. The loops within each tile are then optimized for data locality. Unimodular and tiling transformations are determined to maximize the overlap between the localized vector space (the directions in the iteration space along which reuse can be exploited) and the reuse vector space (the directions in the iteration space along which reuse occurs). The importance of exploiting locality between different iterations of the sequential loops that surround the parallel loops is discussed but not solutions are proposed.

The data locality models used by McKinley [McK92], Wolf [Wol92] and Li [Li95] differ from mine in two major respects. I assume that a data item will only remain in the cache during some period of time if it is reused every iteration during that time period. They don't impose this condition, but instead assume that a data item will remain in the cache if the period of time is relative short.

The second way in which McKinley, Wolf and Li's locality models differ from mine is that they are able to consider group reuse (i.e., when different references access the same memory locations). They are able to do this because they use simpler transformation systems in which all statements in a loop nest are transformed in the same way. This allows them to form equivalence classes of memory references that exploit group reuse with respect to a given transformation of the inner most loops. It is not feasible for me to do this since in my case; this inner most set of loops is the entire loop nest, and I apply potentially different transformations to each statement. I would therefore have to analyze the locality that would result from all combinations of the transformations that could be applied to each pair of statements that contain memory references that might exhibit group reuse. I felt that the high cost of this analysis would not be justified by the small potential benefits of exploiting group reuse.

The locality model used by Li and Pingali [LP92] is different in that it is used to derive both a loop transformation and a computation distribution given a data distribution. The first goal is to minimize off-processor data accesses by as much as possible performing computations on the same processor that owns the data being referenced. The second goal is to exploit block transfers by restructuring the loops so that communication can be performed at the outermost loop possible. Their work does not address cache locality within each processor and so is orthogonal to the locality models of McKinley and Wolf. With respect to my work, it is more related to my selection of space mappings to minimize communication, than it is to my locality estimator.

The only work that directly addresses the problem of reordering iterations given a computation distribution is the work on cross-processor loops in Fortran D [HKT91]. Fortran D normally uses an owner-computes rule and user-supplied data decompositions. An algorithm is given in [HKT91] to identify "cross-processor loops". Cross-processor loops are informally defined as:

> Sequential space-bound loops causing computation wavefronts that cross processor boundaries (*i.e.*, sweeps over the distributed dimensions).

For a block decomposition, cross-processor loops are interchanged inward; for a cyclic decomposition cross-processor loops are interchanged outward. While the definition and strategy work for stencil computations, it is not theoretically sound and the conditions it checks are neither necessary nor sufficient for a loop to be able to carry interprocessor dependences. The proposed strategy for moving loops is just a heuristic that works well on stencil computations; it isn't clear that it is valid for loops in other applications, such as linear algebra kernels, and it makes no predictions for block-cyclic decompositions.

Figure 5.1 shows some of these problems. Loops identified by [HKT91] as cross-processor are marked as `do*`. In the *False Positives* column, loops are marked as cross-processor even though there are no inter-processor dependences. In the False Negatives column, no loops are marked as do-across even though there are interprocessor dependences. These examples are designed to be demonstrative rather than realistic. There may not be any realistic stencil computations that demonstrate the problems with the above definition. But for non-stencil computations, there are some real codes on which the problems are manifested. For Cholesky decomposition (Figure 3.15) with the first dimension distributed, it identifies all 3 loops as being "cross-processor", providing no guidance.

The basis for systolic techniques was laid by Karp, Miller and Winograd's paper [KMW67] on uniform recurrences. The basic idea is that given a set of function values to be computed, and a set of recurrence equations constraining the function values, the computation can be organized by specifying a schedule that defines the "time" at which each function value should be computed. Lamport's paper [Lam74] on Hyper-planes, was the first to apply these ideas to parallelizing compilers. These ideas have continued to evolve following the development of systolic arrays. Lengauer [Len93] provides a good summary of traditional systolic techniques.

False Positives	False Negatives

`Decomposition T(N)`
`real A(N)`
`Align A(j) with T(j)`
`Distribute T(block)`
`do* i = 1 to n`
`  do* j = i+1 to n`
`      A(i) = A(i) + 1`
`      A(j) = A(j) - 1`
	`Decomposition T(N)`
	`real A(N,N), B(N,N)`
	`Align A(i,:) with T(i)`
	`Align B(i,:) with T(i)`
	`Distribute T(block)`
	`do i = 1 to n`
	`  do j = 1 to n`
	`    A(i,j) = B(i+1,j-1)`
	`    B(i+1,j) = A(i,j)`

`Decomposition T(N)`
`real A(N), B(N)`
`Align A(i) with T(i)`
`Align B(i) with T(i+1)`
`Distribute T(block)`
`do* i = 1 to n`
`      A(i) = ...`
`      B(i) = A(i+1)`
	`Decomposition T(N)`
	`real A(N,N), B(N,N)`
	`Align A(i,:) with T(i)`
	`Align B(i,:) with T(i+1)`
	`Distribute T(block)`
	`do i = 1 to n`
	`  do j = 1 to n`
	`    A(i,j) = B(i,j-1)`
	`    B(i,j) = A(i,j)`

Figure 5.1: Errors made by the Fortran-D "cross-processor" identification

Pugh [Pug91] gives techniques to represent loop fusion, loop distribution and statement reordering in addition to the transformations representable by unimodular transformations. Because it uses only single level affine schedules and requires that all dependences be carried by the outer loop, it can only be applied to programs that can be executed in linear time on a parallel machine.

Paul Feautrier [Fea92a, Fea92b] independently developed a framework which is very similar to my own. It is similar in the following respects:

- He represents reordering transformations using multi-dimensional schedules which are similar in form to my time mappings.
- He generates a separate schedule for each statement.

However, my work differs from Feautrier's in the following respects:

- Unlike my time mappings, Feautrier's schedules are not required to be 1-1. Instead, iterations that are to be executed in parallel are scheduled at the same point in time. Therefore, Feautrier's schedules only partially specify the transformed code. In a separate decision process, parallel loops are generated to enumerate all the computations that need to be executed at each time point. This framework only allows the generation of innermost parallel loops; outer parallel loops are often desirable.
- His methods are designed to generate a schedule that produces code with a "maximal" amount of parallelism. He does this by generating a large set of constraints which describe all legal schedules. This set of constraints has a variable for each coefficient and each constant term of the schedule for each statement. For example, for the code from `OLDA` in Figure 2.11, the problem generated by Feautrier would have 6 variables for each statement: 3 each for the coefficients of p, q and i, 2 each for the coefficients of n and orb and 1 each for the constant term. He then introduces two linear functions of these variables, one representing the number of iterations that will be executed sequentially and a second

representing how many dependences will be carried. These functions and constraints are then combined and transformed into a dual programming problem that is solved using Parametric Integer Programming (PIP). The net result of this process is that the schedule selected carries as many dependences as possible and among all such schedules, the one selected has as few sequential iterations as possible. These schedules will often not be optimal in practice because he ignores issues such as granularity, data locality and code complexity. It is unclear if his method could be extended to include other criteria, such as good cache performance or parallel outer loops. I expect it would be difficult to encode such an optimization function for a code segment containing several statements.

My approach differs fundamentally from Feautrier's in that at each stage I am "trying" specific loop permutations. Working with actual loop permutations, rather than with formulas describing schedules, makes it much easier to analyze complex performance issues such as data locality.

5.3 Generating Code

The problem of generating code for a convex region was first addressed by Ancourt and Irigoin [AI91]. They use Fourier pairwise elimination at each level to provide bounds on each of the index variables. They then form the union of all of these projections to produce a single set of constraints which explicitly contains all of the information necessary to generate code. They propose that fast inexact techniques be used to remove redundancies from this set before it is used to generate code. They consider only the single mapping convex case.

Lassez et.al. [HLL92] provide a good perspective on different methods used to project integer convex polyhedra.

Li and Pingali [LP92] consider the non-convex case resulting from mappings that are not necessarily onto. They use a linear algebra framework and compute loop bounds and steps using Hermite normal form. They do not consider the multiple mapping case.

Ayguadé and Torres [AT93] consider a limited case of the multiple mapping case where each statement can have a potentially different mapping but all mappings must have the same linear part (i.e., they only differ in their constant parts).

Chamski [Cha93a, Cha93b] generates Nested Loop Structures. He discusses generating code only for the single mapping convex case. He reduces control overhead by generating sequences of loops to remove all min and max expressions in loop bounds. The cost of code duplication may be large when all such overheads are removed.

Chamski claims [Cha93a] that Fourier variable elimination is prohibitively expensive for code generation. I have found it to be a very efficient method, and suspect he used unrealistic examples and/or a poor implementation of Fourier variable elimination. It is well known that Fourier variable eliminate performs poorly on moderate to large systems of constraints where the constraints are dense (each constraint involves many variables). However, the constraints I have seen in both dependence analysis and code generation are quite sparse, and Fourier elimination is quite efficient for sparse constraints [PW94a].

Collard, Feautrier and Risset [CFR93] show how PIP, a parametrized version of the Dual Simplex Method, can be used to solve the single mapping case. Collard and Feautrier [CF93] address the multiple mapping case; however, only one dimensional iteration spaces are considered and many guards are generated. They provide some interesting solutions to the situation where statements have incompatible stride constraints (e.g., t_1 is even and t_1 is odd). Such stride constraints arise frequently in Feautrier's parallelization framework [Fea92a, Fea92b]; I try to avoid them in my framework, since generating good code for them is difficult.

In earlier work Pugh, Rosser and I [KPR95] developed an effective code generation algorithm for the general multiple time mapping problem. That earlier work included techniques to generate non-unit steps to iterate over non-convex regions and a framework within which to trade-off control overhead for code duplication. Our algorithm can be summarized as follows:

We first construct an *abstract syntax tree* (AST) that defines an initial structure of the loops and conditions. In determining the initial structure, we try to introduce as little control overhead as possible under the restriction that no code duplication is introduced. We augment this tree with more detailed information regarding the conditions and loop bounds of the conditionals and loops respectively. Next we consider removing control overhead. Sources of overhead nested inside the most deeply nested loops will be executed most frequently and are the most important to remove. Unfortunately, further removing overhead requires code duplication and an increase in code size. This trade-off is controlled by specifying the depths from which overhead will be eliminated. Once we have performed these optimizations, we generate the actual code using the abstract syntax tree and the information that it contains.

Chapter 6

Future Work

6.1 Distributed Memory

So far, I have targeted my system at machines with logically shared but physically distributed memory systems, such as the Stanford DASH and SGI POWER CHALLENGE. In the future, I would like to also target machines with logically distributed memory systems. Both architectures require communication volume to be minimized, however, in a distributed memory setting, it may be necessary to augment or replace computation decompositions with data decompositions. It is much harder to select a data decomposition than a computation decomposition if the computation order is not known. Having to explicitly manage data storage and communication also gives rise to a few additional problems such as how to aggregate messages.

6.2 Arbitrary Control Flow

I would also like to extend my system to accept a wider range of input programs. Specifically, I would like to accept programs with arbitrary control flow constructs, such as conditionals with non-affine conditions, loops with non-affine bounds, while loops, and gotos.

6.3 Multiple Procedures

I would also like to accept as input entire programs rather than just single procedures. In such a system, I would have to decide whether each procedure would be optimized in isolation or the whether the entire program would to optimized as a whole. If each procedure is to be optimized in isolation then the order in which procedures are optimized becomes important; for example, there is no point in trying to parallelize a procedure if it is always called from within a loop that is totally parallel. If only some of the calls to a procedure are from within a parallel loop then it may be desirable to clone the procedure and only parallelize one of the clones. If any cloning is considered then care must be taken that it doesn't occur so often that the code size becomes unreasonably large. Assumptions made about the interfaces between procedures such as how the procedure parameters are distributed in memory also becomes an interesting question.

6.4 Improved Space Mappings

My space mapping selection algorithm remains heuristic in one major way: I combine the effects of parallelism and communication simply by multiplying one by a constant parameter and then adding them together. This method of combination will be inaccurate if communication can be substantially overlapped with computation or with other communication. This heuristic was forced on me by a "chicken and egg" problem: it is difficult to distribute the computations until the final order of the computations is known, but it is also difficult to order the computations until the distribution is known. The heuristic works well in practice because the largest communications are unlikely to be substantially overlapped with computation. I would, however, like to improve this aspect of the system.

The algorithm that I use to detect false sharing in Section 3.2.4 is overly pessimistic in some cases and overly optimistic in others. I would like to correct this situation by better understanding when false sharing occurs and how much of an impact it has.

I would also like to extend my algorithms to produce multi-dimensional space mappings. Multi-dimensional virtual processor spaces often map better to the physical topology of actual machines and can reduce communication costs by reducing the ratio of "surface area" to "volume" on each processor, and by making use of faster nearest neighbor interconnection networks.

6.5 Improved Time Mappings

My time mapping selection algorithm is also heuristic. When selecting constant levels I always separate statements from one another as soon as possible. This corresponds to applying maximal loop distribution and is not always desirable. The issues governing whether or not to distribute are very complex and need to be better understood before a attempting anything other than maximal distribution.

The selection of alignment constants is also heuristic. The current approach works very well in practice but this is probably only because maximal separation of statements is also being performed. If I decide to change that policy then I will also have to reconsider the alignment problem.

The time mapping selection algorithm is only capable of producing a small subset of the transformations that can be represented using time mappings. I would like to extend the algorithm to produce time mappings that correspond to additional transformations such as loop skewing, loop tiling and index set splitting.

Other, non iteration reordering transformations, such as the data transformations proposed by Anderson et.al. [AAL95], should also be incorporated into the overall framework.

6.6 My Ultimate Goal

My ultimate goal is to strengthen the system, both in terms of efficiency and effectiveness, to a point where it could be directly incorporated into a production-quality parallelizing compiler.

Chapter 7

Conclusions

The main contribution of my work has been to improve the state of the art for parallelizing compilers by replacing existing ad hoc parallelization techniques with a sound underlying foundation on which future work can be built. I have developed a new framework that unifies the problems of distributing and ordering computation. The new framework is based on a simple but very powerful mathematical abstraction for representing these decisions. I have also developed algorithms for making these decisions within this framework.

The framework is extremely extensible, in that the set of transformations considered and the performance estimators used to decide which transformation to apply, are not hard wired into the system. Users can to modify or write performance estimators to reflect the factors which affect performance on their particular architecture. They can also modify the set of transformations considered, to obtain the trade-off between efficiency and effectiveness that best suits their individual needs.

Conversely, many of the abstractions and algorithms I have developed have applications outside of the framework. For example, my data locality model could be used by any transformation system, our code generation algorithm could be used to generate code even for transformation systems that aren't based on time and space mappings and my barrier placement algorithm could be used to optimally place communication statements in a distributed memory environment.

The generality and flexibility of my system stems from the fact that all of my abstractions (such as space mappings, time mappings and data dependence relations) and operations (such as testing legality and generating code) are represented in a common mathematical framework (Presburger arithmetic). By making use of the Omega library, I have been able to produce a system that is extremely general, while still practical for everyday problems. Using a high level language such as Presburger arithmetic also has the effect that many problems can be expressed and solved in a simple and elegant way.

Most of the problems dealt with in this thesis are either undecidable or NP-hard. Two basic approaches have been taken in this thesis:

- Use polynomial time heuristic based algorithms that sometimes produce sub-optimal results, but demonstrate that acceptable results are obtained in most realistic cases.
- Use algorithms that theoretically have exponential worst case execution time, but demonstrate that in realistic cases, either the exponential behavior does not occur, or the problem size is small enough that exponential behavior is not a concern.

In both cases, it is necessary to implement the algorithms and perform many experiments to prove that they are both efficient and effective. In fact, it is even necessary to perform experiments to determine which of these two approaches to use for a given problem. Implementation and experimentation have therefore been the primary tools that I have used to develop and refine new algorithms.

The most important aid to developing good algorithms is the use of the appropriate abstraction for representing the major data objects being manipulated. For example, data dependences can be represented exactly using tuple relations or they can be summarized using dependence directions. Tuple relations provide the most information but are too expensive for some applications. The appropriate abstraction must be chosen by considering how much information is actually required and, if necessary, sacrificing some effectiveness for efficiency. The primary lesson that I have learned from this experience is that it is easier to develop new algorithms by starting with the most exact abstraction possible and only resorting to a weaker abstraction if the more exact abstraction is unnecessary or too expensive. Starting with the most exact abstraction makes it easier to identify and understand the problem inherently being solved and makes the algorithm designer aware of any compromises bring made and the scope of their effect.

Bibliography

[AAL95] J. Anderson, S. Amarasinghe, and M. Lam. Data and computation transformations for multiprocessors. In *Proc. of the 5th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, July 1995.

[ACK87] R. Allen, D. Callahan, and K. Kennedy. Automatic decomposition of scientific programs for parallel execution. In *Conference Record of the Fourteenth ACM Symposium on Principles of Programming Languages*, pages 63–76, January 1987.

[AG94] George S. Almasi and Allan Gottlieb. *Highly Parallel Computing.* Benjanin/Cummings, 1994.

[AI91] Corinne Ancourt and Franccois Irigoin. Scanning polyhedra with DO loops. In *Proc. of the 3rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 39–50, April 1991.

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

[AL93a] Saman P. Amarasinghe and Monica S. Lam. Communication optimization and code generation for distributed memory machines. In *ACM '93 Conf. on Programming Language Design and Implementation*, June 1993.

[AL93b] Jennifer M. Anderson and Monica S. Lam. Global optimizations for parallelism and locality on scalable parallel machines. In *ACM '93 Conf. on Programming Language Design and Implementation*, June 1993.

[AT93] Eduard Ayguadé and Jordi Torres. Partitioning the statement per iteration space using non-singular matrices. In *International Conference on Supercomputing*, pages 407–415, July 1993.

[Ban90] U. Banerjee. Unimodular transformations of double loops. In *Proc. of the 3rd Workshop on Programming Languages and Compilers for Parallel Computing*, pages 192–219, Irvine, CA, August 1990.

[BKK93] R. Bixby, K. Kennedy, and U. Kremer. Automatic data layout using 0-1 integer programming. Technical Report CRPC-TR93349-S, Center for Research on Parallel Computation, Rice University, November 1993.

[Blu92] William Joseph Blume. Success and limitations in automatic parallelization of the Perfect benchmarks^{TM} programs. Master's thesis, Dept. of Computer Science, U. of Illinois at Urbana-Champaign, 1992.

[CF93] J.-F. Collard and P. Feautrier. Automatic generation of data parallel code. In H.J. Sips, editor, *Proceedings of the Fourth International Workshop on Compilers for Parallel Computers*, pages 321–332, Delft, The Netherlands, December 1993.

[CFR93] Jean-Franccois Collard, Paul Feautrier, and Tanguy Risset. Construction of DO loops from systems of affine constraints. Technical Report N° 93-15, Laboratoire de l'Informatique du Parallélisme, Ecolo Normal Supérieure de Lyon, Instiut IMAG, May 1993.

[CGC96] Soumen Chakraharti, Manish Gupta, and Jong-Deok Choi. Global communication analysis and optimization. In *ACM SIGPLAN '96 Conf. on Programming Language Design and Implementation*, May 1996.

[Cha93a] Zbigniew Chamski. Fast and efficient generation of loop bounds. Publication interne 771, Institut de Recherche en Informatique et Systèmes Aléatoires, October 1993.

[Cha93b] Zbigniew Chamski. Nested loop sequences: Towards efficient loop structures in automatic parallelization. Publication interne 772, Institut de Recherche en Informatique et Systèmes Aléatoires, October 1993.

[CK88] D. Callahan and K. Kennedy. Compiling for distributed-memory multiprocessors. *Journal of Supercomputing*, 2:151–169, October 1988.

[CK92] Steve Carr and Ken Kennedy. Compiler blockability of numerical algorithms. In *Proceedings Supercomputing'92*, pages 114–125, Minneapolis, Minnesota, Nov 1992.

[Fea92a] Paul Feautrier. Some efficient solutions to the affine scheduling problem, Part I, One-dimensional time. *Int. J. of Parallel Programming*, 21(5), Oct 1992.

[Fea92b] Paul Feautrier. Some efficient solutions to the affine scheduling problem, Part II, Multidimensional time. *Int. J. of Parallel Programming*, 21(6), Dec 1992.

[Fea94] Paul Feautrier. Toward automatic distribution. *Parallel Processing Letters*, 4(3):233–244, September 1994.

[For92] High Performance Fortran Forum. High performance fortran language specification, version 1.0. Technical Report CRPC-TR92225, Center for Research on Parallel Computation, Rice University, 1992. Revised May, 1993.

[GAL95] Jordi Garcia, Eduard Ayguade, and Jesus Labarta. A novel approach towards automatic data distribution. In *Workshop on Automatic Data Layout and Performance Predition*, April 1995.

[Gav72] Fanica Gavril. Algorithms for minimum coloring, maximum clique, minimum covering by cliques, and maximum independent set of a chordal graph. *SIAM J. Computing*, 1(2):180–187, June 1972.

[Gup92] M. Gupta. *Automatic Data Partioning on Distributed Memory Multicomputers.* PhD thesis, Dept. of Computer Science, U. of Illinois at Urbana-Champaign, 1992.

[HKT91] Seema Hiranandani, Ken Kennedy, and Chau-Wen Tseng. Compiler optimizations for FORTRAN D on MIMD distributed memory machines. In *Supercomputing '91*, November 1991.

[HLL92] Tien Huynh, Catherine Lassez, and Jean-Louis Lassez. Practical issues on the projection of polyhedral sets. *Annals of mathematics and artificial intelligence*, November 1992.

[Ken92] Ken Kennedy. Guest editorial: Software for supercomputers of the future. *The Journal of Supercomputing*, pages 251–262, 1992.

[KKB92] K. G. Kumar, D. Kulkarni, and A. Basu. Deriving good transformations for mapping nested loops on hieracical parallel machines in polynomial time. In *Proc. of the 1992 International Conference on Supercomputing*, pages 82–92, July 1992.

[KMP+95] Wayne Kelly, Vadim Maslov, William Pugh, Evan Rosser, Tatiana Shpeisman, and David Wonnacott. The Omega Library interface guide. Technical Report CS-TR-3445, Dept. of Computer Science, University of Maryland, College Park, March 1995. The Omega library is available from http://www.cs.umd.edu/projects/omega.

[KMW67] R.M. Karp, R.E. Miller, and S. Winograd. The organization of computations for uniform recurrence equations. *Journal of the ACM*, 14:563–590, 1967.

[KP93] Wayne Kelly and William Pugh. A framework for unifying reordering transformations. Technical Report CS-TR-2995.1, Dept. of Computer Science, University of Maryland, College Park, May 1993.

[KPR95] Wayne Kelly, William Pugh, and Evan Rosser. Code generation for multiple mappings. In *The 5th Symposium on the Frontiers of Massively Parallel Computation*, pages 332–341, McLean, Virginia, February 1995.

[KPRS95] Wayne Kelly, William Pugh, Evan Rosser, and Tatiana Shpeisman. Transitive closure of infinite graphs and its applications. In *Eighth Annual Workshop on Programming Languages and Compilers for Parallel Computing*, Columbus, OH, August 1995.

[Lam74] Leslie Lamport. The parallel execution of DO loops. *Communications of the ACM*, 17(2):83–93, February 1974.

[Len93] Christian Lengauer. Loop parallelization in the polytope model. In *CONCUR*, 1993.

[Li95] Wei Li. Compiler cache optimizations for banded matrix problems. In *9th ACM International Conference on Supercomputing*, Barcelona, Spain, July 1995.

[LP92] Wei Li and Keshav Pingali. A singular loop transformation framework based on non-singular matrices. In *5th Workshop on Languages and Compilers for Parallel Computing*, pages 249–260, Yale University, August 1992.

[McK92] Kathryn S. McKinley. *Automatic and Interactive Parallelization*. PhD thesis, Dept. of Computer Science, Rice U., April 1992.

[Pug91] William Pugh. Uniform techniques for loop optimization. In *1991 International Conference on Supercomputing*, pages 341–352, Cologne, Germany, June 1991.

[Pug92] William Pugh. The Omega test: a fast and practical integer programming algorithm for dependence analysis. *Communications of the ACM*, 8:102–114, August 1992.

[Pug94] William Pugh. Counting solutions to presburger formulas: How and why. In *SIGPLAN Conference on Programming Language Design and Implementation*, Orlando, FL, June 1994.

[PW92] William Pugh and David Wonnacott. Going beyond integer programming with the Omega test to eliminate false data dependences. Technical Report CS-TR-3191, Dept. of Computer Science, University of Maryland, College Park, December 1992. An earlier version of this paper appeared at the SIGPLAN PLDI'92 conference.

[PW93] William Pugh and David Wonnacott. An exact method for analysis of value-based array data dependences. In *Lecture Notes in Computer Science 768: Sixth Annual Workshop on Programming Languages and Compilers for Parallel Computing*, Portland, OR, August 1993. Springer-Verlag.

[PW94a] William Pugh and David Wonnacott. Experiences with constraint-based array dependence analysis. In *Principles and Practice of Constraint Programming Workshop*, Orcas Island, Washington, May 1994.

[PW94b] William Pugh and David Wonnacott. Static analysis of upper and lower bounds on dependences and parallelism. *ACM Transactions on Programming Languages and Systems*, 14(3):1248–1278, July 1994.

[SH91] Jaswinder Pal Singh and John L. Hennessy. An empirical investigation of the effectiveness and limitations of automatic parallelization. In *Proceedings of the International Symposium on Shared Memory Multiprocessing*, April 1991.

[SSP+95] T. J. Sheffler, R. Schreiber, W. Pugh, J. R. Gilbert, and S. Chatterjee. Efficient distribution analysis via graph contraction. In *Eighth Annual Workshop on Programming Languages and Compilers for Parallel Computing*, Columbus, OH, August 1995.

[ST92] Vivek Sarkar and Radhika Thekkath. A general framework for iteration-reordering loop transformations. In *ACM SIGPLAN'92 Conference on Programming Language Design and Implementation*, pages 175–187, San Francisco, California, Jun 1992.

[WB87] Michael Wolfe and Utpal Banerjee. Data dependence and its application to parallel processing. *Internation J. Parallel Programming*, 16(2):137–178, April 1987.

[WL91] Michael E. Wolf and Monica S. Lam. A data locality optimizing algorithm. In *ACM SIGPLAN'91 Conference on Programming Language Design and Implementation*, 1991.

[Wol82] M. J. Wolfe. *Optimizing Supercompilers for Supercomputers*. PhD thesis, Dept. of Computer Science, University of Illinois at Urbana-Champaign, October 1982.

[Wol89a] M. J. Wolfe. *Optimizing Supercompilers for Supercomputers*. The MIT Press, Cambridge, MA, 1989.

[Wol89b] Michael Wolfe. More iteration space tiling. In *Proc. Supercomputing 89*, pages 655–664, November 1989.

[Wol89c] Michael Wolfe. *Optimizing Supercompilers for Supercomputers*. Pitman Publishing, London, 1989.

[Wol90] Michael Wolfe. Massive parallelism through program restructuring. In *Symposium on Frontiers on Massively Parallel Computation*, pages 407–415, October 1990.

[Wol91a] Michael Wolfe. Experiences with data dependence abstractions. In *Proc. of the 1991 International Conference on Supercomputing*, pages 321–329, June 1991.

[Wol91b] Michael Wolfe. The tiny loop restructuring research tool. In *Proc of 1991 International Conference on Parallel Processing*, pages II–46 – II–53, 1991.

[Wol92] Michael Edward Wolf. *Improving Locality and Parallelism in Nested Loops*. PhD thesis, Dept. of Computer Science, Stanford U., August 1992.

[Won95] David G. Wonnacott. *Constraint-Based Array Data Dependence Analysis*. PhD thesis, Dept. of Computer Science, The University of Maryland, August 1995.

[ZC91] Hans Zima and Barbara Chapman. *Supercompilers for Parallel and Vector Computers.* ACM Press, 1991.