Compiler Analysis to Implement Point-to-Point Synchronization in Parallel Programs

by

John Nguyen

S.B., Computer Science
S.B., Mathematics
Massachusetts Institute of Technology
(1987)

S.M., Electrical Engineering and Computer Science
Massachusetts Institute of Technology
(1989)

Submitted to the Department of
ELECTRICAL ENGINEERING AND COMPUTER SCIENCE
in partial fulfillment of the requirements
for the degree of
DOCTOR OF PHILOSOPHY
at the
MASSACHUSETTS INSTITUTE OF TECHNOLOGY

August 1993

Signature of Author: ______________________________
Department of Electrical Engineering and Computer Science
August 13, 1993

Certified by: ______________________________
Stephen A. Ward
Professor of Computer Science and Engineering
Thesis Supervisor

Accepted by: ______________________________
Frederic R. Morgenthaler
Chairman, Department Committee on Graduate Students

Compiler Analysis to Implement Point-to-Point Synchronization in Parallel Programs

by

John Nguyen

Submitted to the Department of Electrical Engineering and Computer Science
on August 13, 1993 in partial fulfillment of the requirements
for the Degree of Doctor of Philosophy
in Electrical Engineering and Computer Science

Abstract

The shared-memory data-parallel model presents an attractive interface for programming multiprocessors by allowing for easy management of parallel tasks while hiding details of the underlying machine architecture. Unfortunately, the shared-memory abstraction requires synchronization in order to maintain data consistency. Present compilers provide consistency between parallel code sections by enforcing a global point of synchrony with a barrier synchronization. Such a simple mechanism possesses several disadvantages. First, the required global collection of information generates significant overhead which leads machine designers to employ special hardware to support barriers. Second, global synchronization reduces parallelism by requiring needless serialization of independent tasks. This work aims to reduce the costs associated with these disadvantages by generating pairwise point-to-point synchronization between specific tasks.

Implementation of point-to-point synchronization demands extensive analysis of program dependences. A compiler must perform flow analysis and dependence testing in order to compute lexical dependences between program statements. In addition, dynamic dependences between processors must be computed by examining array references and statement contexts. The final synchronization scheme must support any dependences that arise in the program while ensuring that no deadlock scenarios can occur. This work proposes algorithms that satisfy such requirements and presents some encouraging results from a preliminary implementation.

Thesis Supervisor: Stephen A. Ward

Title: Professor of Electrical Engineering and Computer Science

Acknowledgments

In finishing a work that represents the culmination of my many years at MIT, I owe my gratitude to many people whose contributions range from technical advice to non-academic diversions to general support and encouragement.

I thank my advisor, Steve Ward, for his constructive questions that have enabled me to more clearly grasp my thoughts, for his attitude that has made the NuMesh group such a fun and interesting place to work, and for his support that has allowed me to grow as a person during my years with the group.

My readers, Anant Agarwal and Greg Papadopoulos, have helped me greatly to focus on a thesis topic and provided advice and encouragement which have made the process much smoother.

Many members of the NuMesh group and the Computer Architecture Group have contributed to make the last six years enjoyable. In particular, I would like to thank Gill Pratt for supplying some interesting collaborations, Andy Ayers for providing thesis discussion and commiseration, and my officemate Milan Singh for putting up with my thesis ramblings. I would also like to thank the group assistant, Anne McCarthy, for tackling the bureaucracy in my behalf.

The people who have been a part of the Vile Servers volleyball team through the years have provided a great source for camaraderie, friendly competition, and steady release from the pressures of work.

My roommate of many years, Bill Schmitt, and Tony Bogner have supplied numerous interesting diversions ranging from road trips to the many late-night card games. I would also like to thank the New House IV alumni who comprise the ''gang'' e-mail list for the jokes, brain teasers, and reunions.

I would like to thank my fiancée, Tricia, for being an endless source of support and encouragement, for making the best of a long-distance relationship with a graduate student, and for her friendship.

Finally, I would like to thank my parents for their guidance, generosity, and love through the years. My accomplishments would not have been possible without their support.

This work has been sponsored in part by a National Science Foundation Fellowship, in part by DARPA contract #DABT63-93-C-0008, and in part by Texas Instruments.

Table of Contents

1. Introduction 15
   1.1. Related work 16
   1.2. Problem identification 18
   1.3. Approach 20
      1.3.1. Synchronization variables 21
      1.3.2. Computing statement dependences 22
      1.3.3. Computing processor dependences 22
      1.3.4. Optimizing point-to-point synchronization 23
      1.3.5. Variable replication 24
   1.4. Thesis outline 26

2. Background 29
   2.1. Language description 29
   2.2. Control flow graph 30
   2.3. Machine model 33

3. Statement dependences 35
   3.1. Introduction 35
   3.2. Propagation of linear induction variables 35
      3.2.1. Value lattices 36
      3.2.2. Propagation of linear induction variables 39
   3.3. Flow analysis on arrays 43
      3.3.1. Linear integer sequences 44
      3.3.2. Summary of array index approximations 48
      3.3.3. Subarrays 48
      3.3.4. Array flow analysis algorithm 49
      3.3.5. Flow analysis on multi-dimensional arrays 53

3.3.6. Related work 54

3.4. Detection of dependences 54

3.4.1. Invariant expressions 58

3.5. Interprocedural support 59

3.6. Other applications of array flow analysis 60

3.6.1. Parallelism detection 61

3.6.2. Private variable detection 61

3.6.3. Data and loop partitioning 62

3.6.4. Static routing of data 63

3.7. Summary 63

4. Processor dependences and synchronization 65

4.1. Introduction 65

4.2. Motivation 65

4.2.1. Synchronization model 66

4.2.2. Implementation issues 67

4.2.3. Termination issues 70

4.3. Overview of processor dependences 71

4.4. Dynamic instances of statements 73

4.5. Deriving synchronization relationships 75

4.5.1. The problem 75

4.5.2. Orthogonal derivation of instance relationships 76

4.5.3. Instance relationships 77

4.5.4. Related work 79

4.6. Execution model of parallel loops 80

4.7. Execution order of statement instances 85

4.8. Computation of synchronization targets 87

4.8.1. Motivation 88

4.8.2. Static computation of synchronization targets 89

4.8.3. Framework for deducing processor targets 91

4.8.4. Computation of processor targets 94

4.8.5. Computation of temporal targets 96
4.8.6. An example 100

4.9. Implementation issues 101
4.9.1. An algorithm 102
4.9.2. Implementation of synchronization primitives 103

4.10. Deadlock avoidance 105

4.11. DOACROSS loops 109

4.12. Summary 110

5. Optimizations 113

5.1. Introduction 113

5.2. Synchronization by message-passing 113

5.3. Redundant dependences 116
5.3.1. Motivation 117
5.3.2. Problem definition 117
5.3.3. A solution for a simple problem domain 119
5.3.4. General removal of redundant dependences 125
5.3.5. Redundant dependences in structured programs 129

5.4. Eliminating false dependences 130

5.5. Summary 135

6. Results 137

6.1. Applications 137

6.2. Simulation environment 139

6.3. An example 141

6.4. General application study 147

6.5. Summary 152

7. Future work 153

7.1. Introduction 153

7.2. Multiple loop indices 153

7.3. Interprocedural analysis 154

7.4. Synchronization groups 155

7.5. Partitioning 157

8. Conclusion 161

8.1. Summary 161

8.2. Contributions of this thesis 163

Bibliography 165

Index of terms 175

Index of notation 177

Table of Figures

Figure 1.1 ... 17
Figure 1.2 ... 18
Figure 1.3 ... 18
Figure 1.4 ... 20
Figure 1.5 ... 21
Figure 1.6 ... 21
Figure 1.7 ... 23
Figure 1.8 ... 25
Figure 1.9 ... 25
Figure 1.10 ... 26

Figure 2.1 ... 29
Figure 2.2 ... 30
Figure 2.3 ... 31

Figure 3.1 ... 36
Figure 3.2 ... 37
Figure 3.3 ... 38
Figure 3.4 ... 39
Figure 3.5 ... 41
Figure 3.6 ... 42
Figure 3.7 ... 44
Figure 3.8 ... 47
Figure 3.9 ... 50
Figure 3.10 ... 51
Figure 3.11 ... 52
Figure 3.12 ... 53
Figure 3.13 ... 56
Figure 3.14 ... 57
Figure 3.15 ... 59
Figure 3.16 ... 60
Figure 3.17 ... 61

Figure 3.18 62

Figure 4.1 67
Figure 4.2 68
Figure 4.3 69
Figure 4.4 69
Figure 4.5 70
Figure 4.6 71
Figure 4.7 72
Figure 4.8 78
Figure 4.9 80
Figure 4.10 81
Figure 4.11 83
Figure 4.12 85
Figure 4.13 88
Figure 4.14 90
Figure 4.15 98
Figure 4.16 100
Figure 4.17 103
Figure 4.18 104
Figure 4.19 108

Figure 5.1 114
Figure 5.2 115
Figure 5.3 117
Figure 5.4 118
Figure 5.5 119
Figure 5.6 121
Figure 5.7 123
Figure 5.8 124
Figure 5.9 127
Figure 5.10 128
Figure 5.11 130
Figure 5.12 131
Figure 5.13 132
Figure 5.14 132

Figure 5.15 ... 133
Figure 5.16 ... 134

Figure 6.1 ... 140
Figure 6.2 ... 141
Figure 6.3 ... 142
Figure 6.4 ... 143
Figure 6.5 ... 144
Figure 6.6 ... 145
Figure 6.7 ... 146
Figure 6.8 ... 148
Figure 6.9 ... 149
Figure 6.10 ... 151
Figure 6.11 ... 152

Figure 7.1 ... 153
Figure 7.2 ... 155
Figure 7.3 ... 155
Figure 7.4 ... 157
Figure 7.5 ... 158
Figure 7.6 ... 158

Chapter 1

Introduction

The concept of devoting many processing elements to one task in order to increase performance has existed for several decades. Implementations of this concept vary from early array processors such as the Illiac IV [Bou72] to the more decoupled MIMD machines of today [Smi78][Sei85][Thi91]. Early array processors and SIMD machines allow parallelism through repeated application of a single computation or instruction to different data. Though this sort of concurrency is effective for certain program domains, the inability to follow different instructions and control paths in parallel reduces its generality. On the other hand, MIMD machines allow each processor to follow independent asynchronous programs with a data communication network forming the only link between processors. However, this independence comes at a price: Explicit synchronization must be performed to ensure correct ordering of accesses to shared memory.

This thesis focuses on the domain of programs that make extensive use of parallel loops and arrays to express data parallelism. The common model for invoking such programs on multiprocessors involves two modes of execution: sequential and parallel. Sequential code segments are executed on a single processor or host, while parallel code can be executed on all processors. Sections of code containing parallel instructions can be represented as `DOALL` loop statements which specify that all iterations can be executed in parallel. On array and SIMD machines, the transition between parallel and sequential sections comes at no additional cost since all processors execute in lock-step. On MIMD machines, a barrier synchronization is typically performed between parallel and sequential sections to ensure correctness of results. When a barrier synchronization appears in a program, no processors can proceed past the barrier point until all processors have reached that point.

The barrier synchronization allows MIMD machines to follow the SIMD model of program execution by requiring all processors to wait at the barrier point until all other processors have arrived at that point. On machines with many processors, this global propagation of information can require a significant amount of time to execute. Equally importantly, barrier synchronizations can force serialization of operations on different processors even when no dependences exist between them. If parallel loop iterations possess fairly dynamic control flow, this can result in unnecessary idling and imply that the time required to execute each loop is equal to the maximum time required by any processor [DH88]. With a more decoupled synchronization scheme, consecutive loops can be allowed to stagger, thus providing higher processor utilization. The above disadvantages can be addressed by employing a point-to-point synchronization scheme in which processors synchronize individually with other processors.

1.1 Related work

Barrier synchronization has become popular as a necessary tool for implementing the SPMD (Single Program Multiple Data) model on MIMD machines. Consequently, many efforts have been made to reduce the potentially high expense of this operation [Pol88][AJ87]. However, many of these schemes still rely on global propagation and do not address the problem of processor idling at barrier points.

"Fuzzy" barriers [Gup89] reduce idling by breaking barrier synchronization into two phases: signaling and waiting. In conventional execution, a processor arrives at the barrier point, signals that it has arrived at that point, then waits until all other processors have signaled their arrival. In the fuzzy barrier scheme, a processor can signal ahead of its arrival at the barrier point, thus allowing it to execute instructions before waiting. A compiler can schedule signals at the earliest possible point in order to maximize processor utilization. Although fuzzy barriers offer improved performance, they still suffer from some of the same disadvantages of barrier synchronization. The overhead of accumulating and transmitting information globally still scales as the log of the number of processors. In addition, the number of instructions that can be scheduled between signaling and waiting is dependent on the particular program. If accesses that require the barrier cannot be moved very far apart at compilation, then processors still spend a large amount of time idle.

A point-to-point synchronization scheme for `DOACROSS` loops is presented in [MP87]. Even though all iterations of a `DOACROSS` loop can be executed in parallel, dependences can exist between iterations. In Figure 1-1a, the definition and use of elements of array `a` in different iterations imply that synchronization must be performed between those iterations. A compiler can automatically insert synchronization primitives (represented as boldface pseudocode) for any such dependences and thereby allow all loop iterations to be executed in parallel without implicit scheduling constraints. The same dependence patterns that exist between iterations of `DOACROSS` loops can also occur with `DOALL` loops as shown in Figure 1-1b. Before reading an element of the array `a`, a processor must synchronize with the write event of that element which occurs in a previous iteration of `i`. Consequently, the analysis done in this thesis must deal with all the issues that arise in synchronization within `DOACROSS` loops. In addition, synchronization across `DOALL` loops requires consideration of dependences between separate loops, which is not considered in [MP87].

```
doacross (i=1,100) {
  a[i] = ...;
  synch with iteration i-5
  ... = a[i-5];
}
```

(a)

```
do (i=1,100) {
  doall (j=1,50)
    a[i,j] = ...;
  doall (j=1,50) {
    synch with iteration i-5,j
    ... = a[i-5,j];
  }
}
```

(b)

Figure 1-1

While synchronization for `DOACROSS` loops requires study of dependences across loop iterations, dependences within a loop iteration or within a general sequence of statements are considered in [CHH89]. A sequence of statements can be mapped into a directed acyclic graph of code blocks with edges representing dependences between blocks. Since each block can be executed by a different processor, synchronization must be performed for each edge in the graph. In Figure 1-2a, the definition and use of variable `a` by different processors requires synchronization between the writing and reading statement blocks. Such dependences between different statements in a sequence also arise when one considers `DOALL` loops. As shown in Figure 1-2b, the definition and use of array `a` also requires synchronization between two different statements in a sequence. In general, for any situation that arises in DAG dependences, an equivalent scenario exists in the context of `DOALL` loops. In addition, synchronization between `DOALL` loops must be concerned with groups of processors that execute each loop rather than merely synchronizing between single processors that execute each node in a DAG.

```
cobegin
  block 1
    a = ...;
  ...
  block 5
    synch with block 1
    ... = a;
end
```

(a)

```
doall (j=1,100)
  a[j] = ...;
...
synch with first loop
doall (j=1,100)
  ... = a[j];
```

(b)

Figure 1-2

1.2 Problem identification

Despite its disadvantages, the barrier synchronization is the simplest and most general method of forcing correct ordering of execution in parallel programs. However, many loop-based programs contain array references that are generally linear functions of loop indices, thus providing statically-obtainable dependence information between individual elements [SLY89]. This thesis aims to use that dependence information to implement point-to-point synchronization schemes which can reduce the costs associated with barrier synchronization.

```
do (i=1,100) {
  doall (j=1,256)
    b[j] = a[j];                        /* S1 */
  Barrier synch #1
  doall (j=1,256)
    a[j] = (b[j-1] + b[j+1]) * .5;      /* S2 */
  Barrier synch #2
}
```

Figure 1-3

Consider the code fragment in Figure 1-3. Let us assume that each `DOALL` iteration `j` is performed on a separate processor P_j on a shared-memory machine and arrays `A` and `B` are partitioned similarly. If no synchronization is performed, one can imagine the scenario where processor P_1 assigns to `B[1]`, then assigns to `A[1]`, then assigns to `B[1]` again before processor P_2 can read the first value of `B[1]`. Consequently, the result of a program can be incorrect due to data dependence violations. To rectify this problem, a barrier synchronization is typically inserted after each `DOALL` loop as indicated in the above example. This solution has the effect of serializing the execution of `DOALL` loops, thus providing correct if not efficient execution. In order to place synchronizations more strategically, data dependence analysis must be performed.

A data dependence arises when the order of two accesses to a memory location must be preserved in order to ensure correctness. Since two read accesses do not require an ordering, a dependence only occurs when one of the accesses is a write to memory. Dependences can be classified into three types:

- *Flow dependence*: a write must be performed before a read.
- *Anti-dependence*: a read must be performed before a write.
- *Output dependence*: a write must be performed before another write.

In order to specify exact iterations of the program, invocations of statements will be labeled by the value of loop indices. For example, the invocation of statement `S1` in Figure 1-3 with `I=5` and `J=6` will be labeled as $\texttt{S1}(5,6)$. A statement invocation $\texttt{S1}(i)$ that is flow-dependent on $\texttt{S2}(i')$ is written as $\texttt{S1}(i)\ \delta^f\ \texttt{S2}(i')$, an output dependence is indicated as $\texttt{S1}(i)\ \delta^o\ \texttt{S2}(i')$, while $\texttt{S1}(i)\ \overline{\delta}\ \texttt{S2}(i')$ represents an anti-dependence. The following dependences exist for Figure 1-3 and are illustrated in Figure 1-4.

$$\texttt{S2}(\texttt{i},\texttt{j})\ \delta^f\ \texttt{S1}(\texttt{i}+1,\texttt{j}) \tag{Δ_1}$$

$$\texttt{S1}(\texttt{i},\texttt{j})\ \delta^f\ \texttt{S2}(\texttt{i},\texttt{j}+1) \tag{Δ_2}$$

$$\texttt{S1}(\texttt{i},\texttt{j})\ \delta^f\ \texttt{S2}(\texttt{i},\texttt{j}-1) \tag{Δ_3}$$

$$\texttt{S1}(\texttt{i},\texttt{j})\ \delta^o\ \texttt{S1}(\texttt{i}+1,\texttt{j}) \tag{Δ_4}$$

$$\texttt{S2}(\texttt{i},\texttt{j})\ \delta^o\ \texttt{S2}(\texttt{i}+1,\texttt{j}) \tag{Δ_5}$$

$$\texttt{S1}(\texttt{i},\texttt{j})\ \overline{\delta}\ \texttt{S2}(\texttt{i},\texttt{j}) \tag{Δ_6}$$

$$\texttt{S2}(\texttt{i},\texttt{j})\ \overline{\delta}\ \texttt{S1}(\texttt{i}+1,\texttt{j}+1) \tag{Δ_7}$$

$$\texttt{S2}(\texttt{i},\texttt{j})\ \overline{\delta}\ \texttt{S1}(\texttt{i}+1,\texttt{j}-1) \tag{Δ_8}$$

For a particular processor partitioning scheme, there exists an ordering on the execution of some statement invocations. When two statement invocations are assigned to the same processor, the order of their execution is predetermined. Let `S1(i,j)` $<$ `S2(i',j')` denote the fact that `S1(i,j)` must execute before `S2(i',j')`. Note that the $<$ relation is anti-reflexive and transitive. If we assume that each `DOALL` iteration `j`

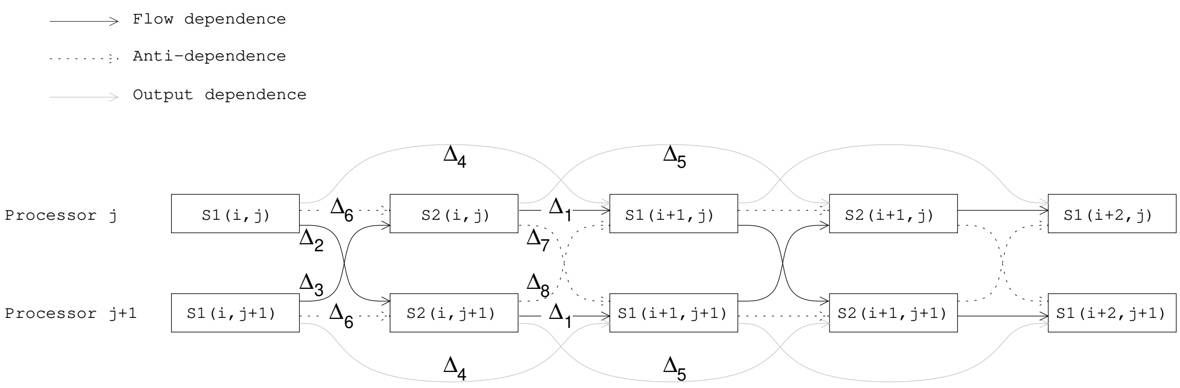


Figure 1-4: Dependence graph for Figure 1-3

of the current example is assigned to processor P_j, then the following ordering arises:

$$\texttt{S1(i,j)} < \texttt{S2(i,j)}$$

$$\texttt{S2(i,j)} < \texttt{S1(i+1,j)}$$

When processor execution obeys this ordering, some dependences are automatically satisfied, such as Δ_1, Δ_4, Δ_5, and Δ_6 in the current example. The remaining dependences Δ_2 and Δ_3 are satisfied by barrier #1 and Δ_7 and Δ_8 are satisfied by barrier #2. If point-to-point synchronization can be performed for those dependences, then the barrier synchronizations can be eliminated.
Figure 1-5

shows execution profiles of the above example on a 16-processor machine. The barrier-synchronization profile uses a tree-based software barrier which requires around 450 cycles. Dark areas represent non-synchronization processing while light areas represent idle time waiting for or performing synchronization. One can see that the 450-cycle overhead for global propagation adds significantly to the overall running time of the application. Moreover, one can also observe that the point-to-point scheme allows for more computation skew among processors which can improve performance in other applications.

1.3 Approach

In order to reduce synchronization costs in loop-based parallel programs, this thesis proposes replacing barrier synchronizations with point-to-point synchronization schemes. The realization of this goal involves careful study of the topics outlined below.

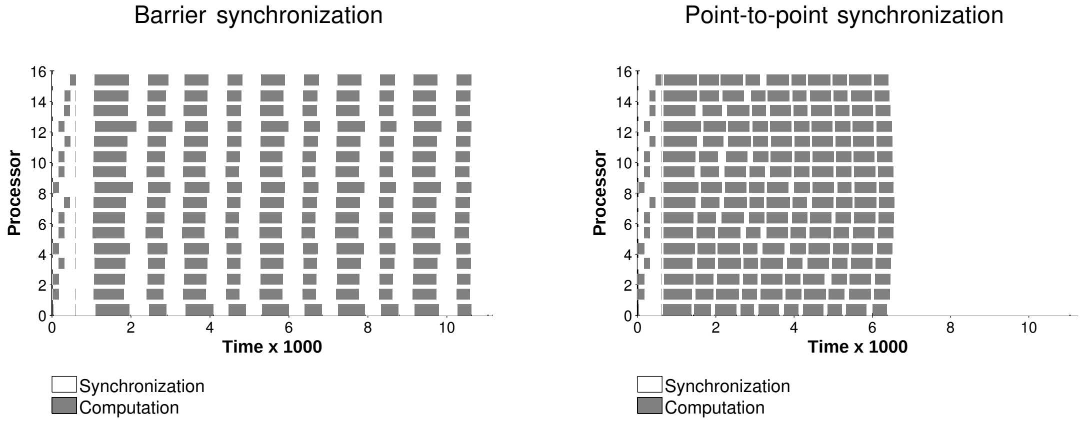


Figure 1-5: Execution of 5 iterations of Figure 1-3

1.3.1 Synchronization variables

Point-to-point synchronization can be implemented by the use of a shared variable which indicates the current loop iteration of each processor as in [MP87]. Before fully completing each `DOALL` iteration in the previous example, each processor updates a synchronization variable to indicate that it has finished that particular iteration.

```
do (i=1,100) {
  doall (j=1,256) {
    wait until sync2[j-1] = i-1 and sync2[j+1] = i-1
    b[j] = a[j];                          /* S1 */
    sync1[j] = i;
  }
  doall (j=1,256) {
    wait until sync1[j-1] = i and sync1[j+1] = i
    a[j] = (b[j-1] + b[j+1]) * .5;        /* S2 */
    sync2[j] = i;
  }
}
```

Figure 1-6

In Figure 1-6, the synchronization arrays `sync1` and `sync2` are partitioned like the arrays `a` and `b`, so for example, processor P_j "owns" element `sync1[j]`. For any dependence, a processor executing the statement that is on the right of the dependence must wait until a processor has executed the statement on the left of the dependence. This is accomplished by setting and waiting for appropriate values in the `sync` arrays. Although there is a spin-locking action on elements of the `sync` arrays, no extra network traffic is induced on machines with caching schemes that allow shared copies of variables.

1.3.2 Computing statement dependences

In order to determine processor synchronization requirements, dependences between statements of a program need to be computed. The calculation of such data dependence information can be adapted primarily from two areas of research: sequential data-flow analysis and array-dependence analysis for parallelizing `DO` loops.

Standard data-flow analysis techniques [ASU86] can provide definition-use chains for computing flow dependences. The algorithms can also be adapted to generate information necessary for calculating output and anti-dependences. Unfortunately, these techniques are primarily concerned with scalar variables and pay little attention to flow information on individual array elements. In order to effectively compute dependence information for point-to-point synchronizations, the scalar flow analysis framework must be augmented to operate on arrays and subsets of arrays as specified by linear index functions. Although questions involving relations on such sets requires the application of linear diophantine equation theory, previous work in the field of array-dependence analysis can be used to provide the answers.

A large amount of work has been done on calculating dependences between arrays for loop parallelization [Ban88][Wol89]. However, such works are primarily concerned with dependences within a loop body rather than dependences between separate loops that require more detailed attention to program control flow. In addition, these works are only concerned with the question of whether a dependence exists between two statements. In order to compute point-to-point synchronizations, this question needs to be extended to include the calculation of the exact data elements that are involved in a dependence, as discussed in the following section.

1.3.3 Computing processor dependences

Point-to-point synchronization can replace barrier synchronization effectively in cases where data dependences can be determined at compile time. In other words, synchronization should only be inserted when the source and sink processors can be computed efficiently. Although the above array data flow analysis provides the information on dependences between statements, it does not yield information on dependences between processors. In order to compute interprocessor dependence relations, more analysis must be done on array access patterns.

In the examples presented thus far, dependence relations between processor can be derived in a straightforward manner from the array accesses. Indeed, when two array indices contain linear functions of the same loop index, dependence relations can be computed easily from the linear functions. However, more difficult cases exist. Array access patterns can relate loop indices that occur at different nesting levels and in different loop nests. Loop indices can occur multiply in some array references and not at all in others. Some of these situations are illustrated in Figure 1-7.

```
do (i=1,100) {
  do (j=1,100) {
    doall (k=1,100)
      a[i,i,k+1] = ...;
    ...
    doall (k=1,100)
      ... = a[i-3,k,x];
  }
}
```

Figure 1-7

1.3.4 Optimizing point-to-point synchronization

Point-to-point synchronization can be inserted once dependence information is obtained. However, this insertion process must be done intelligently to maintain the ultimate goal of faster program execution. Since testing of synchronization variables can result in additional network traffic and increased latency, synchronizations produced by redundant dependences must be eliminated.

To reduce execution time, the checking of synchronizations must result in as little delay as possible. Consequently, setting the values of synchronization variables should be done at the earliest possible point. With straight-line code, this problem seems trivial since one could easily enforce the constraint that synchronizations be set immediately after the source of the dependence is satisfied. However, in the presence of conditionals, each synchronization variable must be set in every control path to any check of that variable. In other words, if the source of a dependence does not dominate the sink, then the corresponding synchronization variable must be set in other paths to the sink. Any scheme to reduce idle time must obey this condition for correctness.

In previous examples, synchronization is performed for every dependence that exists in the program. Several steps can be taken to reduce the number of synchronization operations required. In typical programs, many dependences are automatically satisfied by synchronization provided for other dependences. As an example, assume that S_1, S_2, S_3, and S_4 are statements in a straight-line program such that each S_i precedes S_{i+1}. If a dependence Δ_1 exists between S_2 and S_3 and another dependence Δ_2 exists between S_1 and S_4, and Δ_1 and Δ_2 have the same processor relationships, then there is no need to support Δ_2 with synchronization since the processors are already synchronized due to Δ_1. Thus when two processors are already synchronized due to other dependences, then a dependence between the two processors is redundant and can be eliminated. Reduction of redundant synchronization has been studied in the context of `DOACROSS` loops in [MP87] and [KS91]. In these works, redundant dependences can be defined as duplicate edges in the transitive closure of the dependence graph. Again, as applied to this thesis, the analysis is required to be more complex due to interactions between data dependences and control flow. In this context, calculating the minimum number of dependences for a given program is a problem of both theoretical and practical interest. A related optimization to the above involves replicating variables to cause output and anti-dependences to become redundant, as discussed in the following section.

1.3.5 Variable replication

In a single-assignment language, output and anti-dependences cannot occur because variables can hold only one value. In an imperative language, variables can be renamed or replicated to avoid these dependences in certain circumstances, although at a cost in memory usage [Kum87].

An example is presented here to illustrate variable replication as well as removal of redundant dependences. Consider a transformation of the program of Figure 1-3 as illustrated by Figure 1-8. In this version, a different version of each array is kept for each outer iteration, thus resulting in each array element being assigned a value only once. The anti-dependences from `S2` to `S1` (Δ_7 and Δ_8) no longer appear since each update of the array `b` changes a different location. Therefore the second barrier synchronization can be eliminated altogether. In addition, if the target machine supports full/empty bit synchronization, then the flow dependences from S1 to S2 (Δ_2 and Δ_3) also do not require additional synchronization.

```
do (i=1,100) {
  doall (j=1,256)
    b[j][i] = a[j][i-1];                          /* S1 */
  doall (j=1,256)
    a[j][i] = (b[j-1][i] + b[j+1][i]) * .5;      /* S2 */
}
```

Figure 1-8

Instead of maintaining many different versions of the arrays, consider now the possibility of obtaining the same benefits from a smaller number of replicated arrays. With the current example, the same result can be achieved by using only two different copies of each array, as illustrated by Figure 1-9.

```
do (i=1,100) {
  lastk = k;
  k = i mod 2;
  doall (j=1,256)
    b[j][k] = a[j][lastk];                          /* S1 */
  doall (j=1,256)
    a[j][k] = (b[j-1][k] + b[j+1][k]) * .5;        /* S2 */
}
```

Figure 1-9

The following dependences are introduced:

$$\mathtt{S1}(\mathtt{i},\mathtt{j})\ \delta^{o}\ \mathtt{S1}(\mathtt{i}+2,\mathtt{j}) \qquad (\Delta_4)$$

$$\mathtt{S2}(\mathtt{i},\mathtt{j})\ \delta^{o}\ \mathtt{S2}(\mathtt{i}+2,\mathtt{j}) \qquad (\Delta_5)$$

$$\mathtt{S1}(\mathtt{i},\mathtt{j})\ \overline{\delta}\ \mathtt{S2}(\mathtt{i}+1,\mathtt{j}) \qquad (\Delta_6)$$

$$\mathtt{S2}(\mathtt{i},\mathtt{j})\ \overline{\delta}\ \mathtt{S1}(\mathtt{i}+2,\mathtt{j}+1) \qquad (\Delta_7)$$

$$\mathtt{S2}(\mathtt{i},\mathtt{j})\ \overline{\delta}\ \mathtt{S1}(\mathtt{i}+2,\mathtt{j}-1) \qquad (\Delta_8)$$

Dependences Δ_1, Δ_2, and Δ_3 remain unchanged from the original version. And once again, dependences Δ_4, Δ_5, and Δ_6 are satisfied by sequential execution of statements on each processor. However, dependence Δ_7 is redundant as illustrated by Figure 1-10. $\texttt{S2}(\texttt{i},\texttt{j}) < \texttt{S1}(\texttt{i}+1,\texttt{j})$ due to execution ordering, $\texttt{S1}(\texttt{i}+1,\texttt{j}) < \texttt{S2}(\texttt{i}+1,\texttt{j}+1)$ because Δ_2 is satisfied, and $\texttt{S2}(\texttt{i}+1,\texttt{j}+1) < \texttt{S1}(\texttt{i}+2,\texttt{j}+1)$ due to execution ordering. Therefore, $\texttt{S2}(\texttt{i},\texttt{j}) < \texttt{S1}(\texttt{i}+2,\texttt{j}+1)$ and Δ_7 is automatically satisfied. Intuitively, it is impossible for $\texttt{S2}(\texttt{i},\texttt{j})$ to execute after $\texttt{S1}(\texttt{i}+2,\texttt{j}+1)$ because an earlier statement in processor P_j depends on output from $\texttt{S2}(\texttt{i},\texttt{j})$. Likewise, Δ_8 is satisfied, and synchronization only needs to be performed for dependences Δ_2 and Δ_3. Therefore, doubling the storage requirements of the arrays results in the elimination of the anti-dependences in this example. Note that the same results can be obtained by only replicating array `b` since elements of array `a` are never shared.

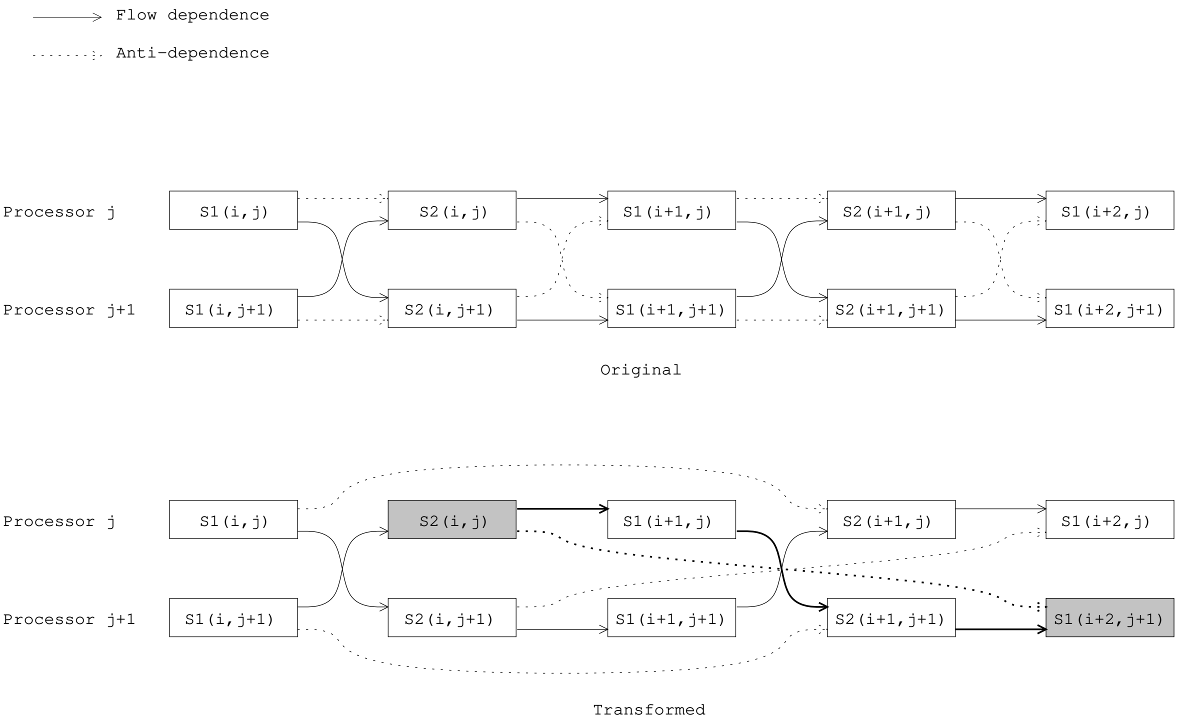


Figure 1-10: Eliminating anti-dependences

1.4 Thesis outline

The remainder of the thesis is organized as follows: Chapter 2 presents the background and assumptions used in the rest of the thesis. Chapter 3 shows how array flow analysis and dependence testing can be used to compute dependences between statements. Chapter 4 then presents schemes for detecting dependences between processors and deriving synchronization to support such dependences. Chapter 5 discusses several optimizations to remove redundant dependences through dynamic programming techniques and eliminating false dependences by array replication. Finally, some results of an implementation are presented in Chapter 6, followed by a discussion of future topics and the conclusion.

Chapter 2

Background

2.1 Language description

In order to illustrate the optimizations presented in this thesis, a skeletal language is now introduced. However, it is important to note that these optimizations are applicable to the general array-based data-parallel programming style rather than any particular language. Indeed, a program using such a style in any language can probably benefit from these synchronization-reduction techniques if the proper compilation mechanisms are added to support salient features of the particular language.

$$
\begin{aligned}
S ::= {} & V = E; \\
& \mid \texttt{if (}V\texttt{)}\ S\ \texttt{else}\ S \\
& \mid \texttt{while}\ (V)\ S \\
& \mid \texttt{do}\ (V = K, K, K)\ S \\
& \mid \texttt{doall}\ (V = K, K, K)\ S \\
& \mid \texttt{\{}\ S\ \ S\ \texttt{\}}
\end{aligned}
$$

Figure 2-1: Language syntax

The syntax of the language is shown in Figure 2-1. The terminal V is assumed to be a variable, K is an integer variable or constant, and E is an expression. Although the sequential looping constructs `DO` and `WHILE` are semantically very similar, they are both included since a large amount of analysis is done on indices of `DO` loops. On the other hand, the `WHILE` statement represents a more general looping construct with a terminating condition and without explicitly specified indices. The sequence operator $\{\ S\ \ S\ \}$ is restricted to contain two statements for ease of proofs in later chapters. A general sequence of many statements can be viewed as a cascade of many two-statement sequences.

In addition to standard control-flow constructs, the `DOALL` construct is provided for specification of explicit parallelism. The use of a DOALL statement is a declaration that all iterations of the loop can be executed in parallel. Semantically, DOALL execution behaves as if barrier synchronizations existed before and after the DOALL. Furthermore, the body of each DOALL is not assumed to be atomic. All iterations can be invoked simultaneously and can compete for the same resources. Consequently, the program shown in Figure 2-2 is incorrect since other iterations can be started and finished between the fetch and assignment of sum in a particular iteration. Although alternate models of executing DOALL loops exist in the literature which allow atomicity of iterations or copy-in/copy-out semantics [CHH89], this thesis focuses only on the simpler semantics presented above. For simplicity of presentation, the DOACROSS construct is omitted in most of the discussion of this thesis.

```
sum = 0;
doall (i=1,100,1)
  sum = sum + a[i];
```

Figure 2-2: Sum the elements of an array

In this thesis, DOALL loops are assumed to be partitioned at compile time. Thus for a particular loop with index variable i, it is assumed that the mapping from the value domain of i to the processor space has been done either by the programmer or by an earlier phase of the compiler. The automatic partitioning of loop iterations into processors is a topic of active research [Sar87][AH91]. In certain situations including those where the index set of a loop cannot be determined statically, a dynamic scheduling scheme must be used. However, such cases are not considered here. A detailed specification of the execution of statically-scheduled DOALL loops is given in Chapter 4.

2.2 Control flow graph

The *control flow graph* of a program can be defined to form a framework for managing relationships between statements. Each node in the graph represents a statement in the program. If it is possible that execution of statement S_1 can be followed by statement S_2, then a directed edge exists from S_1 to S_2. Figure 2-3 shows how a flow graph can be constructed from sequential constructs in the language.

From [ASU86], an edge in the control-flow graph is a *forward edge* if it is part of a spanning tree formed from a particular depth-first traversal of the graph. Forward edges

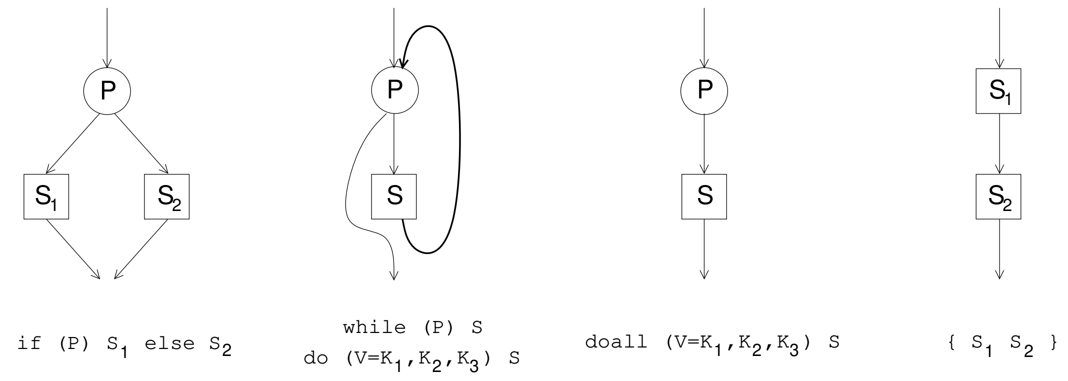


Figure 2-3: Control flow graph for language constructs

are represented by solid arrows in the figure. A statement S *precedes* or is a *predecessor* another statement S' if there is a path composed of forward edges from S to S'. The statement S' is said to be a *successor* of S. A *back edge* is an edge from a statement S to a predecessor of S. The back edges are represented by highlighted arrows in the figure. Since forward edges form a tree, no forward edge can be a back edge. In a general program, *cross edges* also exist that are neither forward nor back edges, but they do not occur in programs that use the above syntax. A cross edge can be produced by a forward jump such as a non-local loop exit.

A statement S *dominates* or is a *dominator* of S' if any path from the start of the program to S' must pass through S. If S dominates S', then S precedes S'. Likewise, S *post-dominates* S' if any path from S' to the end of the program must pass through S. If S post-dominates S', then S is a successor of S'. We introduce the concept of *relative dominance*, a more general notion of dominance. A statement S dominates S' relative to S'' if any path from S'' to S' must pass through S. A statement S post-dominates S' relative to S'' if any path from S' to S'' must pass through S. Thus S dominates S' if it dominates S' relative to the start of the program and S post-dominates S' if it post-dominates S' relative to the end of the program.

A `DOALL` statement can be viewed as specifying a collection of statements with an entry node S and an exit node S' such that the collection is composed of exactly the statements that are dominated by S and post-dominated by S'. Clearly, S is a predecessor and S' is a successor of all other nodes in the collection. The collection of nodes is executed once for each iteration value of the loop. Note that there is not a back edge from S' to S since all iterations of a `DOALL` can be all executed in parallel.

Using the structure of the program, we define the *child* of a statement as the inner statement of conditional, loop, or sequence statements. Each of those statements in turn is a *parent* of the inner statement. A statement S is an *ancestor* of S' if there are statements $\{S_1, \ldots, S_n\}$ such that $S = S_1$, each S_{i+1} is a child of S_i, and $S' = S_n$. Each statement S is also an ancestor of itself. S is a *descendant* of S' if S' is an ancestor of S. Note that if S is a parent of S', then S dominates S' and S precedes S' since the predicates of conditionals and loops are executed before the body. We also use the term S *encloses* S' if S is an ancestor of S'.

The above definitions can be used to show that if two statements do not precede each other, then there must be a conditional that encloses them:

Lemma 2.1: If S_1 and S_2 are statements such that $S_1 \neq S_2$, S_1 does not precede S_2 and S_2 does not precede S_1, then there exists a statement S such that S_1 and S_2 are descendants of S and S is a conditional.

Proof: Let $\{S_1^k, \ldots, S_{n_k}^k\}$ be ancestors of S_k such that S_i^k is a parent of S_{i+1}^k. Then $S_1^1 = S_1^2$ since they are both equal to the outermost program statement. Let j be the highest integer such that $S_j^1 = S_j^2$. Let $S = S_j^1$. Then S_1 and S_2 are both descendants of S. If $S = S_1$, then $S \neq S_2$ and S precedes S_2, which implies a contradiction. Likewise, $S \neq S_2$. Therefore S_{j+1}^1 and S_{j+1}^2 are distinct statements that are children of S, which implies that S is either a conditional or a sequence. If S is a sequence, then a precedence relationship exists between S_{j+1}^1 and S_{j+1}^2, which implies that one exists between S_1 and S_2. Therefore S is a conditional. □

A *partial ordering* is a relation $<$ on a set A with the following properties on set elements:

$$a \not< a \qquad \text{(anti-reflexive)}$$
$$a < b \Rightarrow b \not< a \qquad \text{(anti-symmetric)}$$
$$a < b \ \text{ and } \ b < c \Rightarrow a < c \qquad \text{(transitive)}$$

This relation is sometimes known as a strict partial ordering. As an example, the precedence of statements above is a partial ordering: It is anti-reflexive because there are no forward edges from a node to itself, anti-symmetric because there are no cycles in the tree of forward edges, and transitive because the concatenation of two paths of forward edges is itself a path.

2.3 Machine model

This thesis assumes a cache-coherent shared-memory interface found on machines such as Alewife [Aga91] and Dash [Len92]. Such a multiprocessor can be modeled as a collection of processors $P = \{p_1, \ldots p_n\}$ and a shared pool of memory units that can be accessed through a network. In addition, each processor is associated with a data cache to reduce memory-access latency. The resolution and maintenance of multiply-cached copies of data is performed by a cache-coherence protocol [CFKA90]. Processors are completely independent from each other in the sense that they are able to execute completely different programs from each other. However, the execution model assumed here is one in which all processors execute the same program, although on different data. This is commonly called the Single-Program-Multiple-Data (SPMD) model in the literature.

Chapter 3

Statement dependences

3.1 Introduction

Pursuing the goal of replacing barrier synchronizations with point-to-point synchronizations requires that detailed information about data dependences be computed. This knowledge can be derived from array data flow analysis, an adaptation of conventional scalar data flow analysis. Since this thesis focuses primarily on the domain of array and loop-based data-parallel programs, it is very important that accurate information on array usage be obtained. Conventional data flow analysis techniques [ASU86] tend to treat arrays as single variables. A reference to any element of an array is considered a reference to the entire array. Clearly, such conservative analysis cannot be used to derive dependences needed for point-to-point synchronization. Instead, the array data flow analysis technique can be used to monitor accesses to individual elements of an array. Accurate approximations of values of array indices must be available to yield needed information on array usage. Consequently, the important topic of deducing values of array indices is outlined in the first section. Subsequent sections discuss array flow analysis and its potential uses, as well as its application to dependence detection.

3.2 Propagation of linear induction variables

The flow analysis technique outlined in this chapter focuses on arrays whose indices are linear loop induction variables. In other words, relevant array accesses are those whose indices can be represented by linear functions of loop indices. A constant array index can be viewed as a linear function with a multiplicative factor of zero. However, detecting whether an expression is a linear function of a loop index is not a trivial problem. Consider the program in Figure 3-1. It is obvious in this case that the assignment to array `b` in statement `S1` uses an array index that is linear with respect to the loop index `i`. Furthermore, the value of the array index `k` is always equal to `2i+5`. However, it is not clear how this information can be deduced in a general manner. Fortunately, existing value propagation algorithms can be adapted to propagate linear loop induction variables. In the literature [PW86], this optimization is known as *forward substitution*.

```
for (i=1,100) {
  j=2i+1;
  if (a[i]>0)
    k=2i+5;
  else
    k=j+4;
  b[k] = c[i];              /* S1 */
}
```

Figure 3-1

3.2.1 Value lattices

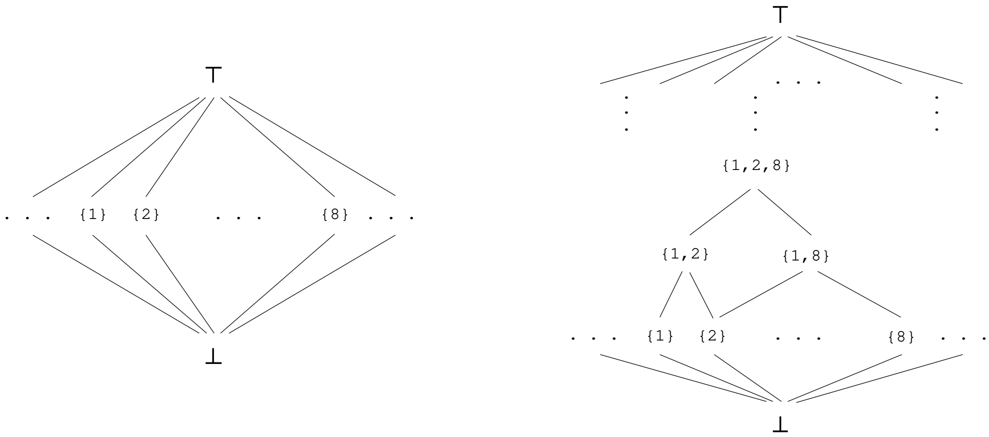
For each lexical expression in a program, let the *value set* of the expression be the set of values that it can take on during program execution. Since we are primarily interested in deducing information on array indices that are linear functions of loop indices, value sets are subsets of the integer space and are derived only for scalar variables. A value set can be viewed as an approximation $\mathcal{A}(E)$ of the value of an expression E and represented as an element on a value lattice. Propagation of both constants and linear induction variables can then be viewed as propagation of value sets using different lattices.

A *lattice* is defined as a partial ordering on a set such that there exists an element that is greater than all others ($\top$) and an element that is less than all others ($\perp$). In the current context, lattice elements correspond to value sets and each lattice represents a partial ordering on the set of value sets. A value set e_1 is greater than another set e_2 if e_1 is a superset of e_2. In constant propagation, $\top_c$ can be viewed as the set of all integers and $\perp_c$ can be viewed as the empty set.

As illustrated in Figure 3-2a, the *single integer lattice* for conventional constant propagation consists of three levels: a bottom element ($\perp_c$) which indicates that no approximation exists for an expression, a top element ($\top_c$) which indicates that the expression can take on any possible value, and sets of single integers that correspond to constant values. Conventionally, constant propagation is performed in order to avoid computing and fetching values that are known to be constants at compilation. If a value is not constant, then the compiler does not benefit from any additional information and the value approximation can be set to $\top_c$. However, the optimizations presented in this thesis can make use of approximations that represent the union of several values. Consider the following code segment:

```
if (p)
  a = 1;
else
  a = 2;
```

In conventional constant propagation, the value of a after the conditional is deduced to be $\top_c$ since it is neither always 1 nor always 2. By allowing unions of constants in the lattice, the value of a can be deduced to be "1 or 2" which is a much more accurate approximation than $\top_c$. A *multiple integer lattice* is a single integer lattice augmented with unions of constants and is shown in Figure 3-2b. The height of this lattice can be forced to be finite by imposing the restriction that the size of each set cannot exceed some limit H. Any expression whose value set requires more than H integers can be approximated as $\top_c$. This restriction enables propagation algorithms that use the lattice to terminate after each variable has gone through $O(H)$ approximations.



(a) Single integer lattice

(b) Multiple integer lattice

Figure 3-2: Constant propagation lattices

For typical programs, the representation of value sets as sets of integers can quickly become unwieldy for expressions that take on many values. Since multiple executions of a single array reference can access a region of an array, it may be possible to abbreviate a value set as an integer range. Indeed, this sort of analysis has been studied in the context of eliminating unnecessary array-bounds checking [Har77][MCM82][Gup90]. If an array index can be approximated by a range that is within the array bounds, there is no need to perform a bounds check. Using integer ranges as lattice elements is ideal for this kind of optimization since the only information required are the minimum and maximum values of each array index.

Although integer ranges can form an effective representation of lattice elements, such approximations do not model effectively the case when an array is being accessed with a stride greater than 1. If an array reference accesses odd indices in an array and another accesses even indices, the two accesses do not interfere with each other. In addition, two accesses with stride 1 may not interfere even if their ranges intersect. Consider the code fragment in Figure 3-3. Since statement `S2` uses an old element of array `a` and not the one that is defined in `S1`, there are no flow dependences from `S1` to `S2` even though the ranges corresponding to `j` and `j+2` intersect. Clearly, information about loop indices must be stored to detect these dependences.

```
do (i=4,100) {
  j = i-3;
  a[j] = b[i];          /* statement S1 */
  c[i] = a[j+2];        /* statement S2 */
}
```

Figure 3-3

A linear induction variable lattice can be defined with a structure that is similar to the multiple integer lattice. Parts of such a lattice are shown in Figure 3-4. Immediately above $\perp_{iv}$ are single linear functions of various loop indices and above those linear induction variables are sets of multiple induction variables. The element $\top_{iv}$ can be viewed as the collection of all linear induction variables. From this point on, linear induction variables will form the basic elements in a value set. Note that for a value set e_1 to be a strict superset of another lattice element e_2 and thus be higher in the lattice than e_2, e_1 must have more induction variables than e_2. Again, we can put a limit H on the number of induction variables allowed in a value set, thus forming a lattice of height $H + 2$.

Note that functions of only one loop index are present in the linear induction variable lattice. Linear functions of multiple loop indices are not supported. Also, detection of induction variables that are not defined directly from loop indices [Wol92] is not done. These topics are viewed as somewhat orthogonal to the approach outlined here. Thus their inclusion in these algorithms can be made in a real system by incorporating relevant techniques in the literature.

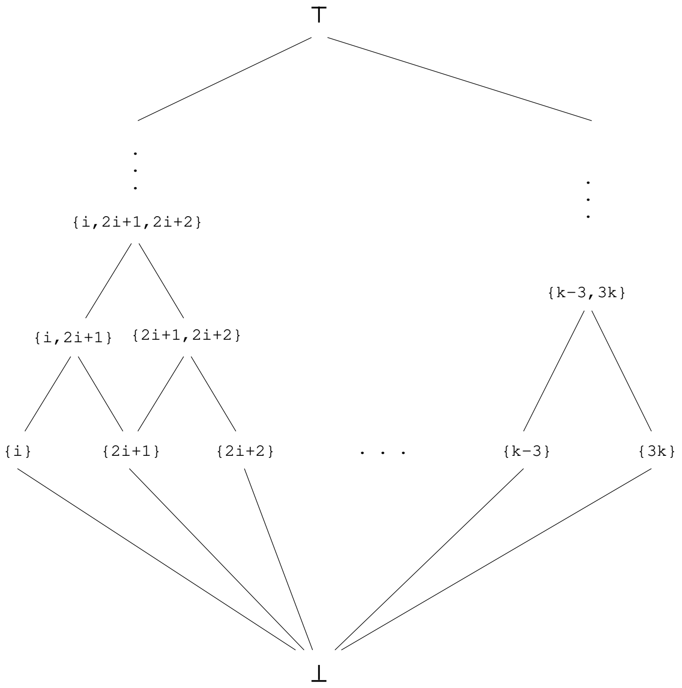


Figure 3-4: Linear induction variable lattice

3.2.2 Propagation of linear induction variables

The algorithm presented here for the propagation of linear induction variables is based on previous work on symbolic value propagation done by Reif and Lewis [RL86]. Wegman and Zadeck [WZ91] show that in the context of constant propagation, derived constant information can aid in the flow analysis as well. In this section, an algorithm is shown for propagation of linear induction variables on scalars using a sparse flow graph representation.

Compiler analysis to discover linear relationships among variables was first studied by Karr [Kar76]. Linear relationships among variables can also be shown to be derivable in the general framework of abstract interpretation [CH78][CC77]. These approaches focus towards a general treatment of the problem rather than developing an efficient algorithm for the language features used here.

3.2.2.1 SSA form

Value propagation can be done efficiently on a sparse flow graph representation using static single assignment form. The term *single assignment* is typically used to represent an execution model where only one assignment is done for each variable during the entire program execution. Similarly, a program is in *static single assignment* form when each variable is assigned by only one statement [Cyt91]. Note that each variable can still be assigned many times dynamically due to the presence of loops. However, each of those assignments is done in the same statement.

A program in SSA form has at most one assignment statement for each variable. Any program can be transformed into SSA form by observing the following rules:

1. At the beginning of the program, assignments are inserted for each variable to initialize the variable to its default value at program startup.

2. Each assignment to a variable `v` is replaced by an assignment to a renamed variable $\mathtt{v_i}$ where `i` is different for every assignment to `v`.

3. For each join point in a program flow graph, if several different names $\mathtt{v_i}$ and $\mathtt{v_j}$ of the same variable reach the join point, then a new assignment is inserted after the join point of the form $\mathtt{v_k} = \phi(\mathtt{v_i},\mathtt{v_j})$. Again, `k` is distinct from all other renamings of `v` in the program. The ϕ form can be viewed as a merge of variable definitions.

4. Each use of a variable is renamed to the name of the definition of the variable that reaches it. This definition is unique since ϕ assignments are inserted at every join point where multiple reaching definitions can arise.

Algorithms for computing SSA form in general are given in [Cyt91] and for structured programs in [RWZ88]. An illustration of the SSA transformation is shown in Figure 3-5.

A *definition-use graph* can be defined as a directed graph with statements as vertices and edges from definitions of variables to their uses. For $\mathcal{N}$ occurrences of a variable in a general program, there can be potentially $O(\mathcal{N}^2)$ definition-use edges corresponding to that variable. In a program transformed to SSA form, definition-use edges can be easily computed by matching each use of a variable with its corresponding definition. The number of def-use edges for each variable in an SSA program is at most $\mathcal{E} + \mathcal{N}$ where

```
doall (i = 1,99,2) {
  j = i * 2;
  if (a[i] > 0)
    j = j + 1;
  b[j] = b[j] / 3;
}
doall (i = 2,100,2)
  c[i] = b[i*2];
```

Original program

```
doall (i1 = 1,99,2) {
  j1 = i1 * 2;
  if (a[i1] > 0)
    j2 = j1 + 1;
  j3 = φ(j1, j2);
  b[j3] = b[j3] / 3;
}
doall (i2 = 2,100,2)
  c[i2] = b[i2*2];
```

Transformed program

Figure 3-5

$\mathcal{E}$ is the number of edges in the program control flow graph [RL86]. The def-use graph of a program in SSA form can thus be viewed as a sparse representation of the general definition-use graph.

3.2.2.2 Propagation algorithm

After a program has been transformed into SSA form, linear induction variables can be propagated over the definition-use graph. Throughout the execution of the algorithm, outstanding propagation values are maintained in a work-list of def-use edges. An edge is in the work-list if its definition variable v has approximation $\mathcal{A}(v) \neq \perp_{iv}$ and if the definition has been changed since the last examination of the edge. Associated with each expression E in the program is its lattice element approximation $\mathcal{A}(E)$ which traverses up the lattice as the algorithm proceeds and new values are discovered for the expression.

Recall that a value set is the set of values that an expression may have at run time and that value sets are represented as linear functions of loop indices. Through the propagation process, calculations are performed on these linear functions according to the program text. For a linear function of a loop index of the form $(\alpha\mathtt{i} + \beta)$, a list of rules for linear function calculations is given in Figure 3-6. Constant expressions (γ) can also be viewed as linear functions with $\alpha = 0$. All arithmetic operations with $\top_{iv}$ yield $\top_{iv}$ and all arithmetic operations with $\perp_{iv}$ yield $\perp_{iv}$.

At initialization, the approximation $\mathcal{A}(E)$ of each expression E is set to the lattice element that can be derived immediately from its text. Thus constant expressions and loop index variables are approximated to be the respective constant or loop index. If the value set of the expression text is inconclusive, then $\mathcal{A}(E)$ is set to $\perp_{iv}$. If the value

$$(\alpha_1 \mathtt{i} + \beta_1) \text{ ``+'' } (\alpha_1 \mathtt{i} + \beta_2) = ((\alpha_1 + \alpha_2)\ \mathtt{i} + \beta_1 + \beta_2) \quad (1)$$

$$(\alpha_1 \mathtt{i} + \beta_1) \text{ ``}-\text{'' } (\alpha_1 \mathtt{i} + \beta_2) = ((\alpha_1 - \alpha_2)\ \mathtt{i} + \beta_1 - \beta_2) \quad (2)$$

$$(\alpha \mathtt{i} + \beta) \text{ ``}\star\text{'' } \gamma = \alpha\gamma\ \mathtt{i} + \beta\gamma \quad (3)$$

$$(\alpha \mathtt{i} + \beta) \text{ ``/'' } \gamma = \alpha/\gamma\ \mathtt{i} + \beta/\gamma \quad (4)$$

$$\phi(e, \perp_{iv}) = e \quad (5)$$

$$\phi(\perp_{iv}, e) = e \quad (6)$$

$$\phi(e_1, e_2) = \begin{cases} \top_{iv} & \text{if } |e_1| + |e_2| > H \\ e_1 \cup e_2 & \text{otherwise} \end{cases} \quad (7)$$

Figure 3-6: Rules for operations on linear index functions

set cannot be represented by any lattice element, then the expression is approximated as $\top_{iv}$. Note that if E contains no free variables, then $\mathcal{A}(E)$ cannot be $\perp_{iv}$ since its approximation can be determined at compile time. The work-list is then initialized to contain all def-use edges with definition variable v such that $\mathcal{A}(v) \neq \perp_{iv}$. The algorithm then proceeds as follows:

1. Remove a def-use edge from the work-list. If the work-list is empty, then terminate.

2. Let V be the variable corresponding to the removed edge. Let S be the statement where V is used (pointed to by the def-use edge). For each expression E in S, a new approximation of the value of E can be made using the approximation of V at the definition.

3. If S is an assignment statement to a variable V' and its right-hand-side approximation changes in step 2, then all def-use edges for which V' is a definition are added to the work-list.

The following statement proves that the propagation algorithm is correct by showing that any value that can occur at run time for an expression is included in the approximation for that expression.

Claim 3.1: For each expression E, if $\omega(E)$ is the set of values that E can take on at run time, then $\omega(E) \subseteq \mathcal{A}(E)$.

Proof: By contradiction, suppose that for some expression E', $\omega(E') \not\subseteq \mathcal{A}(E')$ during program execution. Then there some earliest execution of an expression E such that $\omega(E) \not\subseteq \mathcal{A}(E)$. Let $V_1, \ldots, V_n$ be free variables in V. Then each previous definition of each V_i produces a value that is in $\mathcal{A}(V_i)$ since E is the earliest execution that violates the subset relation. But then the value of E is in $\mathcal{A}(E)$ because the rules in Figure 3-6 preserve the subset relation. □

Corollary 3.2: At algorithm termination, there can be no executable expression E such that $\mathcal{A}(E) = \perp_{iv}$.
Proof: Since E has a run-time value, $\omega(E) \neq \emptyset$ and thus $\mathcal{A}(E) \neq \perp_{iv}$ due to Claim 3.1. □

The running-time analysis of this algorithm makes use of the height of the lattice. Recall that for a lattice element e_1 to be higher in the lattice than another element e_2, e_1 must have more linear functions than e_2. The key to studying running time involves examining the number of times that each statement is invoked by step 3. For each assignment statement with variable V, step 3 can generate new edges at most $H + 2$ times since each change in the approximation of the ϕ expression involves moving up one level in the lattice. Overall, the number of times that an edge can be placed in the work-list corresponds to the number of def-use edges in the SSA graph times H. From [RL86], there are at most $\mathcal{E}$ def-use edges for each variable in an SSA graph where $\mathcal{E}$ is the number of edges in the control flow graph. Hence, the worst-case running time for the propagation algorithm is $O(H \times \mathcal{N} \times \mathcal{E})$ where $\mathcal{N}$ is the number of variables in a program. From [WZ91], empirical evidence suggests that constant propagation runs in time linear to program size. The typical running time of linear induction variable propagation is expected to also be linear but with an additional multiplicative factor of H.

After all induction variables have been propagated, most expressions that are linear functions of loop indices can be detected. The next compiler phase can then use this information to perform data flow analysis on sections of arrays.

3.3 Flow analysis on arrays

Data flow analysis involves the study of data interaction between different points in a program. In the context of data dependence detection, interactions between definitions and uses of variables are analyzed to determine whether dependences exist. Flow dependences and output dependences arise when previous definitions conflict with current uses and current definitions, respectively. Anti-dependences arise from conflicts between previous uses and current definitions. Consequently, accurate information on the previous uses and definitions that reach a statement is needed to generate useful dependence information. An algorithm is presented in the following section to determine the set of reaching uses and definitions for each statement in a program.

3.3.1 Linear integer sequences

When array flow analysis is performed, there are many operations that need to be performed on value sets such as union, intersection, subtraction, and comparison. Unfortunately, the linear induction variable representation presented in the previous section is unwieldy for certain operations. In Figure 3-7, the definition of `a` in statement `S1` should not progress past the second loop since it is killed by the definitions of `a` in statements `S2` and `S3`. Intuitively, statement `S2` modifies odd indices of `a` and statement `S3` modifies even indices of `a`. However, it is hard to extrapolate the fact that the two definitions cover all indices of `a` from the linear induction variable representation.

```
doall (i=1,200)
  a[i] = b[i];          /* S1 */
do (k=1,10) {
  doall (j=1,100)
    a[2*j-1] = c[j];    /* S2 */
  doall (j=2,200,2)
    a[j] = d[j];        /* S3 */
}
```

Figure 3-7

An alternate representation for linear induction variables is needed to manage array subsets to support cases similar to the previous example. Although dependence analysis requires the preservation of linear induction variables, such a representation is not necessary for the purposes of strictly performing flow analysis. Instead, each array index expression can be approximated by a less specific linear sequence of integers with an associated range. A *linear integer sequence* can be represented as a 3-tuple $\langle\!\langle n_{lo}, n_{hi}, n_{step}\rangle\!\rangle$ where n_{lo} and n_{hi} are the low and high limits of the sequence and n_{step} is the stride of the sequence. Such a tuple represents the set of all integers n of the form $n_{lo} + kn_{step}$ such that $k \geq 0$ and $n \leq n_{hi}$. For example, the 3-tuple $\langle\!\langle 10, 98, 2\rangle\!\rangle$ represents all even

2-digit integers. Note that the high value of the tuple must itself be in the representative set. In Figure 3-7, the tuples for statements `S2` and `S3` are $\langle\langle 1, 199, 2\rangle\rangle$ and $\langle\langle 2, 200, 2\rangle\rangle$, respectively. Their union forms the tuple $\langle\langle 1, 200, 1\rangle\rangle$ which is a superset of the tuple in statement `S1`.

A set-inclusion ordering on linear integer sequences can be defined as follows:

$$\langle\langle m_{lo}, m_{hi}, m_{step}\rangle\rangle \subseteq \langle\langle n_{lo}, n_{hi}, n_{step}\rangle\rangle \iff$$
$$m_{step} = k_1 n_{step} \text{ and } m_{lo} = n_{lo} + k_2 n_{step} \text{ and } m_{hi} \leq n_{hi} \text{ for integers } k_1, k_2 \geq 0$$

The ordering defines a lattice with $\top_{is}$ as all integers and $\perp_{is}$ as the empty set.

In order to convert value sets that consist of linear induction variables into linear integer sequences, mappings need to be introduced between the two domains. For singleton linear induction variables whose loop index bounds are known statically, the mapping $\mathcal{L}'$ from a linear induction variable to a linear integer sequence can be defined as:

$$\mathcal{L}'(\alpha\mathtt{i} + \beta) = \langle\langle \alpha k_1 + \beta, \alpha k_2 + \beta, \alpha k_3\rangle\rangle \text{ for loop index bounds } (\mathtt{i}{=}k_1, k_2, k_3)$$

Additionally, a straightforward modification needs to be performed to ensure that the high bound for the tuple is an element of the integer sequence.

The complication of mapping value sets into integer sequences arises either when the loop bounds are not known or when multiple linear induction variables occur in a value set. In these cases, it is important to keep in mind the question one is asking when performing the comparison. Since the eventual optimizations that use array flow analysis do not require exact answers, an approximation can be used as long as it is inaccurate in the right direction. Consider the case where a response of ''yes'' causes an optimization to be performed and a response of ''no'' results in no transformations to the program. Then answering ''no'' all the time would produce a correct although slow program whereas answering ''yes'' falsely results in a fast but incorrect program. Since the entire goal of this chapter is dependence analysis, the conservative view states that every dependence that can exist should be detected. Even if additional false dependences are detected, the resulting program would still work.

In consideration of the above principles, the conversion of linear induction variables into linear integer sequences requires both an under-approximation and an over-approximation. The two cases arise from different uses of flow elements that contain linear induction variables as indices, and will be discussed in a later section.

In one case, we desire a linear integer sequence that is the smallest computable superset of the integers in a value set. The superset mapping $\mathcal{L}_{\supset}$ from value sets into linear integer sequences is introduced for this case. For each linear induction variable $e \equiv \alpha\mathtt{i} + \beta$ in the value set, the mapping $\mathcal{L}'_{\supset}$ can be defined to yield an integer sequence that is a superset of the integers represented by e. When a linear induction variable contains a loop index whose bounds are not known, then $\mathcal{L}'_{\supset}(e)$ yields $\mathcal{L}'(e)$ on a superset of the loop space of $\mathtt{i}$. For a value set of several linear induction variables $\{e_1, e_2, \ldots, e_n\}$, the superset mapping is defined as:

$$\mathcal{L}_{\supset}(\{e_1, e_2, \ldots\}) = \bigcup_{i=1}^{n} \mathcal{L}'_{\supset}(e_i)$$

Although one would prefer the smallest possible superset from this mapping, a very conservative implementation of $\mathcal{L}_{\supset}$ can always return $\top_{is}$ and still produce a correct result. Indeed, one can view the treatment of arrays in conventional scalar flow analysis as using such an approximation function.

In the other case, we desire a linear integer sequence that is the largest computable subset of the value set. Likewise, for a linear induction variable $e \equiv \alpha\mathtt{i} + \beta$, the mapping $\mathcal{L}'_{\subset}$ can be introduced to yield $\mathcal{L}'(e)$ on a subset of the loop space of $\mathtt{i}$. The subset mapping can then be defined as:

$$\mathcal{L}_{\subset}(\{e_1, e_2, \ldots\}) = \bigcap_{i=1}^{n} \mathcal{L}'_{\subset}(e_i)$$

Again, a very conservative implementation of $\mathcal{L}_{\subset}$ can always return $\perp_{is}$ and still be correct.

The above equations require union and intersection operations on linear integer sequences which can be defined by a set of rules. Again, it is important to note that the union and intersection operations only need to be conservative and not absolutely correct. Thus the union operation listed above can actually return a superset of the actual union while the intersection operation can return a subset of the actual intersection. An ambitious implementation can trade off compiler time for execution of complex rules to increase accuracy of dependence information. A small and by no means exhaustive set of rules is given in Figure 3-8.

$$
\begin{aligned}
\langle\!\langle m_1, m_2, m_3 \rangle\!\rangle \cup \langle\!\langle n_1, n_2, n_3 \rangle\!\rangle &= \langle\!\langle n_1, n_2, n_3 \rangle\!\rangle && \text{if } \langle\!\langle m_1, m_2, m_3 \rangle\!\rangle \subseteq \langle\!\langle n_1, n_2, n_3 \rangle\!\rangle \\
\langle\!\langle m_1, m_2, m_3 \rangle\!\rangle \cup \langle\!\langle n_1, n_2, n_3 \rangle\!\rangle &= \langle\!\langle m_1, m_2, m_3 \rangle\!\rangle && \text{if } \langle\!\langle n_1, n_2, n_3 \rangle\!\rangle \subseteq \langle\!\langle m_1, m_2, m_3 \rangle\!\rangle \\
\langle\!\langle m_1, m_2, m_3 \rangle\!\rangle \cap \langle\!\langle n_1, n_2, n_3 \rangle\!\rangle &= \langle\!\langle n_1, n_2, n_3 \rangle\!\rangle && \text{if } \langle\!\langle n_1, n_2, n_3 \rangle\!\rangle \subseteq \langle\!\langle m_1, m_2, m_3 \rangle\!\rangle \\
\langle\!\langle m_1, m_2, m_3 \rangle\!\rangle \cap \langle\!\langle n_1, n_2, n_3 \rangle\!\rangle &= \langle\!\langle m_1, m_2, m_3 \rangle\!\rangle && \text{if } \langle\!\langle m_1, m_2, m_3 \rangle\!\rangle \subseteq \langle\!\langle n_1, n_2, n_3 \rangle\!\rangle \\
\langle\!\langle m_1, m_2, m_3 \rangle\!\rangle \cup \langle\!\langle n_1, n_2, n_3 \rangle\!\rangle &= \langle\!\langle m_1, n_2, m_3 \rangle\!\rangle && \text{if } m_3 = n_3 \text{ and } m_2 = km_3 + n_1 \text{ for } k \geq 0 \\
\langle\!\langle m_1, m_2, m_3 \rangle\!\rangle \cap \langle\!\langle n_1, n_2, n_3 \rangle\!\rangle &= \langle\!\langle n_1, m_2, m_3 \rangle\!\rangle && \text{if } m_3 = n_3 \text{ and } m_2 = km_3 + n_1 \text{ for } k \geq 0 \\
\langle\!\langle m_1, m_2, m_3 \rangle\!\rangle \cup \langle\!\langle n_1, n_2, n_3 \rangle\!\rangle &= \langle\!\langle \min(m_1, n_1), \max(n_1, n_2), m_3/2 \rangle\!\rangle \\
&\qquad \text{if } m_3 = n_3 \text{ and } |m_1 - n_1| = m_3/2 \text{ and } |m_2 - n_2| = m_3/2
\end{aligned}
$$

Figure 3-8: Rules for combining linear integer sequences

The above definitions imply that the superset mapping preserves order in the lattice while the subset mapping causes an ordering reversal. Recall that a set of linear induction variables e_1 is higher in the lattice than another set e_2 if $e_1 \supset e_2$. The following claim can be made:

Lemma 3.3: If e_1 and e_2 are value sets and $e_1 \supseteq e_2$ then $\mathcal{L}_\supset(e_1) \supseteq \mathcal{L}_\supset(e_2)$ and $\mathcal{L}_\subset(e_1) \subseteq \mathcal{L}_\subset(e_2)$.

Proof: From the set inclusion ordering on value sets, the superset and subset mappings on e_2 can be defined as:

$$\mathcal{L}_\supset(e_1) = \mathcal{L}_\supset(e_2) \cup \mathcal{L}_\supset(e_1 - e_2)$$

$$\mathcal{L}_\subset(e_1) = \mathcal{L}_\subset(e_2) \cap \mathcal{L}_\subset(e_1 - e_2)$$

Clearly, if $e_1 \supseteq e_2$, then $\mathcal{L}_\supset(e_1) \supseteq \mathcal{L}_\supset(e_2)$ and the ordering is preserved for the superset mapping. In addition, $\mathcal{L}_\subset(e_1) \subseteq \mathcal{L}_\subset(e_2)$ and ordering is reversed for the subset mapping. □

For a collection of linear induction variables e, one can view the superset mapping as an upper bound on the integers represented by e. The subset mapping can then be viewed as the set-negation of an upper bound of the integers not in e. In particular, observe the following mappings of $\top$ and $\perp$:

$$\mathcal{L}_\supset(\top_{iv}) = \top_{is} \quad \text{and} \quad \mathcal{L}_\supset(\perp_{iv}) = \perp_{is}$$

$$\mathcal{L}_\subset(\top_{iv}) = \perp_{is} \quad \text{and} \quad \mathcal{L}_\subset(\perp_{iv}) = \top_{is}$$

Intuitively, since $\top_{iv}$ is the collection of all linear index functions, its union produces all integers while its intersection produces the empty set. The definition of mappings on $\perp_{iv}$ exists only for consistency since no executable expression has approximation $\perp_{iv}$ from Corollary 3.2.

3.3.2 Summary of array index approximations

At this point, it may be helpful to summarize the various representations for value set approximations of array indices. At the most specific level, the possible values of an array index can be represented as a collection of integers. If the collection is too large or is not known at compile time, then it can be approximated as $\top_c$, the collection of all integers. For the purpose of dependence analysis, linear induction variables representing linear functions of loop indices form a more efficient and accurate representation than sets of integers. A single integer can be represented as a linear function with a multiplicative factor of 0. The existence of branches in a program implies that an array index can be defined as multiple linear induction variables. Again, the element $\top_{iv}$ can be viewed as the collection of all linear induction variables.

A collection of linear induction variables forms an approximation in that it is a superset of the actual linear loop index functions that correspond to an array index at run time. However, each linear induction variable is exact in the sense that it represents each linear loop index function precisely. Unfortunately, that exactness produces difficulty in comparing and combining linear induction variables. In order to perform operations on value sets more easily, mappings are introduced to convert collections of linear induction variables into linear integer sequences. The lack of full data knowledge at compilation requires that the mappings be inexact. A superset mapping $\mathcal{L}_\supset$ of a collection produces a linear integer sequence that is guaranteed to include every integer that is in the collection. A subset mapping $\mathcal{L}_\subset$ produces a linear integer sequence that is guaranteed to be in every dynamic instantiation of the collection.

3.3.3 Subarrays

In array flow analysis, the basic units to be propagated are either scalar variables or subsets of arrays. Since scalars can be considered zero-dimensional arrays, the basic unit of propagation in array flow analysis can be defined as a *subarray*–a subset of an array. The subarray $\mathcal{S}(\texttt{a[i]})$ of an array access `a[i]` is defined as the elements of array `a` with indices in the value set $\mathcal{A}(\texttt{i})$. For example, $\mathcal{S}(\texttt{a[j]}) = a[\alpha_1\texttt{i} + \beta_1, \alpha_2\texttt{i} + \beta_2]$ if linear induction variable propagation yields an approximation $\mathcal{A}(\texttt{j}) = \alpha_1\texttt{i} + \beta_1, \alpha_2\texttt{i} + \beta_2$. For clarity, the flow algorithms are presented for only scalars and one-dimensional arrays. Multi-dimensional arrays are discussed in a subsequent section.

The subarray is an accurate but unwieldy representation. Each value set that forms a subarray index can be comprised of several linear induction variables whose bounds are not known. In the previous section, a mapping is described to obtain more manageable approximations of value sets. Likewise, we can introduce mappings to form approximations of subarrays. The subset and superset mappings on subarrays can be defined as follows:

$$\mathcal{M}_{\subset}(a[e]) = a[\mathcal{L}_{\subset}(e)] \text{ (elements of } a \text{ with index in } \mathcal{L}_{\subset}(e))$$

$$\mathcal{M}_{\supset}(a[e]) = a[\mathcal{L}_{\supset}(e)] \text{ (elements of } a \text{ with index in } \mathcal{L}_{\supset}(e))$$

Intuitively, one can view the superset mapping $\mathcal{M}_{\supset}$ on a subarray is a portion of the array that is guaranteed to contain all elements in the subarray. Similarly, the subset mapping $\mathcal{M}_{\subset}$ on a subarray is a portion of the array that is guaranteed to be contained in the subarray.

3.3.4 Array flow analysis algorithm

An algorithm is given in detail for calculating previous reaching definitions of each statement. Reaching uses can be computed in a similar manner and are summarized at the end of the section.

For each statement S in a program, we associate four sets of subarrays:

$defGen[S]$	Definitions that are generated by S
$defKill[S]$	Definitions that are removed by S
$defIn[S]$	Definitions that reach the beginning of S
$defOut[S]$	Definitions that are active at the end of S

Computation of the four sets can be done in two passes. The first derives the generation and kill sets ($defGen$ and $defKill$) in a bottom-up manner and the second derives the flow sets ($defIn$ and $defOut$). The algorithm is illustrated in Figure 3-9.

Observe that the gen sets contain over-approximations of subarrays while kill sets contain under-approximations of subarrays. Since the information in gen sets is used for dependence analysis, all actual generated values of a statement must be guaranteed to exist in its gen set. On the other hand, since kill sets exist only to mask out non-reaching definitions, the computed kill set of a statement must be a subset of the actual set of values killed by the statement. In addition, loop index information of subarrays must be preserved in gen sets for use in dependence testing, while no need exists for keeping information on induction variables of each subarray index in kill sets.

$$
\begin{aligned}
\mathit{defGen}[\![V = E]\!] &= \mathcal{M}_{\supset}(\mathcal{S}(V)) \\
\mathit{defGen}[\![\texttt{if (}V\texttt{) } S_1 \texttt{ else } S_2]\!] &= \mathit{defGen}[S_1] \cup \mathit{defGen}[S_2] \\
\mathit{defGen}[\![\texttt{while (}V\texttt{) } S]\!] &= \mathit{defGen}[S] \\
\mathit{defGen}[\![\texttt{do (}I\texttt{=}K_1\texttt{,}K_2\texttt{,}K_3\texttt{) } S]\!] &= \mathit{defGen}[S] \\
\mathit{defGen}[\![\texttt{doall (}I\texttt{=}K_1\texttt{,}K_2\texttt{,}K_3\texttt{) } S]\!] &= \mathit{defGen}[S] \\
\mathit{defGen}[\![\texttt{\{}S_1\ S_2\texttt{\}}]\!] &= (\mathit{defGen}[S_1] \setminus \mathit{defKill}[S_2]) \cup \mathit{defGen}[S_2]
\end{aligned}
$$

$$
\begin{aligned}
\mathit{defKill}[\![V = E]\!] &= \mathcal{M}_{\subset}(\mathcal{S}(V)) \\
\mathit{defKill}[\![\texttt{if (}V\texttt{) } S_1 \texttt{ else } S_2]\!] &= \mathit{defKill}[S_1] \cap \mathit{defKill}[S_2] \\
\mathit{defKill}[\![\texttt{while (}V\texttt{) } S]\!] &= \mathit{defKill}[S] \\
\mathit{defKill}[\![\texttt{do (}I\texttt{=}K_1\texttt{,}K_2\texttt{,}K_3\texttt{) } S]\!] &= \mathit{defKill}[S] \\
\mathit{defKill}[\![\texttt{doall (}I\texttt{=}K_1\texttt{,}K_2\texttt{,}K_3\texttt{) } S]\!] &= \mathit{defKill}[S] \\
\mathit{defKill}[\![\texttt{\{}S_1\ S_2\texttt{\}}]\!] &= (\mathit{defKill}[S_1] \setminus \mathit{defGen}[S_2]) \cup \mathit{defKill}[S_2]
\end{aligned}
$$

Figure 3-9: Computation of definition gen and kill sets

The following claim shows that gen and kill sets are derived correctly in the intuitive sense: For any statement S, the set of definitions that can be generated by S at run time is a subset of the gen set and a superset of the kill set.

Claim 3.4: For a statement S, let $\mathit{defReal}[S]$ be the set of definitions that can be generated by S at run time. Then $\mathit{defKill}[S] \subseteq \mathit{defReal}[S] \subseteq \mathit{defGen}[S]$.

Proof: This can be shown by structural induction on the statement S. From the definitions of $\mathcal{M}_{\subset}$ and $\mathcal{M}_{\supset}$ and Lemma 3.3, the claim is true for assignment statements since the propagation algorithm produces a superset of the linear induction variables that can occur in an array index (Claim 3.1). By induction, the claim can be shown easily when S is a loop or conditional.

For statement S as a sequence $[\texttt{\{}S_1\ S_2\texttt{\}}]$, the set of real definitions of an invocation of S can be defined as $\mathit{defReal}[S] = \mathit{defReal}[S_1] \cup \mathit{defReal}[S_2]$.

To show that $\mathit{defReal}[S] \subseteq \mathit{defGen}[S]$, if definition d is in $\mathit{defReal}[S]$, then either $d \in \mathit{defReal}[S_1]$ or $d \in \mathit{defReal}[S_2]$. If $d \in \mathit{defReal}[S_2]$, then $d \in \mathit{defGen}[S_2]$ by induction and $d \in \mathit{defGen}[S]$. If $d \in \mathit{defReal}[S_1]$ and $d \notin \mathit{defReal}[S_2]$, then by induction $d \in \mathit{defGen}[S_1]$ and $d \notin \mathit{defKill}[S_2]$ which implies $d \in (\mathit{defGen}[S_1] \setminus \mathit{defKill}[S_2])$ and thus $d \in \mathit{defGen}[S]$.

To show that $\mathit{defKill}[S] \subseteq \mathit{defReal}[S]$, if definition d is in $\mathit{defKill}[S]$, then either $d \in \mathit{defKill}[S_2]$ or $d \in (\mathit{defKill}[S_1] \setminus \mathit{defGen}[S_2])$. If $d \in \mathit{defKill}[S_2]$, then $d \in \mathit{defKill}[S]$ by induction. If $d \in (\mathit{defKill}[S_1] \setminus \mathit{defGen}[S_2])$, then $d \in \mathit{defKill}[S_1]$ and $d \notin \mathit{defGen}[S_2]$. Therefore $d \in \mathit{defReal}[S_1]$ and $d \notin \mathit{defReal}[S_2]$ by induction and $d \in \mathit{defReal}[S]$. □

At this point, one may argue that since kill sets are always subsets of gen sets, there is really no need to subtract a kill set and insert a gen set of the same statement. However, it is also important to keep in mind that dependence analysis not only requires knowledge of which variables can reach certain points, but also which statements those variables come from. Implicitly associated with each subarray in a set of definitions is the statement where that subarray was defined. Kill sets thus play an important role in that they mask definitions from particular statements as definitions from new statements are added to the gen set.

$S = [V = E]$	$\mathit{defOut}[S] = (\mathit{defIn}[S] \setminus \mathit{defKill}[S]) \cup \mathit{defGen}[S]$
$S = [\texttt{if (}V\texttt{) } S_1 \texttt{ else } S_2]$	$\mathit{defIn}[S_1] = \mathit{defIn}[S_2] = \mathit{defIn}[S]$
	$\mathit{defOut}[S] = \mathit{defOut}[S_1] \cup \mathit{defOut}[S_2]$
$S = [\texttt{while (}V\texttt{) } S']$	$\mathit{defIn}[S'] = \mathit{defIn}[S] \cup \mathit{defGen}[S']$
	$\mathit{defOut}[S] = \mathit{defGen}[S'] \cup \mathit{defIn}[S]$
$S = [\texttt{do (}I\texttt{=}K_1\texttt{,}K_2\texttt{,}K_3\texttt{) } S']$	$\mathit{defIn}[S'] = \mathit{defIn}[S] \cup \mathit{Dec}(\mathit{defGen}[S'], I, K_3)$
	$\mathit{defOut}[S] = \mathit{markExt}(\mathit{defGen}[S'], I) \cup \mathit{defIn}[S]$
$S = [\texttt{doall (}I\texttt{=}K_1\texttt{,}K_2\texttt{,}K_3\texttt{) } S']$	$\mathit{defIn}[S'] = \mathit{defIn}[S]$
	$\mathit{defOut}[S] = \mathit{markExt}(\mathit{defGen}[S'], I) \cup \mathit{defIn}[S]$
$S = [\texttt{\{}S_1\ S_2\texttt{\}}]$	$\mathit{defIn}[S_1] = \mathit{defIn}[S]$
	$\mathit{defIn}[S_2] = \mathit{defOut}[S_1]$
	$\mathit{defOut}[S] = \mathit{defOut}[S_2]$

Figure 3-10: Computation of definition in and out sets

After gen and kill sets are computed, in and out sets can be derived from the gen sets as in Figure 3-11. The algorithm is generally patterned after conventional scalar flow analysis with several notable differences. To compute the in set for a DO loop, the $\mathit{Dec}(G, I, K)$ function is used to decrement by K any linear induction variables using I that appear in subarrays of the gen set G, where I is the loop index and K is the loop step. When a subarray flows back into the top of a loop from the bottom, the index I

which appears in that subarray is K less than the same index in subarrays of the current iteration. The justification for this operation can be explained by considering the flow dependences in the loop in Figure 3-11. Although statements `S1` and `S3` textually define the same element in `a`, it is statement `S3` that has a flow dependence to `S2`. Statement `S1` has no dependence to `S2` since its definition is killed by `S3`. Using the function Dec, the dependence can be explained by observing that $Dec(a[i], i, 1)$ produces the subarray $a[i-1]$ which then matches the use of `a` in statement `S2`.

```
do (i=1,100,1) {
  a[i] = ...;        /* S1 */
  ... = a[i-1];      /* S2 */
  a[i] = ...;        /* S3 */
}
```

Figure 3-11

The second function introduced is $markExt(G, I)$. For each subarray, we associate an extra external field that specifies whether that subarray has been propagated outside the loop specified by the field. By default, when a subarray is created, its external field is set to null. The function $markExt(G, I)$ sets the external field in all subarrays of G to the loop specified by I. Since this field is used for improving dependence testing, its motivation is presented in the later section on detection of dependences.

The algorithm computes the out set of a loop as the union of the gen set of its body and the in set from its predecessor. Ideally, we would prefer to be able to subtract the kill set of the body from the in set of the predecessor. However, the computation must account for the case when the loop body is not executed at all. If the loop can be assured to be executed at least once such as the case of `DO` loops with known bounds, then the out set can be defined as $defOut[S] = markExt(defGen[S'], I) \cup (defIn[S] \setminus defKill[S'])$.

Correctness of the computation of in and out sets is fairly immediate from correctness of gen and kill sets and can be shown from works on data flow analysis of scalars. Note that since in and out sets are derived from gen sets, they are supersets of the actual definitions that enter and exit a statement. This is consistent with our conservative aim to detect a superset of all dependences that can arise in a program.

The use sets can be defined in the same manner as def sets. Since definitions kill previous uses as well as previous definitions, the kill set calculation of uses is exactly the same as that of definitions. Likewise, the propagation of in and out is identical. The only difference appears in the computation of gen sets, where variable uses instead of variable definitions are merged into the gen sets.

3.3.5 Flow analysis on multi-dimensional arrays

At first glance, one can imagine the above analysis extending to multi-dimensional arrays in a straightforward manner. Since an n-dimensional array index can be represented as an n-tuple of integers, its value set consists of n-tuples of linear induction variables. Such value sets can then be approximated as grids in n-dimensional space rather than just linear integer sequences. At each level, approximations can be done on individual indices in each dimension separately.

```
float a[100,100];
doall (i=1,100)
  doall (j=1,100)
    a[i,j] = ...;          /* S1 */
doall (i=1,100)
  a[i,i] = ...;            /* S2 */
```

Figure 3-12

Unfortunately, the above specification produces incorrect results for cases where indices in different dimensions are related to each other. Consider the program in Figure 3-12. If approximations are derived on indices in each dimension separately, then each array index in the program can be approximated as the linear integer sequence from 1 to 100. The kill set for statement `S2` is the entire array `a`, and the definition of statement `S2` kills the definition of statement `S1`. However, this is not correct since the definition in statement `S2` actually only kills 100 elements on the diagonal of array `a`.

Several schemes can be used to address this problem. The first and easiest involves arguing that such array reference patterns are rare and consequently only require an inefficient solution. If two array indices can ever contain linear functions of the same loop index in their respective value sets, then the subset mapping $\mathcal{L}_{\subset}$ on each array index returns $\perp_{is}$. Therefore, if an array access is approximated as a grid in n-dimensional space, then none of its indices have been approximated to $\perp_{is}$ and each index is independent of the other. This is the approach used in the implementation of this thesis. In the above example, the kill set of statement `S2` would be the empty set since both indices would be approximated as $\perp_{is}$. The second solution involves expanding the representation of array index approximations to include shapes other than rectangular grids in n-dimensional space. In the above example, the kill set of statement `S2` would be the diagonal of array `a`. Although such an approach requires additional compiler complexity over the first, it is not clear how much additional benefit it provides for dependence analysis.

3.3.6 Related work

The algorithms given here are adapted from well-known flow analysis algorithms for scalars [ASU86][MJ81]. Although the basic high-level structure of the algorithms are the same, major differences do exist between the comparison of flow units which form the basic mechanism for managing flow information.

The topic of array flow analysis has not been explored until very recently, when it has suddenly become somewhat popular. Initial efforts at array flow analysis focused on the goal of detecting loop-based parallelism. Gross and Steenkiste [GS90] and Granston and Veidenbaum [GV91] rely on the structure of scalar flow analysis, but use array regions as flow elements. Unfortunately, as shown in the above examples, a more effective representation is needed to compute accurate dependence information. Rau [Rau91] and Duesterwald, et al [DGS93] use the linear induction variables themselves as indices of flow elements. However, such an exact representation causes set operations on flow elements to become unwieldy or almost impossible.

3.4 Detection of dependences

As stated previously, dependences can be computed from interactions between definitions and uses in the current statement and definitions and uses in previous statements. A dependence arises when a subarray from a previous statement intersects with a subarray in the current statement. Determining whether two subarrays intersect requires tests on array subscripts that indicate whether the two subscripts can ever have the same value.

In every dependence between two statements, one statement is the *source* that performs some action and the other statement is the *sink* that must wait until that action is completed. For example, in a flow dependence, the source statement writes some value to some memory location and the sink statement can only read that memory location after waiting for the writer to finish.

In order to specify how dependences are derived from array flow analysis results, several other subarray sets need to be defined. Let $Def[S]$ be the set of definitions and $Use[S]$ be the set of uses in statement S. Then dependences can be computed for each statement S as follows:

$$\text{Flow dependences of } S = \{(d, d') : d \in \mathit{defIn}[S] \text{ and } d' \in \mathit{Use}[S] \text{ and } d \; \delta \; d'\}$$
$$\text{Anti-dependences of } S = \{(d, d') : d \in \mathit{useIn}[S] \text{ and } d' \in \mathit{Def}[S] \text{ and } d \; \delta \; d'\}$$
$$\text{Output dependences of } S = \{(d, d') : d \in \mathit{defIn}[S] \text{ and } d' \in \mathit{Def}[S] \text{ and } d \; \delta \; d'\}$$

The above definitions require the computation of dependences (δ) between subarrays, which is defined below. Let subarray $a[e_1]$ correspond to the source statement and $a[e_2]$ correspond to the sink. If a dependence arises ($a[e_1] \; \delta \; a[e_2]$), then it is possible during program execution for the sink statement to access some memory location after the source accesses it. Since it is assumed that there is no aliasing of variables, the two subarrays must refer to the same array for there to be a dependence. Dependence testing of subarrays thus can be accomplished by testing whether dependences exist between linear induction variables. For now, we consider dependence testing on one-dimensional arrays.

To determine dependences between linear induction variables, we can apply well-known tests for detecting array dependences in nested `DO` loops [Wol89][Ban88]. Although these tests are normally used to recognize whether different iterations of a `DO` loop can be executed in parallel, the same dependence-testing mechanism can be used to detect dependences for synchronization. In general, dependence testing can be reduced to determining whether two array references can ever represent the same array element at the same time. For two array references $a[f_1(i_1)]$ and $a[f_2(i_2)]$ to intersect, there must be some point in the program where $f_1(i_1) = f_2(i_2)$. Let $f_1(i_1) = \alpha_1 i_1 + \beta_1$ and $f_2(i_2) = \alpha_2 i_2 + \beta_2$. For each index variable i_j, let the loop statement corresponding to it be `do` $(i_j{=}l_j, h_j, s_j)$. Clearly, if the index ranges of the two references do not overlap, then no dependences can exist. However, overlapping ranges do necessarily imply dependence. One needs to study the linear index functions themselves in order to determine whether two array indices can possess the same value at the same time. Although more complex and effective dependence tests exist, the GCD test shall be used here for simplicity. If there is an intersection and a solution exists for the equality $f_1(i_1) = f_2(i_2)$, then from linear diophantine equation theory, the following must be true:

$$\gcd(\alpha_1 s_1, \alpha_2 s_2) \ \text{ divides } \ \alpha_1 l_1 - \alpha_2 l_2 + \beta_1 - \beta_2$$

In adapting the GCD test to subarrays, first consider the case where e_1 and e_2 are each approximated by only one linear induction variable, so that $e_1 = \{\alpha_1 i_1 + \beta_1\}$ and $e_2 = \{\alpha_2 i_2 + \beta_2\}$. Typically, the GCD test can be used to determine whether the linear induction variables intersect. However, when the two loop indices are equal ($i_1 = i_2$) and is the index of a sequential `DO` loop, then another condition needs to be true for a dependence to exist. Consider the test for flow dependence between statements `S1` and `S2` in Figure 3-13. Although the GCD test returns true in this case (1 divides -1 for the index `j`), no flow dependence actually exists since each element should be read one iteration of `j` before the write.

```
do (j=1,100) {
  doall (k=1,100) a[j-1,k-1] = ...;   /* S1 */
  doall (k=1,100) ... = a[j,k];       /* S2 */
}
```

Figure 3-13

In the case of `DO` loops, a simple GCD test for intersection of linear induction variables can produce many false dependences. The test must take into account the sequential nature of `DO` loop indices. Once again, although more complex and effective tests exists, we present a more straightforward test for simplicity: For a source subarray $a_1[e_1]$ and sink subarray $a_2[e_2]$ with $e_1 = \{\alpha_1 i + \beta_1\}$ and $e_2 = \{\alpha_2 i + \beta_2\}$ where i is the index of a `DO` loop, then $a_1[e_1] \; \delta \; a_2[e_2]$ if the GCD test is true and

$$\alpha_1 \neq \alpha_2 \ \text{ or } \ \alpha_1\beta_1 \geq \alpha_2\beta_2$$

In the above example, $\alpha_1 = \alpha_2$ and $\beta_1 < \beta_2$, so no flow dependence exists.

Unfortunately, the above condition is not sufficient for activating this more accurate test. Consider the program in Figure 3-14, an augmented version of Figure 3-13. Even though a flow dependence does not exist from `S1` to `S2` through the `j` loop, a flow dependence does exist through the `i` loop. At the end of the `j` loop, the entire array `a` has been written by statement `S1`. Thus any reads done by `S2` in the next iteration of `i` cannot be done before the write in the previous iteration of `i`. There are actually two entries in the *defIn* set that reaches `S2`, one from the definition of `a` in the current `i` iteration and one from the previous iteration. In a sense, the `j` index from the previous iteration is different from the one in the current iteration. The *markExt* function introduced earlier can be used to mark the fact that the subarray propagated from the previous iteration of `i` is external to the loop `j`. In the terminology of [AK87], this field is equivalent to specifying that the dependence is *loop-independent* rather than *loop-carried* with respect to an outer loop. The application of this external field also corresponds to the different cases of dependence checking for different *data direction vectors* of [BC86] and [Wol89]. In summary, the above test can be done only if sequential loop index variables are identical and the source subarray is external with respect to the relevant loop.

```
do (i=1,10)
  do (j=1,100) {
    doall (k=1,100) a[j-1,k-1] = ...;  /* S1 */
    doall (k=1,100) ... = a[j,k];      /* S2 */
  }
```

Figure 3-14

In order to detect dependences between two linear induction variables with identical `DOALL` loop indices, an even simpler test can be used. If the source subarray is not external to the `DOALL` loop, then a dependence can only arise if the linear functions can ever produce the same result for a particular value of the loop index. We use the following simple test: For a source subarray $a_1[e_1]$ and sink subarray $a_2[e_2]$ with $e_1 = \{\alpha_1 i + \beta_1\}$ and $e_2 = \{\alpha_2 i + \beta_2\}$ and i as the index of a `DOALL` loop, a dependence exists if the GCD test is true and

$$\alpha_1 \neq \alpha_2 \quad \text{or} \quad \beta_1 = \beta_2$$

Observe that when a source subarray $a[e]$ is external with respect to a loop, then any loop indices in e are in effect different from loop indices in the sink subarray. In summary, the following table can be used to specify tests for different scenarios of source and sink loop induction variables. We assume once again that source and sink subarrays are $a_1[e_1]$ and $a_2[e_2]$ with $e_1 = \{\alpha_1 i_1 + \beta_1\}$ and $e_2 = \{\alpha_2 i_2 + \beta_2\}$. Then $a_1[e_1]\ \delta\ a_2[e_2]$ under the following condition:

<table>
<tr><td rowspan="2"></td><td rowspan="2">$i_1 \neq i_2$</td><td colspan="2">$i = i_1 = i_2$</td></tr>
<tr><td>i = <code>DO</code> index</td><td>i = <code>DOALL</code> index</td></tr>
<tr><td>$a_1[e_1]$ not extern of i</td><td rowspan="2">GCD</td><td>($\alpha_1 \neq \alpha_2$ or $\alpha_1\beta_1 \geq \alpha_2\beta_2$)
and GCD</td><td>($\alpha_1 \neq \alpha_2$ or $\beta_1 = \beta_2$)
and GCD</td></tr>
<tr><td>$a_1[e_1]$ extern of i</td><td>GCD</td><td>GCD</td></tr>
</table>

In general, the value set that is a subarray index consists of several linear induction variables. A dependence test must be done for every pair of linear induction variables of two subarray indices. Since any of the linear induction variables can be the actual array index, the final result is true if any of the pairwise tests were true. In addition, any dependence test involving array indices with approximation $\top_{iv}$ always returns true.

For subarrays of multiple dimensions, two approaches can be used. The first involves doing dependence testing dimension-by-dimension and deducing a dependence only if every dimension deduces a dependence. The second involves linearization of the array reference by using known array bounds to map the multiple-dimensional index space into a one-dimensional space. Unfortunately, each approach has cases in which the other approach produces a more accurate answer [Wol89]. For the best solution, both approaches can be used to test each dependence. However, the implementation in this thesis only performs dimension-by-dimension testing.

3.4.1 Invariant expressions

In addition to linear functions of loop indices, one can also imagine propagating and performing dependence analysis on linear functions of invariant variables as well. Consider the example in Figure 3-15. Although nothing is known about the value of the variable `x`, we do know that it is invariant in the context of the two statements. Thus the values `x` and `x+1` cannot possibly be equal, and no flow dependence exists between the two statements. One can thus perform dependence analysis on linear functions of general variables as well by applying the same test as the case of linear functions with identical DOALL indices. In the literature [AK87][PW86], propagation and dependence analysis are specified with respect to these general linear functions rather than only loop induction variables.

```
doall (i=1,100) a[x,i] = ...;
doall (i=1,100) ... = a[x+1,i];
```

Figure 3-15

Note that the restriction that the unknown variable be invariant is very important. In our example, if an assignment to x appears between the two statements, then the same dependence test cannot be applied. Likewise, if the two statements appear in a sequential loop and x is modified anywhere in the loop, then one must also use a different test for dependences across different iterations of the outer loop. As a rule, for a source subarray that is external with respect to a loop L, the above test can be applied only when the unknown variable is invariant in the body of L.

3.5 Interprocedural support

The analysis presented thus far has only focused on performing flow analysis within a procedure. When fully general user-defined functions are allowed, provisions must be made to analyze the flow of data into and out of procedure calls. This section presents a practical but by no means thorough discussion of the interprocedural array flow analysis approach used in the implementation of this thesis.

Constants and linear induction variables can be propagated across procedures by allowing the algorithm to propagate functions of integer procedure parameters as well as loop indices. In the function of Figure 3-16a, although the algorithm knows nothing about x, it can speculate and assume that x is a loop induction variable. The value of y can then be determined to be x + 1. The question of whether x is a loop induction variable is not resolved until one applies flow analysis.

The algorithm which computes reaching information assumes that interprocedural dependences are not detected at the level of the procedure being called, but instead at the level of the caller. Thus any statement that invokes the procedure f above must ensure that any dependences to f are supported before the call and any dependences

```
void f(int x)
{
  y = x+1;
  doall (i=1,100)
    a[i,y] = ...;
}
```

(a)

```
f(5);           /* S1 */
do (j=10,100)
  f(j);         /* S2 */
f(z);           /* S3 */
```

(b)

Figure 3-16

from f are supported immediately after the call. This allows the compiler to generate only one version of the function f instead of potentially creating a different copy of the procedure for each call. Of course, any dependences occurring within f would be supported inside its body. With this assumption, correct reaching information can be derived by separately computing the gen sets for each call to f. For instance, the call to f in statement S1 of Figure 3-16b produces the gen set containing the subarray a[i,6], while the call in statement S2 produces the subarray a[i,j+1]. In both cases, the results are derived from resolving the gen set produced by the body f, which contains a[i,x+1], with the arguments passed to f. For the call in statement S3, if nothing is known about the value of z, then the subarray returned is a[i,⊤]

Note that much potentially derivable information is ignored by the above scheme. In particular, the lack of specialization of procedure calls requires one to be overly pessimistic when generating code for the procedure. For example, if the procedure makes use of two integer parameters, analysis within the procedure must assume that their values are unknown and that any dependence tests involving them return true. One can easily imagine scenarios where some calls to such a procedure are made with arguments that cause the dependences to not exist. For those cases, it can be beneficial to make two versions of the procedure, one that supports the dependence and one that does not. At the call site, analysis can be done to determine which procedure should be invoked.

3.6 Other applications of array flow analysis

Array flow analysis provides information on data usage relationships between statements. In addition to using this information for deriving point-to-point synchronization, other optimizations for parallel programs can benefit from results of flow analysis. Such optimizations include parallelism detection, private variable detection, data and loop partitioning, and static data routing. Other optimizations that can benefit from array flow analysis are shown in [DGS93].

3.6.1 Parallelism detection

In compiling programs for multiprocessors, a very useful optimization involves the detection of parallelism in sequential `DO` loops [AK87][Wol89]. When a statement in a `DO` loop body does not depend on other statements in the body, then it can be vectorized by being moved out of the loop and placed in a `DOALL` loop. Array flow analysis provides more accurate information for determining whether a loop can be vectorized.

```
do (i=1,100) {
  a[i-1] = f1(b[i]);          /* S1 */
  c[i] = f2(a[i-1]);          /* S2 */
  a[i] = f3(c[i]);            /* S3 */
  d[i] = f4(a[i-2]);          /* S4 */
}
```

Figure 3-17

Without array flow analysis, dependence testing is done on all definitions and uses in loop, thus possibly producing some false dependences. Consider dependence testing on all definitions and uses of the loop in Figure 3-17. We must conclude that the loop cannot be vectorized since there seems to be a cyclic dependence involving statements `S2` and `S3` generating the equation

```
c[i] = f2(f3(c[i-1]))
```

However, using array flow analysis, we can deduce that the definition of `a` in `S1` actually kills the definition in `S3`. Thus there is no cyclic dependence, and each statement can be vectorized.

3.6.2 Private variable detection

When detecting parallelism from sequential `DO` loops, certain transformations can be performed on a loop to make it more easily parallelized. One of these transformations is the introduction of private variables to remove output and anti-dependences across loop iterations. A variable is *private* in some loop if every iteration of the loop can be viewed as possessing a private copy of that variable. Consider the programs for exchanging two arrays in Figure 3-18. In both cases, if the variable `temp` is privatized, then all iterations can be executed in parallel. The topic of privatizing arrays has only recently been discussed in the literature [EHLP91][MAL93].

```
do (i=1,100)
  do (j=1,100) {
    temp = a[i,j];
    a[i,j] = b[i,j];
    b[i,j] = temp;
  }
```

(a)

```
do (i=1,100)
  do (j=1,100) {
    temp[j] = a[i,j];
    a[i,j] = b[i,j];
    b[i,j] = temp[j];
  }
```

(b)

Figure 3-18

A variable v is a candidate for privatization within some loop l when certain conditions can be satisfied. First, any flow dependences involving v must only occur within single iterations of l. If one were to allow for copying, then flow dependences can also occur to statements outside of the loop, but never across iterations of a loop. Second, v must appear in some output and anti-dependences across loop iterations, otherwise there is no need for it to be privatized. While scalar flow analysis can verify these conditions for scalars, array flow analysis allows verification for arrays as well. In Figure 3-18, the variable `temp` can be privatized with respect to both loops in case (a) and can be privatized with respect to loop `i` in case (b).

3.6.3 Data and loop partitioning

Even when all potential parallelism is detected in a program, its performance can still be heavily affected by communication costs. In many cases, effective static allocation of tasks and data to processors can reduce these costs significantly. Data partitioning involves splitting and aligning data to minimize communication distance between processors and the data they access [KLS90][LC91][GB92][RS91]. In loop partitioning, nested loops can be mapped to processors to minimize non-local memory accesses [AH91]. In these techniques, constraints between arrays are formed from flow dependences and occurrences in common statements. A partitioning algorithm then performs heuristics to resolve cyclic constraints and produce a partitioning scheme. Using array flow analysis, more accurate flow dependences can be computed to produce improved partitioning results.

3.6.4 Static routing of data

In most multiprocessors, interprocessor communication is accomplished by sending messages through a network. In the conventional scheme of dynamic routing, a message is routed by examining its header which identifies the destination processor for the message. In situations where two messages need to access the same resource, one message must be either blocked or buffered. Architectures such as iWarp [Bor90] or NuMesh [War93] seek to alleviate contention costs by introducing the idea of static routing. When destinations of messages are known at compilation, then routing can be scheduled statically to avoid unnecessary contentions [SA91]. Furthermore, hardware which supports static routing can avoid the latency associated with examining headers as in dynamic routing.

In loop-based parallel programs, communication between processors arises primarily from flow dependences between different processors. When these flow dependences involve arrays whose indices are constants or linear functions of loop indices, then static routing can be applied. In the program of Figure 3-17, let us assume a machine topology of 100 processors in a line where each processor is responsible for one loop iteration. Since statement `S4` requires a read of `a[i-2]` and statement `S3` writes `a[i]`, each processor must send its result from `S3` two processors to the right. Since the communication destination for each processor is known at compile time, static routing can be applied. Compilation for systolic arrays is a particular approach towards static routing and has been heavily studied [Kun82][Che86][Cap87]. These works focus on the optimal execution of a set of nested `DO` loops without dynamic control flow such as conditionals. Since static routing allows very high network bandwidths to be available, one solution allows conditionals to be supported by performing all communication that can exist on any path through the program. Some of the ideas used computing processor dependences in the next chapter can be used to support a scheme for static routing in general programs.

3.7 Summary

In order to provide intelligent support for synchronization, one must first be able to detect dependences between statements in a program. Although one can define dependences by searching the program text for any accesses that can overlap, more effective results can be obtained by performing flow analysis to detect the reaching span of each data access. In order to manage references to array elements effectively, we focus on array indices that are linear functions of loop indices. The task of deducing this information can be performed by an adaptation of known value propagation algorithms to the linear function lattice.

Because some data accesses can completely mask others, array flow analysis can be used to determine the region over which each array access is active. Rather than operating on array regions, the flow analysis done here preserves the index function and traversal path of the flow element to allow for accurate dependence testing. Dependences can then be computed between reaching accesses and current accesses for each statement. Since dependence testing has been thoroughly studied in the literature, this thesis proposes only using the simple GCD test to detect dependences. The result of this analysis yields dependence information between statements as well as array accesses that generate those dependences.

Chapter 4

Processor dependences and synchronization

4.1 Introduction

In the previous chapter, it was shown how dependences between statements can be derived from array flow analysis. Given a program, the algorithms of the previous chapter provide dependence relationships between pairs of statements along with the array accesses that cause those dependences. When a dependence exists between two statements, synchronization must be inserted to maintain the proper execution order. However, producing point-to-point synchronization requires additional analysis to derive the pairwise synchronization relationships between processors. In this chapter, we present a scheme for computing and implementing point-to-point synchronization for general array-based programs. First, some examples are discussed for motivation, followed by an overview of the problem of deriving processor dependences. Then the concept of statement instances is introduced along with preliminary computation of relationships between instances. An execution model is then presented, followed by techniques for computing processor synchronization relationships and avoiding deadlock scenarios. Finally, we address efficiency and present an algorithm for computing point-to-point synchronization statically.

4.2 Motivation

When implementing point-to-point synchronization for a data dependence between two statements involving a processor p, two questions arise:

1. What are the processors p' with which p needs to synchronize?

2. Which dynamic activities of p and p' should be synchronized with each other?

Question 2 arises out of the fact that data dependences really exist in the realm of dynamic program execution. Although it may be clear where dependences exist in the text of a program, these lexical locations may actually be executed many times. Thus provisions must be made for recognizing which dynamic invocations of the lexical locations are actually dependent on each other. Question 1 can be viewed as the spatial relationship while question 2 can be viewed as the temporal relationship between the source and sink statements of a data dependence.

The transformation from a program with barrier synchronization semantics to one with point-to-point synchronization must satisfy several conditions. First, it must produce a program that is still correct. Any dependence that exists in a program with barrier semantics must be satisfied in the transformed program by the insertion of a synchronization. Second, the resulting program must terminate in all cases where the original program terminates. In other words, no deadlocks can be introduced by the transformation. In a sense, the first criterion requires that enough synchronizations are produced, while the second requires that not too many synchronizations are produced.

4.2.1 Synchronization model

The synchronization scheme presented in this chapter assumes a shared-memory execution model. To invoke a point-to-point synchronization, the *source* processor writes a value to some memory location and the *sink* processor spin-locks until the memory location reaches that particular value. Only one memory location is needed to support multiple synchronizations involving the same processors if synchronization values are restricted to be monotonically increasing. The sink processor then spin-locks until the memory location contains a value greater than or equal to the desired value.

On cache-coherent shared-memory machines, the costs of reading and writing to memory are influenced by the cache coherence scheme. A write involves a cache access as well as possible invalidations of matching cache entries in other processors. A read corresponds minimally to a cache read. However, if the address is not found in the cache, then a memory or network access is needed. Spin-locking on a read does not necessarily incur a large amount of network traffic since each read request can typically be serviced by a cache read. When a write is performed by the source processor, the cache entry in the sink processor is eventually invalidated. The subsequent read then causes a new cache value to be loaded from the source processor. By using memory accesses to support synchronization, we require that the cache protocol be sequentially consistent. In other words, the order of any two accesses by the same processor must be preserved by the memory hierarchy.

```
do (i=1,100) {
  do (k=1,10) {
    doall (j=1,1000) {
      a[i,j] = ...;           /* S1 */
      sync[j] = i;            /* S3 */
    }
  }
  do (l=1,5) {
    doall (j=1,1000) {
      while (sync[j-1] < i); /* S4 */
      ... = f(a[i-3,j-1]);   /* S2 */
    }
  }
}
```

Figure 4-1

The program in Figure 4-1 shows how the above scheme can be used to support the flow dependence between `S1` and `S2` without resorting to barrier synchronizations. After a value is written to an element of `a`, the synchronization variable for that array element is also updated in statement `S3`. Before the same element `a` is read, statement `S4` ensures that its synchronization variable has the proper value. Although this approach produces a correct program, it suffers from various inefficiencies which are addressed in the next section.

4.2.2 Implementation issues

Consider the execution of the program in Figure 4-1 on a machine with 10 processors. Assume that each of the `DOALL` loops is distributed 100 consecutive iterations per processor so that processor $p_{\mathtt{j}}$ is responsible for iterations $100\mathtt{j} - 99$ to $100\mathtt{j}$. As implemented in the example, each processor needs to check 100 values of the `sync` array before it proceeds. Instead, the partitioning information can be used to observe that each processor $p_{\mathtt{j}}$ needs data written by iterations $100\mathtt{j} - 100$ to $100\mathtt{j} - 1$ of the `DOALL` loop for `S1`. Therefore, processor $p_{\mathtt{j}}$ only needs to synchronize with processor $p_{\mathtt{j}-1}$ and synchronization can be accomplished by checking one value rather than 100. One can view the difference of `j` indices of the array accesses as inducing a *spatial* relationship on the statements. Although the above case seems straightforward, this problem can be more complex when loop partitions are not uniform and multiple array dimensions involve multiple `DOALL` loops.

```
do (i=1,100) {
  do (k=1,10) {
    doall (j=1,1000) {
      a[i,j] = ...;          /* S1 */
    }
  }
  sync[p] = i;
  while (sync[p-1] < i-3);
  do (l=1,5) {
    doall (j=1,1000) {
      ... = f(a[i-3,j-1]);   /* S2 */
    }
  }
}
```

Figure 4-2

In addition to a spatial relationship between statements, a temporal relationship exists as well. In the above example of Figure 4-1, while it is certainly correct to synchronize with the current iteration of `i`, each definition of array `a` is not actually used until 3 iterations of `i` later. Consequently, rather than checking the `sync` variable for the current index of `i`, one can check for the value `i-3` and allow for more variance in execution among the processors. The iteration distance can be viewed as a *temporal* relationship between the statements. In addition, observe that synchronization is unnecessarily performed for every iteration of the inner `DO` loops. Since the dependence really exists from the last iteration of the `k` loop to the first iteration of the `l` loop, synchronization calls can be moved outside of the loops. The improved program is shown in Figure 4-2 with the variable `p` representing the current processor number.

Another example can be used to illustrate the difference between temporal and spatial relationships. In Figure 4-3, an anti-dependence exists between the use of the variable `x` and its definition. Here, we follow an assumption that iterations of sequential loops are not partitioned among different processors. In case (a), synchronization only needs to be done with the last iteration of the sequential loop, while in case (b), synchronization must be done with every iteration of the parallel loop. This can be explained by the observation that sequential loop iterations are ordered in time while parallel loop iterations are not.

The simple examples described above become much more complex upon examination of Figure 4-4a. The first index of the array `a` corresponds to a sequential loop in `S1`

```
do (i=1,100)
  ... = x;
...
x = ...;
```

(a)

```
doall (i=1,100)
  ... = x;
...
x = ...;
```

(b)

Figure 4-3

but corresponds to a parallel loop in `S2`. Conversely, the second array index is a parallel loop index in `S1` but is a sequential one in `S2`. In addition, the last index corresponds to an unknown variable `t` in `S1` and a sequential loop in `S2` and it is not clear whether the parallel loop index `k` influences the value of `t`. If `t` can be shown to be invariant with respect to certain loops, then provisions can perhaps be made to treat `t` as a constant for those loops. Although synchronizing one array element at a time can still be made to work in this case, it is not clear which of the above improvements can be incorporated. In particular, note that since the second index of array `a` in `S2` is a sequential loop index, moving the synchronization checking out of that loop implies that all 100 indices of `j` in `S1` must still be checked. In Figure 4-4b, the complexity comes from the fact that loop indices `i` and `k` are used multiply in array indices while index `j` is not used at all. As evident in these examples, spatial and temporal relationships are not necessarily straightforward and must be defined more clearly.

```
do (i=1,100)
  doall (j=1,100)
    doall (k=1,20) {
      ...
      a[i,j,t] = ...;    /* S1 */
    }
doall (i=1,100)
  do (k=1,85)
    do (j=1,100)
      ... = f(a[i,j,k]); /* S2 */
```

(a)

```
do (i=1,100) {
  do (j=1,100) {
    doall (k=1,100)
      a[i,i,k+1] = ...;
    ...
    doall (k=1,100)
      ... = a[i-3,k,k-1];
  }
}
```

(b)

Figure 4-4

The above examples illustrate the fact that general and efficient implementation of point-to-point synchronization cannot be accomplished through an ad-hoc method. Rather, a formal treatment of synchronization relationships as well as a parallel-loop execution model must be introduced to provide the proper background for considering algorithms which compute synchronization targets.

Throughout this chapter, synchronization relationships are computed with respect to the sink of the dependence rather than the source. As shown in the above examples, each synchronization is implemented using an array of values with one element for each processor. At the source, each processor *asserts* a synchronization by setting its own array element to some value to indicate that it has reached the source statement. At the sink processor, the synchronization *check* requires the computation of array elements to check and values to use for the check. These computations correspond to the two questions posed earlier in this chapter.

4.2.3 Termination issues

Implementing point-to-point synchronization involves transforming a program so that its parallel execution is accurate without always relying on barrier synchronizations. If point-to-point synchronizations are added wherever dependences exist, then the program is guaranteed to produce results that are correct. However, correctness is not the only important criterion. The transformations also must not introduce any deadlock conditions into the program. Unfortunately, a straightforward implementation of synchronization insertion can easily introduce deadlock when conditionals are present. Consider the program in Figure 4-5. If any iteration of `i` results in a `false` value for `f(i)`, then deadlock occurs since no synchronization variables are set to `i` and all processors would wait forever at statement `S4`.

```
do (i=1,100) {
  if (f(i)) {
    doall (j=1,1000)
      a[i,j] = ...;          /* S1 */
    sync[p] = i;             /* S3 */
    ...
  }
  while (sync[p-1] < i);     /* S4 */
  doall (j=1,1000)
    ... = f(a[i,j-1]);       /* S2 */
}
```

Figure 4-5

The above example can be rectified by either moving the synchronization assertion out of the conditional or inserting another assertion in the `else` clause of the conditional. For this case, synchronization by individual array elements can be prohibitively expensive to support since every possible array element that is accessed in the body of a conditional requires an update to the respective synchronization location. On the other hand, synchronization by processor only requires updates to locations of processors that may have executed the conditional. Although the given solution is fairly convincing for the above example, its correctness as a general solution is not readily apparent for cases involving more complex control flow. A more formal model of execution and synchronization must be introduced to allow for construction of a scheme that is provably deadlock-free.

4.3 Overview of processor dependences

The problem of deriving synchronization relationships requires detailed analysis of array references in order to compute dependences between processors. Given a dependence between two lexical statements, many dependences can actually arise between the different run-time invocations of the two statements. Figure 4-6 illustrates an example with arrows representing such dependences. For two invocations, a dependence exists between them if their array indices evaluate to the same value. In cases where dependences cannot be completely determined, one can over-approximate towards having too many dependences. Thus arrows can be drawn between two invocations when there is a chance that their array indices can represent the same value.

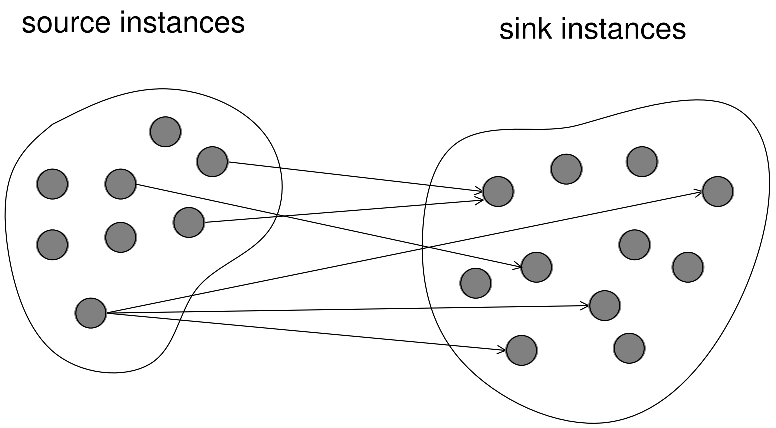
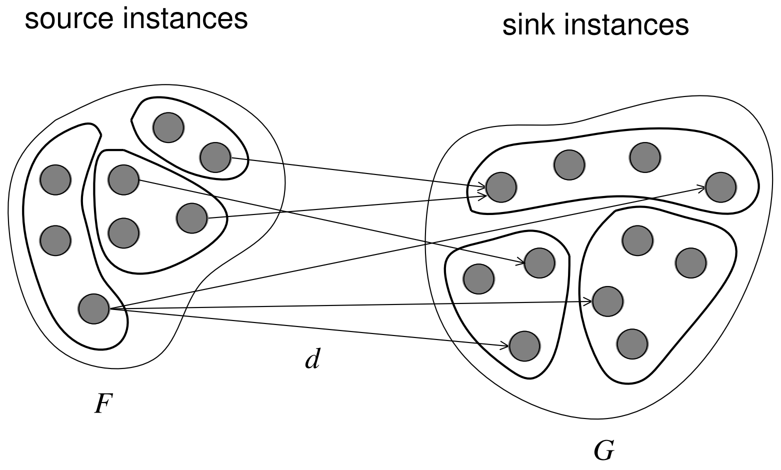


Figure 4-6: Dependences between invocations

Once dependences between invocations of statements are determined, one can use the information to derive dependences between processors. Since the domain of invocations for a statement can be represented by its enclosing loop indices, the results of loop partitioning can be used to provide partitioning functions from statement invocations to processors. As shown in Figure 4-7, a mapping from sink processors to source processors can be obtained by applying the partitioning functions and the dependence relationship between invocations.



Processor p synchronizes with $F(d^{-1}(G^{-1}(p)))$

Figure 4-7: Dependences between processors

Within each processor, its statement invocations are executed in a particular order according to the language semantics. Furthermore, the barrier semantics of `DOALL` loops define an ordering on invocations across processors. If two invocations are separated by a barrier, then one must be executed before the other even if they are partitioned to different processors. This ordering of execution must be obeyed when computing synchronization relationships by ensuring that no synchronization is done for dependences that traverse forward in the execution order. In other words, a sink statement invocation cannot synchronize with a source invocation that is executed after it according to barrier semantics. Thus for each source processor, the sink processor must synchronize only with the source invocations that are executed before the relevant sink invocations.

The above discussion sketches a strategy for computing synchronization relationships between processors. The following few sections focus on a more detailed derivation of this strategy.

4.4 Dynamic instances of statements

In order to more precisely specify relationships between different invocations of statements, we introduce the notion of a statement instance. Each dynamic invocation of a statement is called an *instance* of the statement and is determined by values of loop indices that enclose the statement. Components of an instance that correspond to sequential loops are executed in a particular order and can be considered *temporal* coordinates. Those that correspond to parallel loops are partitioned to processors and can be viewed as *spatial* coordinates.

An instance $S\vec{\omega} = S\langle\omega_1 \ldots, \omega_n\rangle$ of a statement S is defined as an n-tuple where n is the number of loops that enclose S. Specifically, these correspond to parallel `DOALL` loops as well as sequential `DO` and `WHILE` loops. The i-th integer in the n-tuple corresponds to an iteration value of the i-th outermost loop. In the case of `WHILE` loops, we insert a counter to the loop header which can be used for the iteration value.† Although provisions can be made for loop increments that are negative, we assume here that loop indices increase monotonically. In the program of Figure 4-2, valid statement instances are `S1`$\langle 1,1,1\rangle$, `S1`$\langle 100,10,1000\rangle$, `S2`$\langle 1,1,1\rangle$, and `S2`$\langle 100,5,1000\rangle$.

In some cases, it is useful to be able to specify an ordering on when statement instances are executed in a program. Since sequential loop iterations are ordered, an ordering can be defined in terms of the value of sequential loop indices in the instance. For two instances of a statement, the respective temporal coordinates can be compared integer-by-integer from the leftmost position. A *timestamp* can be defined as a tuple that is derived from the temporal coordinates of a statement instance with the function $Tem(\langle\omega_1, \ldots, \omega_n\rangle) = \langle\omega_{i_1}, \ldots, \omega_{i_{n'}}\rangle$ where each i_j-th outermost loop is sequential and n' is the number of sequential loops that enclose the statement. Tuple comparison can be defined as follows:

$$\langle k_1, \ldots, k_m, \ldots, k_n\rangle < \langle l_1, \ldots, l_m, \ldots, l_n\rangle \iff (\forall\, i < m \;\; k_i = l_i) \text{ and } k_m < l_m$$

This definition corresponds to comparing sequential loop index values from the outermost loop inward and can be viewed as an ordering on the sequential *loop iteration space*

† In the unlikely event that the counter overflows, point-to-point synchronization can be abandoned for barrier synchronization in the iteration where the counter is reset.

of a statement. Generally, for two instances of statements S_1 and S_2, comparison must only be done on tuple values that correspond to common loops of the two statements. We introduce the notation $\langle k_1, \ldots, k_n \rangle \uparrow c$ to indicate the subtuple corresponding to the first c elements of a tuple. A *temporal ordering* on statement instances can be defined formally as follows:

$$S_1\vec{\omega}^1 \prec S_2\vec{\omega}^2 \iff \vec{\tau}^1 \uparrow c < \vec{\tau}^2 \uparrow c \ \text{ or}$$
$$\vec{\tau}^1 \uparrow c = \vec{\tau}^2 \uparrow c \text{ and } S_1 \text{ precedes } S_2$$
$$\text{where } \vec{\tau}^1 = Tem(\vec{\omega}^1) \text{ and } \vec{\tau}^2 = Tem(\vec{\omega}^2)$$
$$\text{and } c \text{ is the number of sequential loops that enclose both } S_1 \text{ and } S_2$$

From Chapter 2, a partial ordering is a relation that is anti-symmetric, anti-reflexive, and transitive. The following lemma shows that the temporal ordering relation on statement instances is a true partial ordering:

Lemma 4.1: The relation $S_1\vec{\omega}^1 \prec S_2\vec{\omega}^2$ is a partial ordering.

Proof: Clearly, the relation $\prec$ is anti-reflexive and anti-symmetric since the precedence relationship is anti-reflexive and anti-symmetric. Although it seems intuitively that transitivity is also obvious, its proof is complicated by the fact that statements can appear at different loop nestings. To show transitivity, let $S_1\vec{\omega}^1$, $S_2\vec{\omega}^2$, and $S_3\vec{\omega}^3$ be instances such that $S_1\vec{\omega}^1 \prec S_2\vec{\omega}^2$ and $S_2\vec{\omega}^2 \prec S_3\vec{\omega}^3$. We need to show that $S_1\vec{\omega}^1 \prec S_3\vec{\omega}^3$. Let $c_{i,j}$ be the number of common sequential loops that enclose statements S_i and S_j. Let $L_{i,j}$ be the innermost sequential loop that encloses S_i and S_j. Let $\vec{\tau}^i = Tem(\vec{\omega}^i)$. Suppose by contradiction that $S_1\vec{\omega}^1 \not\prec S_3\vec{\omega}^3$. There are two cases:

For the first case where S_2 is not a descendant of $L_{1,3}$, then $c_{1,2} = c_{2,3} < c_{1,3}$. Further, either S_1 does not precede S_2 or S_2 does not precede S_3. Without loss of generality, assume that S_1 does not precede S_2. If $\vec{\tau}^1 \uparrow c_{1,2} = \vec{\tau}^2 \uparrow c_{1,2}$, then $S_1\vec{\omega}^1 \not\prec S_2\vec{\omega}^2$ and a contradiction arises. Therefore $\vec{\tau}^1 \uparrow c_{1,2} < \vec{\tau}^2 \uparrow c_{1,2}$. Since $\vec{\tau}^2 \uparrow c_{2,3} \leq \vec{\tau}^3 \uparrow c_{2,3}$, $\vec{\tau}^1 \uparrow c_{1,2} < \vec{\tau}^3 \uparrow c_{1,2}$. Thus $\vec{\tau}^1 \uparrow c_{1,3} < \vec{\tau}^3 \uparrow c_{1,3}$ and $S_1\vec{\omega}^1 \prec S_3\vec{\omega}^3$.

For the second case where S_2 is a descendant of $L_{1,3}$, then there are three subcases: (a) $c_{1,2} > c_{2,3}$ (S_3 is outside of $L_{1,2}$), (b) $c_{2,3} > c_{1,2}$ (S_1 is outside of $L_{2,3}$), and (c) $c_{1,2} = c_{2,3}$ (all 3 statements have the same number of common loops). The cases (a) and (b) are similar and the proof is given only for (a): We have $c_{1,2} > c_{2,3} = c_{1,3}$. We know that if S_2 precedes S_3, then S_1 precedes S_3 and $S_1\vec{\omega}^1 \prec S_3\vec{\omega}^3$ since $\vec{\tau}^1 \uparrow c_{1,3} \leq \vec{\tau}^3 \uparrow c_{1,3}$. If S_2 does not precede S_3, then $\vec{\tau}^2 \uparrow c_{2,3} < \vec{\tau}^3 \uparrow c_{2,3}$ which implies that $\vec{\tau}^2 \uparrow c_{1,3} < \vec{\tau}^3 \uparrow c_{1,3}$. Since $\vec{\tau}^1 \uparrow c_{1,3} \leq \vec{\tau}^2 \uparrow c_{1,3}$, we have $\vec{\tau}^1 \uparrow c_{1,3} < \vec{\tau}^3 \uparrow c_{1,3}$. For case (c), let $c = c_{1,2} = c_{2,3} = c_{1,3}$. If either

$\vec{\tau}^1{\uparrow}c < \vec{\tau}^2{\uparrow}c$ or $\vec{\tau}^2{\uparrow}c < \vec{\tau}^3{\uparrow}c$, then $\vec{\tau}^1{\uparrow}c < \vec{\tau}^3{\uparrow}c$. Otherwise, S_1 precedes S_2 and S_2 precedes S_3 which implies that S_1 precedes S_3 and $\vec{\tau}^1{\uparrow}c = \vec{\tau}^3{\uparrow}c$. □

In Figure 4-2, the following temporal ordering exists between statements `S1` and `S2`:

$$\texttt{S1}\langle i_1, k_1, j_1\rangle \prec \texttt{S2}\langle i_2, l_2, j_2\rangle \iff i_1 \leq i_2$$

Note that the temporal ordering does not exactly correspond to the ordering imposed on the instances by the execution semantics. Such an ordering will be defined later. Instead, the temporal ordering specifies in some sense an execution order that is stricter than that defined by the semantics. This is the actual ordering that is obeyed by the synchronization scheme that will be introduced, and can be used to prove that the resulting programs are deadlock-free.

Observe also that the temporal ordering relation does not depend on the spatial coordinates of the instances. Thus if $S_1\vec{\omega}^1 \prec S_2\vec{\omega}^2$, then $S_1\vec{\omega}^3 \prec S_2\vec{\omega}^4$ if $Tem(\vec{\omega}^1) = Tem(\vec{\omega}^3)$ and $Tem(\vec{\omega}^2) = Tem(\vec{\omega}^4)$. Consequently, it makes sense to abbreviate temporal ordering relations to just the timestamps that represent temporal coordinates of statements: $S_1\,Tem(\vec{\omega}^1) \prec S_2\,Tem(\vec{\omega}^2)$.

4.5 Deriving synchronization relationships

In general, a dependence involves two array accesses, one at a source statement S_1 and one at a sink statement S_2. By studying the array indices together with the lexical contexts of the statements, one can derive synchronization relationships for the dependence. This information can in turn be used to implement efficient point-to-point synchronization to ensure that the dependence is obeyed at run time.

4.5.1 The problem

The synchronization model presented here assumes that synchronization is performed through a processor writing a value to a memory location at the source which is then checked by another processor at the sink. In order to implement this mechanism, we need to find the dependence relationship between instances of the source and sink statements for a given dependence. We first focus on the simpler problem of deriving the set of source instances that have a dependence with a particular sink instance. Formally, for each instance $S_2\vec{\omega}^2$ of a sink statement S_2, we wish to derive the set of instances

$S_1\vec{\omega}^1$ of the source S_1 that need to be executed before executing $S_2\vec{\omega}^2$. In other words, we wish to find the set of instances of the source such that a dependence $S_1\vec{\omega}^1 \; \delta \; S_2\vec{\omega}^2$ exists between those instances and the sink instance.

As in the previous chapter, the optimizations here require analysis that can be conservative. The task of finding an exact answer for the above problem is certainly undecidable when a program has conditional statements. Therefore we must approximate towards having too many synchronizations rather than having too few to assure correct execution. Furthermore, approximations also allow many synchronization targets to be computed statically to reduce execution costs. Although additional information is available at run time to allow more accurate computation of synchronization targets, the cost of such computation can overshadow any potential benefits. Consequently, the primary goal here involves deriving synchronization targets statically as much as possible to minimize run-time overhead.

4.5.2 Orthogonal derivation of instance relationships

The problem of deriving the source instances for a given sink instance relies on the following inputs: The sink instance $S_2\vec{\omega}^2$, the sink array reference $a[\vec{e}^{\,2}]$, and the source array reference $a[\vec{e}^{\,1}]$. Each array reference $\vec{e}$ is a collection of expressions e_i each of which can be approximated by a value set $\mathcal{A}(e_i)$ as shown in the previous chapter. For simplicity, we assume that each value set can contain only one linear induction variable. Multiple linear induction variables in a value set need to be analyzed one at a time with the final result being the union of each single analysis. We use the notation $e|S\vec{\omega}$ to indicate the value of the expression e at a particular statement instance. For particular source and sink instances, a dependence exists between them if the array reference values at the respective instances are equal. For an instance $S_i\vec{\omega}^i$, we use the notation ω^i_j to represent the coordinates of $\vec{\omega}^i$.

Rather than deriving the set of complete instances for a source statement, the problem can be simplified by deriving each instance coordinate separately. The cartesian product of computations of the problem on individual instance coordinates represents a superset of the set of desired instances, as shown in the following lemma:

<u>Lemma 4.2</u>: Let $S_2\vec{\omega}^2$ be a particular sink instance and $\Omega' = \prod_j \Omega_j$

where $\Omega_j = \{\omega^1_j : \; \exists \; \omega'_1, \ldots, \omega'_{n_1} \;\; S_1\langle \omega'_1, \ldots, \omega^1_j, \ldots, \omega'_{n_1}\rangle \; \delta \; S_2\vec{\omega}^2\}$

then $\Omega' \supseteq \Omega$ where $\Omega = \{\vec{\omega}^1 : S_1\vec{\omega}^1 \ \delta \ S_2\vec{\omega}^2\}$.
Proof: We can show the superset relation by showing that $\vec{\omega} \in \Omega \Rightarrow \vec{\omega} \in \Omega'$. From above, if $\vec{\omega} \in \Omega$ then $S_1\vec{\omega} \ \delta \ S_2\vec{\omega}^2$. Let $\vec{\omega} = \langle \omega_1^1, \ldots, \omega_{n_1}^1 \rangle$. Then each $\omega_j^1 \in \Omega_j$, and $\vec{\omega} \in \Omega'$. □

The above lemma allows the computation of source instances for a given sink instance to be divided into the smaller problem of computing individual coordinates of source instances separately. However, this separation comes at a price in that any correlations between coordinates are discarded. The set of computed source instances can thus contain some instances that are not involved in any dependences with the sink instance.

4.5.3 Instance relationships

The set of source instances can be derived one coordinate at a time by analyzing the array references of the source and sink statements. Let the expanded sink instance be $S_2\langle \omega_1^2, \ldots, \omega_{n_2}^2 \rangle$. Recall that a dependence exists between the source and sink instances if the evaluation of array references at those instances are equal. In other words, a dependence exists if the following holds:

$$\vec{e}^{\,1}|S_1\vec{\omega}^1 = \vec{e}^{\,2}|S_2\vec{\omega}^2$$

To derive individual instance coordinates, a dependence exists for a coordinate value ω_j^1 under the following condition:

$$\omega_j^1 \in \Omega_j \iff \vec{e}^{\,1}|S_1\langle *, \ldots, \omega_j^1, \ldots, * \rangle = \vec{e}^{\,2}|S_2\langle \omega_1^2, \ldots, \omega_{n_2}^2 \rangle$$

The notation $*$ is used to represent any possible value at a particular location and can be viewed as the variable of an existential quantifier.

For particular array references $\vec{e}^{\,1} = (e_1^1, \ldots, e_n^1)$ and $\vec{e}^{\,2} = (e_1^2, \ldots, e_n^2)$, the above orthogonality principle can be applied across each array index to derive the following condition:

$$\omega_j^1 \in \Omega_j \iff \forall\, i \ \ 1 \le i \le n \ \ e_i^1|S_1\langle *, \ldots, \omega_j^1, \ldots, * \rangle = e_i^2|S_2\langle \omega_1^2, \ldots, \omega_{n_2}^2 \rangle$$

For a particular instance coordinate ω_j^k, let L_j^k be the loop associated with that coordinate and let I_j^k be the loop index of L_j^k. The above relation can be specialized to the following cases:

$$\omega_j^1 \in \Omega_j \iff \forall\, i \;\; 1 \le i \le n \;\; \begin{cases} e_i^1 = f_1(I_j^1),\ e_i^2 = f_2(I_k^2) \text{ and } \omega_j^1 = f_1^{-1}(f_2(\omega_k^2)) \text{ for some } k \\ e_i^1 = f_1(I_j^1) \text{ and } \omega_j^1 = f_1^{-1}(e_i^2 | S_2\vec{\omega}^2) \\ e_i^1 | S_1\langle *, \ldots, \omega_j^1, \ldots, * \rangle = e_i^2 | S_2\vec{\omega}^2 \end{cases}$$

where each f is a linear function of a loop index. In cases where an expression is known to be a linear function of a loop index, specific computations can be applied to derive synchronization information.

The set of source instances $\{S_1\vec{\omega}^1 : \vec{\omega}^1 \in \Omega'\}$ that are dependent on a sink instance $S_2\vec{\omega}^2$ can thus be derived as follows:

$$\Omega' = \prod_j \Omega_j$$

$$\text{where } \Omega_j = Dom(\omega_j^1) \cap \bigcap \sigma_j^i \text{ and}$$

$$\sigma_j^i = \begin{cases} f_1^{-1}(f_2(\omega_k^2)) & \text{if } e_i^1 = f_1(I_j^1) \text{ and } \exists\, k \;\; e_i^2 = f_2(I_k^2) \\ f_1^{-1}(e_i^2 | S_2\vec{\omega}^2) & \text{if } e_i^1 = f_1(I_j^1) \text{ and } e_i^2 \text{ is not a linear induction var.} \\ \emptyset & \text{if } e_i^1 = C \text{ constant and } e_i^2 = f_2(I_k^2) \text{ and } C \ne f_2(\omega_k^2) \\ Dom(\omega_j^1) & \text{if } e_i^1 \text{ is not a linear induction var.} \end{cases}$$

where $Dom(\omega_j^1)$ represents the domain of each source instance coordinate. The above derivation implies that the source instance coordinates are determined completely by source array index expressions that are linear induction variables. The sets σ_j^i can be viewed as filters on the domain of each source instance coordinate. When a source array index is a linear function of a loop index, then the corresponding sink array index is examined to determine the range of the filter, as shown in Figure 4-8. If nothing is known about the source array index, then no filtering is done.

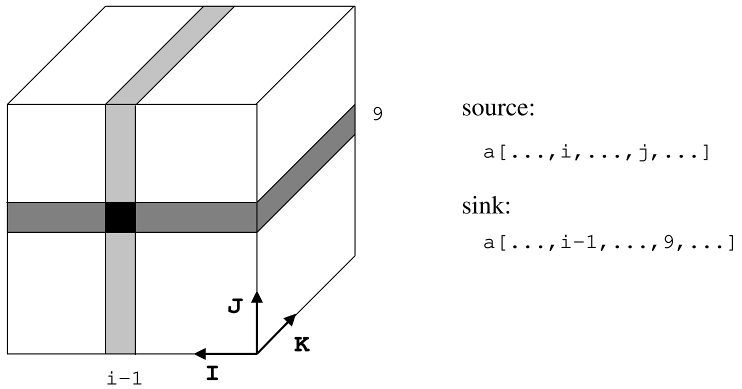


Figure 4-8: Filtering on a 3-dimensional source instance space

For a particular sink instance $S_2\vec{\omega}^2$, the above equations show how to derive the set of source instances that have a dependence with $S_2\vec{\omega}^2$. If run-time efficiency were not a concern, then synchronization can be supported by using an array of size equal to the source instance space, initialized to 0. After each source statement instance is executed, the corresponding element in the array can be set to 1. Before executing each sink statement instance, the set of source elements can be computed by applying the above equations, and synchronization is performed by ensuring that each element in that set is equal to 1.

Unfortunately, realistic memory requirements dictate that we conserve space by maintaining a synchronization array whose size is proportional to the number of processors. One must then consider the mapping from the instance spaces into the smaller space of processors. Since spatial coordinates are partitioned into processors, they are implicitly represented by the processor space. However, one must also account for temporal coordinates, which do not correspond to processors. Fortunately, timestamps that represent temporal coordinates are ordered in that the execution of a particular instance on a processor implies that any instance for that processor with lower timestamps have been executed. Thus they can be represented by only retaining the highest timestamp that have been executed on each processor. Rather than storing a boolean value in the synchronization array, the elements instead contain a tuple representing the greatest timestamp that have been executed. This value along with the source processor completely represents the source instances that are required to perform synchronization. The problem now becomes one of computing the source processors as well as the tuple values that are used by the sink to perform synchronization checking. In order to delve much further into this question, a formal execution model of parallel loops and processor partitioning must be introduced.

4.5.4 Related work

Analysis to compute dependence relationships between instances have been introduced with the goal of privatizing arrays to improve parallelization. Feautrier [Fea91] uses a method which computes constraints on the set of source instances to form a bounded polyhedron. Finding the maximum coordinate in the polyhedron can then be viewed as a parametric integer programming problem. Unfortunately, this general approach produces algorithms that are not efficient enough to be used in practice due to its exponential order of growth. The approach used here can be viewed as solving the problem posed by Feautrier, but for the particular case where all constraint surfaces are orthogonal to axes in the iteration space. Recently, Maydan, et al [MAL93] have introduced a new scheme which solves problems that are almost as general as those of Feautrier, but promises to be more efficient. Although their approach incurs more overhead than the specialized solution presented here, it can be adapted to more general problems, in particular when array indices can be functions of more than one loop index.

4.6 Execution model of parallel loops

Let us first consider the execution model for cases where there are no multiply-nested `DOALL` loops. For each `DOALL` loop, every processor can be assigned a subset of the loop iteration space. Formally, let L represent the loop iteration space and P represent the set of all processors. We can associate with each `DOALL` loop a *loop partitioning function* $\phi : L \rightarrow P$ which maps loop index values into processors. Although loop iterations can in theory be partitioned into processors in many ways, we focus on the case where each processor is responsible for a contiguous block of loop iterations. A loop partitioning also designates a sequential processor p_{seq} in P which is responsible for the execution of sequential code outside of `DOALL` loops. The execution semantics can be defined for a statement S on a processor p as follows:

1. Before execution of any statement, synchronize with all other processors.
2. If S is an assignment, then execute if $p = p_{seq}$.
3. If S is a `DOALL` loop with index variable `i` and partitioning function ϕ, then for every value in $\phi^{-1}(p)$, execute the body with `i` bound to that value.
4. If S is a sequential loop or conditional or sequence, then execute S and execute its body according to the rules.

```
a = b;                  /* S1 */
if (a) {
  z = 5;                /* S2 */
  a = x;                /* S3 */
  doall (i=1,128,1)
    c[i] = d[i];        /* S4 */
}
```

Figure 4-9

In the program of Figure 4-9, the scalar assignments `S1`, `S2`, and `S3` appear outside the `DOALL` statement and are executed by only one processor. Therefore only processor p_{seq} executes `S1`, all processors execute the conditional, only processor p_{seq} executes `S2` and `S3`, and each processor executes its portion of the `DOALL` loop. Synchronization must be done before execution of the conditional to prevent other processors from reading the value of `a` before it is written by p_{seq} in `S1`. Likewise, all processors are synchronized before executing the body of the conditional to prevent the value from being written too early in `S3`.

In alternate execution models, only one processor executes predicates of conditionals and `WHILE` loops. Other processors must then check the result of that test to decide whether to execute the body of the conditional. Such a dependence between the sequential processors and all other processors is called a *control dependence*. Instead of following such semantics, the current model allows all processors to execute the predicates. Since there can be no side-effects in the predicates, any extra assignments needed to compute the predicate are done by a sequential processor before the conditional. Therefore control dependences between the sequential processor and other processors are translated into flow dependences between the same processors.

When `DOALL` loops can be nested, then the processor space must be divided into multiple dimensions. As an example, consider the case where each of 64 processors is assigned a 6-bit address. If there are 2 nested `DOALL` loops, then the processor space must be divided into 2 dimensions. One partitioning scheme views the first 3 bits of the processor address as the address in the first dimension and the last 3 bits as the address in the second dimension. An equally valid scheme involves using all 6 bits of address as the first-dimension address and no bits in the second-dimension address.

```
doall (i=1,128,1) {
  b[i] = ...;           /* S1 */
  doall (j=1,64,1)
    a[i,j] = ...;
}
```

Figure 4-10

The motivation for partitioning the processor space can be illustrated by considering Figure 4-10. Once again suppose that there are 64 processors with 6-bit addresses. There exists many options in mapping the loop iteration space into the processor space. In one case, each processor can be responsible for 2 iterations of the outer loop and all 64 iterations of the inner loop. In another case, each processor is responsible for all 128 outer iterations and 1 inner iteration. Many other alternatives exist between the two extremes. Consider the case where the first 3 bits of the processor address is used for the first-dimension address and the second 3 bits of the processor address is used for the second-dimension address. Each loop can then be divided into 8 equal sections, each corresponding to a dimension coordinate. Each processor is then responsible for 16 outer iterations and 8 inner iterations.

Formally, one can view the separation of the processor addresses into dimensions as the partitioning of the processor space for each dimension. Given a dimension, two processors are in the same partition if they belong to the same coordinate in that dimension. We define a *partitioning set* K of a set S as a set of non-empty subsets of S such that each value of S appears in exactly one element, or *partition*, of K. For each loop in a set of nested `DOALL` loops, we can associate with it a *processor partitioning function* $\psi : P \rightarrow \mathrm{K}$ such that $\psi(p) = \kappa$ for $p \in \kappa$. In considering the previous example, the following are valid partitioning functions for two nested `DOALL` loops where the symbol "&" represents the "logical AND" operation:

$$\psi_1(p) = address(p) \ \& \ 111000$$

$$\psi_2(p) = address(p) \ \& \ 000111$$

In the case of a single `DOALL` loop with no nesting, the most obvious processor partitioning function maps each processor into the singleton set containing itself. From the above definition, the processor partitioning function is onto since partitioning sets can only contain non-empty subsets of P. This fact becomes important when loop iterations are mapped into partitions because processors must exist to do the work of each partition.

For a set of n nested loops, there exists partitioning functions $\{\psi_1, \ldots, \psi_n\}$ that divide the processor space for partitioning sets $\{\mathrm{K}_1, \ldots, \mathrm{K}_n\}$. A *composite partitioning function* Ψ can be defined as the product of the partitioning function of each loop as follows:

$$\Psi(p) = \psi_1(p) \cap \ldots \cap \psi_n(p)$$

The function Ψ maps P into a *composite partitioning set* $\mathbf{K}$ where $\mathbf{K} = \mathrm{K}_1 \Cap \ldots \Cap \mathrm{K}_n$ and $\Cap$ is defined as:

$$A \Cap B = \{a \cap b : a \in A \text{ and } b \in B\}$$

If the partitioning function Ψ is valid, then the composite partitioning set $\mathbf{K}$ can be viewed as any other partitioning set. Each composite partition is thus required to be non-empty since Ψ is itself onto. Therefore for any representative selection of partitions $(\kappa_1, \ldots, \kappa_n) \in \mathrm{K}_1 \circledcap \ldots \circledcap \mathrm{K}_n$, there must exist a processor p such that $\psi_1(p) = \kappa_1, \ldots, \psi_n(p) = \kappa_n$ or equivalently, $\forall\ i \quad p \in \kappa_i$. At the outermost level, all processors are in the same partition, while at the innermost loop level, each processor typically belongs to its own partition.

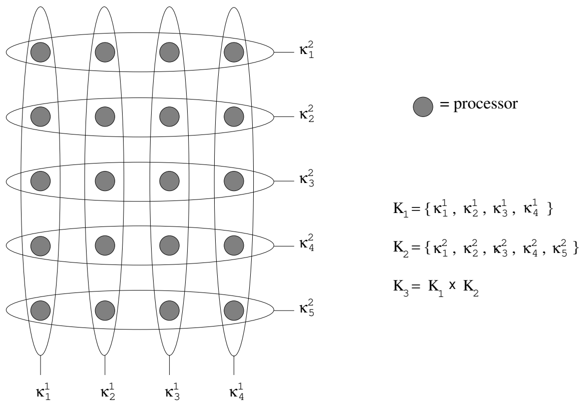


Figure 4-11: Composite processor partitioning

Composite partitioning functions can be viewed as a division of the processor space into an n-dimensional grid as shown in Figure 4-11. The requirement that each partition be non-empty implies that each grid point must contain at least one processor. As an example, the following can be shown to be an invalid partitioning for two nested loops:

$$\psi_1(p) = address(p)\ \&\ 111000$$

$$\psi_2(p) = address(p)\ \&\ 001111$$

By selecting κ_1 as processor addresses matching the pattern `001XXX` and κ_2 as processors matching the pattern `XX0000`, there exists no processor that belongs to both partitions since they require non-matching third-bit values.

To complete the specification of parallel loop execution, the loop partitioning function ϕ is modified to map the loop index space into processor partitions. Associated with each loop is a processor partitioning set K, a processor partitioning function $\psi : P \to \mathrm{K}$, and a loop index space partitioning function $\phi : L \to \mathrm{K}$. For any statement with n outer loops, let $\psi_1, \ldots, \psi_n$ be the processor partitioning functions and $\phi_1, \ldots, \phi_n$ be the loop partitioning functions of its outer loops. Its relevant processor and loop partitioning functions can be computed as:

$$\Psi(p) = \psi_1(p) \cap \ldots \cap \psi_n(p)$$
$$\Phi(\langle \omega_1, \ldots, \omega_n \rangle) = \phi_1(\omega_1) \cap \ldots \cap \phi_n(\omega_n)$$

Note that the loop partitioning functions maps the set of statement instances into a composite processor partition. There is also a representative processor p^{κ}_{seq} of each composite processor partition κ which is responsible for invoking sequential code inside the loops. Execution for processor p proceeds can be defined within the context of an active partition κ. The initial partition κ includes all processors. The execution rules from above can be modified as follows:

1. Before execution of any statement, synchronize with all other processors in partition κ.

2. If S is an assignment, then execute if $p = p^{\kappa}_{seq}$.

3. If S is a loop with index variable `i`, loop partitioning function ϕ, and processor partitioning function ψ, then for every value in $\phi^{-1}(\psi(p))$, execute the body with `i` bound to that value and the new partition $\kappa' = \kappa \cap \psi(p)$.

4. If S is a conditional or sequence, then execute S and execute its body according to the rules.

In this thesis, we restrict the partition set of sequential loops to contain only one element:

$$\forall\, p \;\; \psi_i(p) = P \qquad \text{if the } i\text{-th loop is sequential}$$

Consequently, every iteration of a sequential loop is executed on the same processor partition. By making this assumption, the partition functions for sequential loops can be ignored, and composite partitioning functions can be viewed as being defined entirely by `DOALL` loop partitioning functions. Thus spatial coordinates of the instance space are encapsulated by the partitioning functions ϕ and ψ. Temporal coordinates correspond to sequential loop indices and are captured by the *Tem* function. As we will see, this division has significant implications towards how source processors and timestamps are computed.

Let us once again consider the program in Figure 4-11 with the following processor partitioning:

$$\psi_1(p) = address(p) \ \& \ 111000$$

$$\psi_2(p) = address(p) \ \& \ 000111$$

Processors are partitioned in the outer loop according to ψ_1 and in the inner loop according to ψ_2. For each partition of ψ_1, the statement S1 should be invoked by only one processor. That sequential processor for each partition of ψ_1 can be the one whose address matches the pattern XXX000. Thus the program can be converted as in Figure 4-12 for each processor. Note that since there is only one processor for each composite partition of the two loops, the value of seq2 is always true.

```
partition1 = processor_number & 0b111000;
partition2 = processor_number & 0b000111;
lo1 = 16 * partition1 + 1;
lo2 = 8 * partition2 + 1;
seq1 = (processor_number & 0b000111) == 0;
seq2 = (processor_number & 0b000111 & 0b111000) == 0;

do (i=lo1,lo1+15,1) {
  if (seq1)
    b[i] = ...;           /* S1 */
  do (j=lo2,lo2+7,1)
    if (seq2)
      a[i,j] = ...;
}
```

Figure 4-12

4.7 Execution order of statement instances

With the execution model of parallel loops specified, an execution ordering can be defined on statement instances. An instance is less than another if it must be executed before the other according to the execution model. The *execution ordering* on statement instances can be defined as follows:

$$S_1\vec{\omega}^1 < S_2\vec{\omega}^2 \iff \exists\, c' \ \ \Phi_{c'}(\vec{\omega}^1{\uparrow}c') = \Phi_{c'}(\vec{\omega}^2{\uparrow}c') \text{ and } Tem(\vec{\omega}^1{\uparrow}c') < Tem(\vec{\omega}^2{\uparrow}c') \ \text{ or}$$
$$\Phi_c(\vec{\omega}^1{\uparrow}c) = \Phi_c(\vec{\omega}^2{\uparrow}c) \text{ and } Tem(\vec{\omega}^1{\uparrow}c) = Tem(\vec{\omega}^2{\uparrow}c) \text{ and } S_1 \text{ precedes } S_2$$

where $1 \le c' \le c$ and c is the number of DOALL loops that enclose S_1 and S_2

The functions Φ_i represent the composite processor partitioning function at the i-th outermost loop. Intuitively, the above definition allows comparison of temporal tuple coordinates until processors belong to different composite partitions. At the extreme when

$\vec{\omega}^1{\uparrow}c$ and $\vec{\omega}^2{\uparrow}c$ are mapped to the same processors, then comparisons can be made on all common temporal tuple coordinates.

Similar to the temporal ordering, the execution ordering also satisfies the anti-symmetry, anti-reflexivity, and transitivity properties:

Lemma 4.3: The execution ordering relation $S_1\vec{\omega}^1 < S_2\vec{\omega}^2$ is a partial ordering.
Proof: The proof can be adapted from that of Lemma 4.1 and is omitted. □

The execution ordering of statement instances can be used to infer the order in which instances must be executed according to the rules described by the loop execution semantics. Formally, we can define the notion that an instance A is *executed before* another instance B when one or more barriers are invoked between their executions involving the respective processors. If A is less than B in the execution ordering, then the semantics of the execution model guarantee that A is executed before B. The following shows that the execution ordering on instances implies semantic execution order.

Lemma 4.4: For two statements S_1 and S_2 with instances $\vec{\omega}^1$ and $\vec{\omega}^2$, if $S_1\vec{\omega}^1 < S_2\vec{\omega}^2$ then $S_1\vec{\omega}^1$ is executed before $S_2\vec{\omega}^2$.
Proof: In the following proof, let L_j be the j-th outermost loop. There are two cases that can satisfy $S_1\vec{\omega}^1 < S_2\vec{\omega}^2$:

(a) $\exists\, c' \;\; \Phi_{c'}(\vec{\omega}^1{\uparrow}c') = \Phi_{c'}(\vec{\omega}^2{\uparrow}c')$ and $Tem(\vec{\omega}^1{\uparrow}c') < Tem(\vec{\omega}^2{\uparrow}c')$
(b) $\Phi_c(\vec{\omega}^1{\uparrow}c) = \Phi_c(\vec{\omega}^2{\uparrow}c)$ and $Tem(\vec{\omega}^1{\uparrow}c) = Tem(\vec{\omega}^2{\uparrow}c)$ and S_1 precedes S_2.

For case (a), if $Tem(\vec{\omega}^1{\uparrow}c') < Tem(\vec{\omega}^2{\uparrow}c')$, then there exists $j \leq c'$ such that $\omega^1_j < \omega^2_j$, L_j is a sequential loop index, and $\forall\; i < j \;\; \omega^1_i = \omega^2_i$ if L_i is a sequential loop. This implies that the index values of the outermost sequential loops are equal up to loop L_j. If $j = 1$, then all processors are synchronized between different iterations of L_j and the execution order holds. Otherwise, let Φ_{j-1} be the processor partitioning function at loop L_{j-1}. Since $j \leq c'$, $\Phi_{c'}(\vec{\omega}^1{\uparrow}c') = \Phi_{c'}(\vec{\omega}^2{\uparrow}c') \Rightarrow \Phi_{j-1}(\vec{\omega}^1{\uparrow}j-1) = \Phi_{j-1}(\vec{\omega}^2{\uparrow}j-1)$ and $\Phi_{c'}(\vec{\omega}^1{\uparrow}c') \subseteq \Phi_{j-1}(\vec{\omega}^1{\uparrow}j-1)$ by the definition of processor partitioning functions. Thus all processors in $\Phi_{j-1}(\vec{\omega}^1{\uparrow}j-1)$ are synchronized between iterations of L_j and the execution order holds.

In case (b), for all $i < c$, $\omega^1_i = \omega^2_i$. Let S_3 be the innermost sequence that is a common ancestor of S_1 and S_2. Let S'_1 be the child of S_3 that is an ancestor of S_1 and S'_2 be the child of S_3 that is an ancestor of S_2. Then the c-th loop is the innermost loop that encloses S_3. Let $\kappa = \Phi_c(\vec{\omega}^1{\uparrow}c)$. Since $\Phi_c(\vec{\omega}^1{\uparrow}c) = \Phi_c(\vec{\omega}^2{\uparrow}c)$, all processors in κ are synchronized between executions of S_1' and S_2' and S_1 is executed before S_2 within the same temporal instance and partition. □

In addition, we can show that if instance A is executed before instance B, then $A < B$:

Lemma 4.5: If $S_1\vec{\omega}^1$ is executed before $S_2\vec{\omega}^2$, then $S_1\vec{\omega}^1 < S_2\vec{\omega}^2$.
Proof: This proof relies on many of the same mechanisms as the proof of the previous lemma. Hence only an intuitive sketch is given. By contradiction, assume that $S_1\vec{\omega}^1$ is executed before $S_2\vec{\omega}^2$, but $S_1\vec{\omega}^1 \not< S_2\vec{\omega}^2$. Then we know that both the following are true:

$$\forall\, c' \leq c \;\; \Phi_{c'}(\vec{\omega}^1{\uparrow}c') \neq \Phi_{c'}(\vec{\omega}^2{\uparrow}c') \text{ or } \mathit{Tem}(\vec{\omega}^1{\uparrow}c') \not< \mathit{Tem}(\vec{\omega}^2{\uparrow}c')$$

$$\Phi_c(\vec{\omega}^1{\uparrow}c) \neq \Phi_c(\vec{\omega}^2{\uparrow}c) \text{ or } \mathit{Tem}(\vec{\omega}^1{\uparrow}c) \neq \mathit{Tem}(\vec{\omega}^2{\uparrow}c) \text{ or } S_1 \text{ does not precede } S_2$$

Recall that a barrier is executed only among instances whose processor partitioning functions are equal. Thus the first line implies that there are no barriers executed in inner loop levels between the execution of the two instances. The second line implies that conditions do not exist for barriers at the outermost level between processors that execute the two instances. Therefore, $S_1\vec{\omega}^1$ is not necessarily executed before $S_2\vec{\omega}^2$, and a contradiction exists. □

The execution ordering on statements is included in the temporal ordering, but the converse is not true, as shown by the following lemma.

Lemma 4.6: $S_1\vec{\omega}^1 < S_2\vec{\omega}^2 \Rightarrow S_1\vec{\omega}^1 \prec S_2\vec{\omega}^2$, but $S_1\vec{\omega}^1 \prec S_2\vec{\omega}^2 \not\Rightarrow S_1\vec{\omega}^1 < S_2\vec{\omega}^2$.
Proof: We first show $S_1\vec{\omega}^1 < S_2\vec{\omega}^2 \Rightarrow S_1\vec{\omega}^1 \prec S_2\vec{\omega}^2$. If $S_1\vec{\omega}^1 < S_2\vec{\omega}^2$, there are two cases. For the first case, if $\mathit{Tem}(\vec{\omega}^1{\uparrow}c') < \mathit{Tem}(\vec{\omega}^2{\uparrow}c')$, then $\mathit{Tem}(\vec{\omega}^1{\uparrow}c) < \mathit{Tem}(\vec{\omega}^2{\uparrow}c)$ since $c' \leq c$. For the second case, $\mathit{Tem}(\vec{\omega}^1{\uparrow}c) = \mathit{Tem}(\vec{\omega}^2{\uparrow}c)$ and S_1 precedes S_2. In both cases, we get $S_1\vec{\omega}^1 \prec S_2\vec{\omega}^2 \not\Rightarrow S_1\vec{\omega}^1 < S_2\vec{\omega}^2$

In order to show that the converse is not true, we only need to observe that there exists a scenario where $\Phi_1(\vec{\omega}^1{\uparrow}1) \neq \Phi_1(\vec{\omega}^2{\uparrow}1)$ which implies that $\forall\, c' \;\; \Phi_{c'}(\vec{\omega}^1{\uparrow}c') \neq \Phi_{c'}(\vec{\omega}^2{\uparrow}c')$ and $S_1\vec{\omega}^1 \not< S_2\vec{\omega}^2$, but it is also possible that $\mathit{Tem}(\vec{\omega}^1{\uparrow}c) < \mathit{Tem}(\vec{\omega}^2{\uparrow}c)$, in which case $S_1\vec{\omega}^1 \prec S_2\vec{\omega}^2$. □

4.8 Computation of synchronization targets

By inserting point-to-point synchronization for each dependence, a program can be executed without requiring barrier synchronizations between every statement as specified by the execution semantics. This section presents a general scheme for generating point-to-point synchronization to support dependences that contain certain types of array index expressions. For a given dependence, information about spatial and temporal relationships of statement instances can be merged with partitioning functions to allow implementation of processor-to-processor synchronization. Recalling the problem statement from Section 4.2, we need to derive for each processor p the set of processors with which it needs to synchronize and the dynamic relationship between synchronized statements.

4.8.1 Motivation

Consider the example in Figure 4-13. The vertical axis represents spatial coordinates 1 to 6 and the horizontal axis represents temporal coordinates 1 to 5. The spatial coordinates are partitioned into three processors p_1 through p_3. For sink instance $\langle 1, 5\rangle$, the source instances that result in a dependence are indicated by the arrows. To support the dependences, processor p_1 only needs to synchronize with processor p_2. Furthermore, processor p_1 only needs to synchronize with temporal coordinate 3 of processor p_2 since the instance $\langle 3, 2\rangle$ is executed before instance $\langle 4, 3\rangle$.

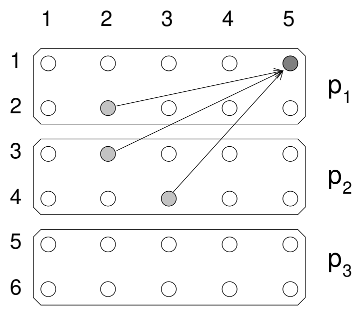


Figure 4-13: Dependence relationships across instances

As mentioned previously, the coordinates of statement instances can be separated into spatial and temporal components. The partitioning functions map the spatial components into processors, while the function Tem maps an instance into a timestamp by selecting its temporal coordinates. Synchronization relationships can then be computed for processors and timestamp values separately. The above subproblems can be formalized by introducing processor and temporal target functions. Before executing an instance $\vec{\omega}^2$ of the sink statement with timestamp $\vec{\tau}^2 = Tem(\vec{\omega}^2)$, the sink processor p must ensure that the source processors have executed particular instances of the source statement. For a dependence Δ, the *processor target function* $\mathcal{P}_\Delta(p, \widehat{\mathrm{T}})$ yields the set of source processors for that dependence. The *temporal target function* $\mathcal{T}_\Delta(p, \widehat{\mathrm{T}})$ yields the upper bound timestamp of the source statement instances required for synchronization. The argument p represents the sink processor and $\widehat{\mathrm{T}}$ represents a set of sink timestamps. This set depends on the lexical location of the respective synchronization check and is specified in the following section.

4.8.2 Static computation of synchronization targets

To effectively implement point-to-point synchronization, it is important that the possibly expensive process of computing spatial and temporal targets be avoided as much as possible at run time. If the computation of synchronization targets is too expensive, then the resulting code may perform no better or even worse than that of barrier synchronization schemes. To reduce the cost of deriving synchronization targets, the computations are performed statically and outside of loops whenever possible.

When implementing point-to-point synchronization, checks should clearly be placed before the sink statement and assertions should be placed after the source statement. Nevertheless, an open question remains on which loop level to place the primitives. In a general dependence Δ involving two statements S_1 and S_2, a barrier synchronization would normally be inserted at the innermost sequential loop level that encloses S_1 and S_2. In other words, the barrier is inserted inside any loops that enclose both S_1 and S_2 and outside any loops that are not shared by the statements. Point-to-point synchronization can either be inserted at the same loop level as barriers or in lower levels. One can imagine placing a synchronization check immediately before the sink statement to ensure that synchronization is not invoked until the data is truly needed. However, at such lower loop levels, the repeated execution of point-to-point synchronization is almost always more expensive than barriers. Thus we impose the requirement that point-to-point synchronization be inserted at the innermost loop level that encloses both S_1 and S_2. A synchronization can then be computed relatively inexpensively if its targets are independent of the inner loop levels that surround S_2.

An additional subtle factor involves the lexical placement of synchronization primitives. In Figure 4-14, two flow dependences exist between definitions and uses of a in statement S1. The first dependence S1$\langle i, j-1, k-1\rangle$ δ^f S1$\langle i, j, k\rangle$ is propagated from one iteration of j to the next. Thus it needs to be supported by point-to-point synchronization inside the loop j as illustrated. However, the second dependence S1$\langle i, j+1, k-1\rangle$ δ^f S1$\langle i, j, k\rangle$ is not propagated between iterations of j, but between iterations of i. Although we can still try to support the dependence between iterations of j, this would merely produce unnecessary assertions and checks. Instead, the synchronization primitives can be moved to the outer loop i to improve efficiency. The external field computed by the *markExt* function of Chapter 3 can be used to determine where source subarrays are propagated. In the first dependence, the subarray a[j,k] is not external to any loop, while the same subarray is external to the j loop in the second dependence. Thus rather than inserting synchronization primitives at the c-th innermost loop where c is the number of loops that enclose both source and sink statements, the definition of c can be modified to be the minimum of the number of loops that enclose both source and sink and the number of loops that enclose the loop specified by the source subarray external field.

```
do (i=1,100) {
  while (sync1[p+1] < i-1);
  do (j=1,100) {
    while (sync2[p-1] < <i,j-1>);
    doall (k=1,100)
      a[j,k] = a[j-1,k-1] + a[j+1,k-1];    /* S1 */
    sync2[p] = <i,j>;
  }
  sync1[p] = i;
}
```

Figure 4-14

With the lexical location of synchronization assertions and checks specified, we can compute the set of timestamps $\widehat{\mathrm{T}}$ to use for deriving synchronization targets. Let c be the sequential loop level of the synchronization primitives and let c' be the number of sequential loops that enclose S_2 so that $c' \geq c$. The synchronization check performed at level c must satisfy all dependences involving any instances at level c'. In other words, for each instance $S_2\vec{\omega}^2$ of the sink statement, any source instance producing a dependence must be represented in the functions $\mathcal{P}_\Delta(p, \widehat{\mathrm{T}})$ and $\mathcal{T}_\Delta(p, \widehat{\mathrm{T}})$. Therefore, the variable $\widehat{\mathrm{T}}$ must contain all timestamps $\vec{\tau}$ such that $\vec{\tau}\uparrow c = \langle \tau_1^2, \ldots, \tau_c^2\rangle$ where each τ_i^2 represents the index value at the i-th outermost sequential loop.

A trade-off exists between the costs of point-to-point synchronization and barrier synchronization. While point-to-point synchronization is effective for dependences involving small numbers of processors, barrier synchronizations are much more efficient in cases where many processors need to be synchronized with each other. For each dependence, if the number of source processors for a particular processor p exceeds a certain threshold θ, i.e. $|\mathcal{P}_\Delta(p, \widehat{\mathrm{T}})| > \theta$, then barrier synchronization can be used rather than point-to-point synchronization. This threshold is dependent on the speed of barrier synchronization on a particular machine as well as the amount of variance in execution times of code sections in a program.

In many cases, the computation of spatial synchronization targets can be done statically. If a spatial target $\mathcal{P}_\Delta(p, \widehat{\mathrm{T}})$ can be derived at compile time, then its value is completely independent of any possible set of sink timestamps $\widehat{\mathrm{T}}$. This property holds when source array indices that are `DOALL` loop indices correspond only to sink array indices that are also `DOALL` indices or constants. This scenario occurs in programs where certain array dimensions are accessed primarily in parallel while others are accessed primarily sequentially. Although static computation of targets allows very efficient implementation of synchronization, it is not absolutely necessary. Point-to-point synchronization can be used as long as their computation and execution can be done in less time than barrier synchronizations. Instead of requiring expressions to be linear functions of `DOALL` indices or constants, one can also allow expressions that are *invariant* with respect to $\widehat{\mathrm{T}}$. Invariance of an expression e can be defined as $\exists\, C \;\; \forall\, \vec{\omega}^2 \;\; Tem(\vec{\omega}^2) \in \widehat{\mathrm{T}} \Rightarrow e|S_2\vec{\omega}^2 = C$. Indices of sequential loops that enclose the synchronization check can be viewed as such invariant expressions.

4.8.3 Framework for deducing processor targets

For a given processor p, the processor target function $\mathcal{P}_\Delta(p, \vec{\tau}^2)$ can be computed from relationships of the dependence and partitioning functions of the relevant loops. The instance relationships of Section 4.5.3 can be developed further and combined with partitioning information to derive processor targets.

As before, let $a[\vec{e}^{\,1}]$ and $a[\vec{e}^{\,2}]$ be the array references of the source and sink statements of the dependence. Let $\vec{e}^{\,1} = (e^1_1, \ldots, e^1_n)$ and $\vec{e}^{\,2} = (e^2_1, \ldots, e^2_n)$. Recall that for a particular sink instance $S_2\vec{\omega}^2$, source instance coordinates can be computed separately. For a particular instance coordinate ω^k_j, let L^k_j be the loop associated with that coordinate and let I_j^k be the loop index of L_j^k. From Section 4.5.3, the set of source instances Ω that are dependent on a sink instance $\vec{\tau}^2$ can be defined as follows:

$$\Omega' = \prod_j \Omega_j$$

$$\text{where } \Omega_j = Dom(\omega_j^1) \cap \bigcap \sigma_j^i \text{ and}$$

$$\sigma_j^i = \begin{cases} f_1^{-1}(f_2(\omega_k^2)) & \text{if } e_i^1 = f_1(I_j^1) \text{ and } \exists\, k \;\; e_i^2 = f_2(I_k^2) \\ f_1^{-1}(e_i^2|S_2\vec{\omega}^2) & \text{if } e_i^1 = f_1(I_j^1) \text{ and } e_i^2 \text{ is not a linear induction var.} \\ \emptyset & \text{if } e_i^1 = C \text{ constant and } e_i^2 = f_2(I_k^2) \text{ and } C \neq f_2(\omega_k^2) \\ Dom(\omega_j^1) & \text{if } e_i^1 \text{ is not a linear induction var.} \end{cases}$$

In order to derive processor-to-processor relations from instance relations, we incorporate partitioning functions into the computation. Let the loop partitioning functions $\Phi_1 = \phi_1^1 \times \ldots \times \phi_{m_1}^1$ and $\Phi_2 = \phi_1^2 \times \ldots \times \phi_{m_2}^2$ map instances of the source and sink statements into processor partitions. Let the processor partitioning functions $\Psi_1 = \psi_1^1 \times \ldots \times \psi_{m_1}^1$ and $\Psi_2 = \psi_1^2 \times \ldots \times \psi_{m_2}^2$ map processors into partitions for the source and sink statements. Let $\mathrm{K}_1^1, \ldots, \mathrm{K}_{m_1}^1$ and $\mathrm{K}_1^2, \ldots, \mathrm{K}_{m_2}^2$ represent the processor partitioning sets for each loop.

A dependence exists if the source and sink array references can evaluate to the same value. By using the partitioning functions, the set of source processors p' that can exist in a dependence with a sink instance $\vec{\omega}^2$ can be written as:

$$\{p' : \vec{e}^{\,1}|S_1\Phi_1^{-1}(\Psi_1(p')) = \vec{e}^{\,2}|S_2\vec{\omega}^2\}$$

Since the result of $\Phi_1^{-1}(\Psi_1(p'))$ actually represent a set of instances, the above statement really means that a dependence exists if the equality holds for any instance in the set.

For a particular sink processor p and timestamp $\vec{\tau}^2$, then set of source processors that can exist in a dependence is thus:

$$\mathcal{P}_\Delta(p, \{\vec{\tau}^2\}) = \{p' : \vec{e}^{\,1}|S_1\Phi_1^{-1}(\Psi_1(p')) = \vec{e}^{\,2}|S_2(\Phi_2^{-1}(\Psi_2(p)) \cap Tem^{-1}(\vec{\tau}^2))\}$$

Again, the above statement really means that a dependence exists if the equality holds for any pair of instances in the sets. Naturally, the above equation only holds if statement S_2 is executable by processor p. If S_2 is an assignment statement and $p \neq p_{seq}^{\Psi_2(p)}$, then S_2 is not executed by p and no dependence exists between p and any other processors. Likewise, the processors derived by $\mathcal{P}_\Delta(p, \{\vec{\tau}^2\})$ should be limited to those that can execute statement S_2. These restrictions are straightforward and are not considered in the following derivation.

For cases where source array indices are linear functions of loop indices, the desired source processors p' can be accurately computed. If an array index expression is of the form $f(I)$ where f is a linear function and I is the index of a loop, then the set of possible values of that expression for a processor p is equal to $f(\phi^{-1}(\psi(p)))$. For a particular sink processor p, a set of partitions K'_j can be derived for each source loop. The set of source processors $\mathcal{P}_\Delta(p, \widehat{\mathrm{T}})$ are exactly those that belong to a partition corresponding to each loop:

$$\mathcal{P}_\Delta(p, \widehat{\mathrm{T}}) = \{p' : \forall\, j \;\; 1 \leq j \leq m_1 \;\; \exists\, \kappa'_j \in \mathrm{K}'_j \;\; p' \in \kappa'_j\}$$

$$\text{where } \mathrm{K}'_j = \mathrm{K}^1_j \cap \bigcap_{i=1}^{n} \chi^j_i \text{ and}$$

$$\chi^j_i = \begin{cases} \phi^1_j(f_1^{-1}(f_2({\phi^2_k}^{-1}(\psi^2_k(p))))) & \text{if } e^1_i = f_1(I^1_j) \text{ and } e^2_i = f_2(I^2_k) \text{ and } L^2_k \text{ is parallel} \quad (4.1) \\ \phi^1_j(f_1^{-1}(e^2_i | S_2\, Tem^{-1}(\widehat{\mathrm{T}}))) & \text{if } e^1_i = f_1(I^1_j) \text{ and } e^2_i \text{ is invariant w.r.t } \widehat{\mathrm{T}} \quad (4.2) \\ \phi^1_j(f_1^{-1}(f_2(Dom(\omega^2_k)))) & \text{if } e^1_i = f_1(I^1_j) \text{ and } e^2_i = f_2(I^2_k) \text{ and } L^2_k \text{ is sequential} \quad (4.3) \\ \emptyset & \text{if } e^1_i = C \text{ and } e^2_i = f_2(I^2_k) \text{ and } C \notin f_2({\phi^2_k}^{-1}(\psi^2_k(p))) \quad (4.4) \\ \mathrm{K}^1_j & \text{otherwise} \quad (4.5) \end{cases}$$

The sets χ^j_i and K'_j are subsets of the partitioning set K^1_j and can again be viewed as filters on K^1_j. The set χ^j_i filters out processor partitions that cannot be part of the dependence due to array accesses e^1_i and e^2_i. The intersection of filters $\bigcap \chi^j_i$ yields the set of partitions that can be part of the dependence for the j-th source loop. Note that since all sequential loops are mapped to the same partitions, the above filters do not really affect any source sequential loop coordinates. Consequently, the set of source processors contain those processors that belong to a resulting partition for each `DOALL` loop.

The following lemma shows that the processor target function $\mathcal{P}_\Delta(p, \widehat{\mathrm{T}})$ is correct. If a dependence exists between two instances where the sink instance has a timestamp in $\widehat{\mathrm{T}}$, then any processor that can execute the source instance is included in the processor target function of any sink processor that can execute the sink instance.

<u>Lemma 4.7</u>: If a dependence exists between two instances $\Delta = S_1\vec{\omega}^1 \;\delta\; S_2\vec{\omega}^2$ and $Tem(\vec{\omega}^2) \in \widehat{\mathrm{T}}$, then for every sink processor $p \in \Psi_2^{-1}(\Phi_2(\vec{\omega}^2))$, the source processors are included in the processor target function: $\Psi_1^{-1}(\Phi_1(\vec{\omega}^1)) \subset \mathcal{P}_\Delta(p, \widehat{\mathrm{T}})$.

Proof: We introduce the notation $\circledcirc$ to denote the following:

$$x \circledcirc S \iff \exists\, R \in S \;\; x \in R$$

For source and sink array accesses $a[\vec{e}^{\,1}]$ and $a[\vec{e}^{\,2}]$, a dependence exists if and only if for every i, $e_i^1|S_1\vec{\omega}^1 = e_i^2|S_2\vec{\omega}^2$. We can show inductively for each i that $p' \in \Psi_1^{-1}(\Phi_1(\vec{\omega}^1)) \Rightarrow \forall\, i' \leq i \;\; \forall\, j \;\; p' \circledcirc \mathrm{K}_j^1 \,\cap\, \bigcap \chi_{i'}^j$ and hence $p' \in \mathcal{P}_\Delta(p, \widehat{\mathrm{T}})$. For the case where a is a scalar and $i = 0$, the above is trivially true.

Inductively, assume that the above is true for $i-1$. We need to show that for all j, $p' \circledcirc \chi_i^j$. For the case where e_i^1 is not constant and not of the form $f_1(I_j^1)$, then $\chi_i^j = \mathrm{K}_j^1$ by case (4.5) and $p' \circledcirc \chi_i^j$. If e_i^1 is of the form $f_1(I_j^1)$, there are four cases:

If $e_i^2 = f_2(I_k^2)$ and L_k^2 is parallel, then $e_i^1|S_1\vec{\omega}^1 = e_i^2|S_2\vec{\omega}^2$ when $f_1(\omega_j^1) = f_2(\omega_k^2)$ or $\omega_j^1 \in f_1^{-1}(f_2(\omega_k^2))$. Since $\omega_k^2 \in {\phi_k^2}^{-1}(\psi_k^2(p))$ and $p' \in {\psi_j^1}^{-1}(\phi_1^j(\omega_j^1))$, we have $p' \circledcirc \chi_i^j$ by case (4.1).

If e_i^2 is invariant with respect to $\widehat{\mathrm{T}}$, then $\exists\, C \;\; \forall\, \vec{\omega}^2 \;\; Tem(\vec{\omega}^2) \in \widehat{\mathrm{T}} \Rightarrow e|S_2\vec{\omega}^2 = C$. Thus $e_i^1|S_1\vec{\omega}^1 = e_i^2|S_2\vec{\omega}^2$ implies that $f_1(\omega_j^1) = C$. Since $e_i^2|S_2\,Tem^{-1}(\widehat{\mathrm{T}}) = \{C\}$ and $\omega_j^1 = f_1^{-1}(C)$ and $p' \in {\psi_j^1}^{-1}(\phi_1^j(\omega_j^1))$, we have $p' \circledcirc \chi_i^j$ by case (4.2).

If $e_i^2 = f_2(I_k^2)$ and L_k^2 is sequential, then $e_i^1|S_1\vec{\omega}^1 = e_i^2|S_2\vec{\omega}^2$ when $f_1(\omega_j^1) = f_2(\omega_k^2)$ or $\omega_j^1 \in f_1^{-1}(f_2(\omega_k^2))$. Since $p' \in {\psi_j^1}^{-1}(\phi_1^j(\omega_j^1))$, we have $p' \circledcirc \chi_i^j$ by case (4.3).

If $e_i^1 = C$ where C is constant and $e_i^2 = f_2(I_k^2)$, then $e_i^1|S_1\vec{\omega}^1 = e_i^2|S_2\vec{\omega}^2$ when $C = f_2(\omega_k^2)$. Since $\omega_k^2 \in {\phi_k^2}^{-1}(\psi_k^2(p))$, the above is true only if $C \in f_2({\phi_k^2}^{-1}(\psi_k^2(p)))$. Case (4.4) is thus not satisfied and $\chi_i^j = \mathrm{K}_j^1$ by case (4.5) which implies that $p' \circledcirc \chi_i^j$.

Therefore, $p' \circledcirc \mathrm{K}_j'$ for every j, and $p' \in \mathcal{P}_\Delta(p, \widehat{\mathrm{T}})$. $\square$

4.8.4 Computation of processor targets

A more concrete algorithm can be presented for deducing processor targets when partitioning functions are more clearly specified. Let the processor partitioning functions be defined as masks of processor address bits $\psi(p) = address(p)\ \&\ mask$. Each processor partition can then be referred by the value of its masked bits. Let loop indices be partitioned contiguously into processor partitions with a *loop partition stride* of λ. For a `DOALL` loop from lo to hi, the loop partitioning function is defined as $\phi(c) = \lfloor (c - lo)/\lambda \rfloor$. In other words, each processor partition y executes indices $\phi^{-1}(y) = [\lambda y + lo, \lambda(y+1) + lo - 1]$ where the notation $[c_1, c_2]$ represents the set of integers from c_1 to c_2.

When both source and sink array indices are linear functions of `DOALL` loops as in case (4.1) above, relevant source processor partitions can be computed at compile time for each sink processor partition. In other words, when the two array indices at position i in the source and sink array reference are linear functions of loop indices, then the set of filtered partitions χ_i^j can be statically computed for each sink partition. For a particular dependence Δ, let the source and sink array references be $a[\ldots, f_1(I_1), \ldots]$ and $a[\ldots, f_2(I_2), \ldots]$ where linear functions $f_1(I_1) = \alpha_1 I_1 + \beta_1$ and $f_2(I_2) = \alpha_2 I_2 + \beta_2$ appear at the i-th array index of both references. Let j be the nesting level of the source `DOALL` loop corresponding to I_1 and let k be the nesting level of the sink loop corresponding to I_2. Let lo_1 and hi_1 be the loop bounds for the source loop and lo_2 and hi_2 be the loop bounds for the sink loop. Let λ_1 and λ_2 be the loop partition strides for the source and sink loops, respectively.

For a sink processor partition y, the loop indices managed by y are

$$\phi_k^{2\,-1}(y) = [\lambda_2 y + lo_2, \lambda_2(y+1) + lo_2 - 1]$$

The array indices managed by y at position i are thus

$$f_2(\phi_k^{2\,-1}(y)) = [\alpha_2\lambda_2 y + \alpha_2 lo_2 + \beta_2, \alpha_2\lambda_2(y+1) + \alpha_2 lo_2 - \alpha_2 + \beta_2]$$

Since we are interested in the cases where expressions $f_1(I_1)$ and $f_2(I_2)$ evaluate to the same value, the set of indices I_1 such that $I_1 \in f_1^{-1}(f_2((I_2))$ can be computed as:

$$f_1^{-1}(f_2(\phi_k^{2\,-1}(y))) = \left[\ \frac{\alpha_2}{\alpha_1}\lambda_2 y + \gamma\ ,\ \frac{\alpha_2}{\alpha_1}\lambda_2 y + \gamma + \frac{\alpha_2}{\alpha_1}(\lambda_2 - 1)\ \right]$$
$$\text{where } \gamma = \frac{\alpha_2}{\alpha_1} lo_2 + \frac{\beta_2 - \beta_1}{\alpha_1}$$

Using the source loop partition stride λ_1, the set of source partitions x that can affect the above set of I_1 indices can then be defined as:

$$\phi_j^1(f_1^{-1}(f_2(\phi_k^{2\,-1}(y)))) = \left[\ \frac{\alpha_2}{\alpha_1}\frac{\lambda_2}{\lambda_1} y + \frac{\gamma - lo_1}{\lambda_1}\ ,\ \frac{\alpha_2}{\alpha_1}\frac{\lambda_2}{\lambda_1} y + \frac{\gamma - lo_1}{\lambda_1} + \frac{\alpha_2}{\alpha_1}\left(\frac{\lambda_2 - 1}{\lambda_1}\right)\ \right]$$

For a particular sink processor partition y, the set of source processor partitions that can generate references to array a for dependence Δ can be specified as:

$$\phi_j^1(f_1^{-1}(f_2(\phi_k^{2\,-1}(y)))) = \left\{ x : x \in \left[\ \left\lfloor \frac{mult\ y + addlo}{div} \right\rfloor\ ,\ \left\lceil \frac{mult\ y + addhi}{div} \right\rceil\ \right] \right\}$$

with the following parameters:

$$\begin{aligned} mult &= \alpha_2 \lambda_2 \\ div &= \alpha_1 \lambda_1 \\ addlo &= \beta_2 - \beta_1 + \alpha_2 lo_2 - \alpha_1 lo_1 \\ addhi &= \beta_2 - \beta_1 + \alpha_2 lo_2 - \alpha_1 lo_1 + \alpha_2(\lambda_2 - 1) \end{aligned}$$

Typically, the upper and lower bounds of the source partition range can be computed at compile time and be used as run-time constants.

The above computation solves the processor partition relationships for the situation where both array indices are linear functions of `DOALL` loop indices. We wish to also derive processor partition sets for invariant expressions as in case (4.2) above. For a sink array index of value C and a source array index $f_1(I_1) = \alpha_1 I_1 + \beta_1$, the relevant source processor partition can be computed as:

$$f_1^{-1}(C) = \frac{C - \beta_1}{\alpha_1}$$

and

$$\phi_j^1(f_1^{-1}(C)) = \left\lfloor \frac{C - \beta_1 - \alpha_1 lo_1}{\alpha_1 \lambda_1} \right\rfloor$$

Note that this computation differs from the `DOALL` to `DOALL` computation in that the value C may change dynamically. Thus the calculation of source partitions must be performed at run time immediately before the synchronization check.

If processor partitioning functions are represented as masks of processor address bits, then each resulting partition can be represented by a sequence of bits. The resulting processor address can then be specified by performing a logical OR operation on the bits. When relationships involve only `DOALL` loop indices and constants, then the entire processor target function $\mathcal{P}_\Delta(p, \widehat{\mathrm{T}})$ can be computed statically for each processor.

4.8.5 Computation of temporal targets

For a sink processor p, the temporal target function $\mathcal{T}_\Delta(p, \widehat{\mathrm{T}})$ returns the timestamp $\vec{\tau}^1$ of source statement S_1 with which p needs to synchronize before executing instances with timestamp $\vec{\tau}^2$ of S_2. Unlike the processor target function, the temporal target function only needs to return one value. Only the upper bound of the timestamps that produce dependences is needed since the execution of an instance with the upper bound timestamp implies that all lower instances have been executed.

Before the upper-bound source timestamp is computed, the set of all source timestamps needs to be derived. As with processor targets, individual timestamps can be computed separately as follows:

$$\mathcal{T}'_\Delta(p, \widehat{\mathrm{T}}) = \prod_{j\,:\,L^1_j \text{ sequential}} \mathrm{T}'_j$$

$$\text{where } \mathrm{T}'_j = \mathrm{T}_j \cap \bigcap_{i=1}^{n} \xi^j_i \text{ and}$$

$$\xi^j_i = \begin{cases} f_1^{-1}(f_2({\phi^2_k}^{-1}(\psi^2_k(p)))) & \text{if } e^1_i = f_1(I^1_j) \text{ and } e^2_i = f_2(I^2_k) \text{ and } L^2_k \text{ is parallel} \quad (4.6) \\ f_1^{-1}(e^2_i | S_2\, Tem^{-1}(\widehat{\mathrm{T}})) & \text{if } e^1_i = f_1(I^1_j) \text{ and } e^2_i \text{ is invariant w.r.t } \widehat{\mathrm{T}} \quad (4.7) \\ f_1^{-1}(f_2(Dom(\omega^2_k))) & \text{if } e^1_i = f_1(I^1_j) \text{ and } e^2_i = f_2(I^2_k) \text{ and } L^2_k \text{ is sequential} \quad (4.8) \\ Dom(\omega^1_k) & \text{otherwise} \quad (4.9) \end{cases}$$

The following lemma shows that the intermediate temporal target function $\mathcal{T}'_\Delta(p, \widehat{\mathrm{T}})$ is correct. If a dependence exists between two instances where the sink timestamp is equal to $\vec{\tau}^2$, then the timestamp of the source instance is included in the temporal target function of any processor that can execute the sink instance.

Lemma 4.8: If a dependence exists between two instances $\Delta = S_1\vec{\omega}^1 \;\delta\; S_2\vec{\omega}^2$ and $Tem(\vec{\omega}^2) \in \widehat{\mathrm{T}}$, then for every $p \in \Psi_2^{-1}(\Phi_2(\vec{\omega}^2))$, $Tem(\vec{\omega}^1) \in \mathcal{T}'_\Delta(p, \widehat{\mathrm{T}})$.
Proof: The proof strategy is very similar to that of Lemma 4.7 and is omitted. □

The product of the temporal coordinate sets T'_j represent the timestamps in which the source can access the same array elements as the given sink instances, with one exception: The source instances cannot be greater than or equal to the sink instances. Intuitively, synchronization should not have to be done for accesses that have not occurred. For a sink instance $S_2\vec{\omega}^2$, the temporal target function can be defined as the upper bound of the set of past timestamps:

$$\mathcal{T}_\Delta(p, \widehat{\mathrm{T}}) = \text{upper bound of } \{\vec{\tau} : \vec{\tau} \in \mathcal{T}'_\Delta(p, \widehat{\mathrm{T}}) \text{ and } S_1\vec{\tau} < S_2\vec{\tau}^2\}$$

Given each coordinate set T'_j, the algorithm in Figure 4-15 shows how to derive an upper bound of the product of coordinates that is not greater than the sink timestamps. The idea involves taking upper bounds of the coordinate sets from the outermost loop inward. From the definition of tuple comparison, this corresponds to taking upper bounds starting with the most significant coordinates. As long as all previous derived coordinates are equal to the sink coordinates, we must ensure that the current source coordinate does not exceed the sink coordinate. This is represented by the $equalFlag$ variable. The argument $\vec{\tau}^2$ represents the lower bound of the sink timestamp set $\widehat{\mathrm{T}}$ and can be computed by observing that for all c outer sequential loops, the respective coordinate value of each $\vec{\tau} \in \widehat{\mathrm{T}}$ is equal to the current loop index value. For inner sequential loops, the coordinate in $\vec{\tau}^2$ is not used and can be set to $-\infty$.

Finally, the source timestamp result can be represented as:

$$\mathcal{T}_\Delta(p, \widehat{\mathrm{T}}) = \langle \tau'_1, \ldots, \tau'_{n_1} \rangle$$

Algorithm $maxTem(S_1, S_2, \vec{\tau}^2, \vec{\mathrm{T}}')$:

Let c be the sequential loop level of the synchronization check.

$equalFlag = True$
for j from 1 to n_1 do
  if (not $equalFlag$) or $j > c$ then
    τ'_j = upper bound of T'_j
  else
    τ'_j = upper bound of $\{\tau \in \mathrm{T}'_j : \tau \leq \tau^2_j\}$
    if $\tau'_j < \tau^2_j$ then $equalFlag = False$
if $\langle \tau'_1, \ldots, \tau'_c \rangle = \vec{\tau}^2 \uparrow c$ and S_1 does not precede S_2 then
  $\tau'_c = \tau'_c - 1$

Figure 4-15: Computation of temporal instance upper bound

Note that this algorithm actually derives timestamps $\vec{\tau}^1$ that are less than $\vec{\tau}^2$ in the temporal ordering $\prec$. Since the ordering on instances implies the ordering on timestamps, any source instances $S_1\vec{\omega}^1$ such that $S_1\vec{\omega}^1 < S_2\vec{\omega}^2$ implies that $S_1\vec{\tau}^1 \prec S_2\vec{\tau}^2$ and $\vec{\tau}^1$ is included in $\mathcal{T}_\Delta(p, \widehat{\mathrm{T}})$. All that remains is to show that the function $\mathcal{T}_\Delta(p, \widehat{\mathrm{T}})$ produces all timestamps that are less than the lower-bound sink timestamp $\vec{\tau}^2$. The first lemma shows that the temporal target function returns an upper bound of the set of all source timestamps that are less than the sink timestamps, and the second shows that the function returns the least upper bound of that set.

Lemma 4.9: If $\vec{\tau}^1 \in \mathcal{T}'_\Delta(p, \widehat{\mathrm{T}})$ and $S_1\vec{\tau}^1 \prec S_2\vec{\tau}^2$, then $\vec{\tau}^1 \leq \mathcal{T}_\Delta(p, \widehat{\mathrm{T}})$.
Proof: By contradiction, assume that $\vec{\tau}^1 \in \mathcal{T}'_\Delta(p, \widehat{\mathrm{T}})$ but $\vec{\tau}^1 > \mathcal{T}_\Delta(p, \widehat{\mathrm{T}})$. This implies that for some j, $\tau^1_j > \tau'_j$ and $\forall\ i < j\ \ \tau^1_i = \tau'_i$. Let k be the iteration in the algorithm where *equalFlag* is set to *False*. Then $\forall\ i < k\ \ \tau'_i = \tau^2_i$. If $j > k$ or $j > c$, then $\tau'_j =$ upper bound of T'_j and $\tau^1_j > \tau'_j$ cannot occur. If $j \leq k$, $\tau'_j =$ upper bound of $\{\tau \in \mathrm{T}'_j : \tau \leq \tau^2_j\}$. Since $S_1\vec{\tau}^1 \prec S_2\vec{\tau}^2 \Rightarrow \tau^1_j \leq \tau^2_j$, the condition $\tau^1_j > \tau'_j$ cannot occur. □

In order to show that $\mathcal{T}_\Delta(p, \widehat{\mathrm{T}})$ is the least upper bound of source timestamps that are less than the sink timestamp, one only needs to show that $\mathcal{T}_\Delta(p, \widehat{\mathrm{T}})$ itself is less than the sink timestamp.

Lemma 4.10: For sink processor p, the following holds:

$$\forall\ \vec{\omega}^1, \vec{\omega}^2\ \ Tem(\vec{\omega}^1) = \mathcal{T}_\Delta(p, \widehat{\mathrm{T}}) \text{ and } Tem(\vec{\omega}^2) \in \widehat{\mathrm{T}} \Rightarrow S_1\vec{\omega}^1 \prec S_2\vec{\omega}^2$$

Proof: Let $\vec{\tau}^1 = Tem(\vec{\omega}^1) = \mathcal{T}_\Delta(p, \widehat{\mathrm{T}})$. If we can show that $S_1\vec{\tau}^1 \prec S_2\vec{\tau}^2$, then $S_1\vec{\omega}^1 \prec S_2\vec{\omega}^2$ since $\vec{\tau}^2$ is a lower bound of $\widehat{\mathrm{T}}$. By contradiction, suppose that $S_1\vec{\tau}^1 \not\prec S_2\vec{\tau}^2$. There are two cases:

(a) There exists $j \leq c$ such that $\tau'_j > \tau^2_j$ and $\forall\ i < j\ \ \tau'_i = \tau^2_i$. Then *equalFlag* is true for all iterations before j. Therefore, the algorithm forces $\tau'_j \leq \tau^2_j$, which produces a contradiction.

(b) For all $j < c$, $\tau'_j = \tau^2_j$ and S_1 doesn't precede S_2. Then the final clause of the algorithm is invoked and the result produces $\tau'_c < \tau^2_c$. □

We can now show that the above derivations of $\mathcal{P}_\Delta(p, \widehat{\mathrm{T}})$ and $\mathcal{T}_\Delta(p, \widehat{\mathrm{T}})$ are correct: If a dependence exists between two instances and if the source instance is executed before the sink instance, then synchronization is provided for them.

Claim 4.11: If $\Delta = S_1\vec{\omega}^1\ \delta\ S_2\vec{\omega}^2$ and $S_1\vec{\omega}^1$ is executed before $S_2\vec{\omega}^2$, then for each processor p' that executes $S_1\vec{\omega}^1$ and each processor p that executes $S_2\vec{\omega}^2$, synchronization is performed between p' and p.
Proof: From Lemma 4.5, $S_1\vec{\omega}^1$ is executed before $S_2\vec{\omega}^2$ implies that $S_1\vec{\omega}^1 < S_2\vec{\omega}^2$ and $S_1\vec{\omega}^1 \prec S_2\vec{\omega}^2$. We need to show that for each $p \in \Psi_2^{-1}(\Phi_2(\vec{\omega}^2))$, the following are true:

(a) $\Psi_1^{-1}(\Phi_1(\vec{\omega}^1)) \in \mathcal{P}_\Delta(p, \widehat{\mathrm{T}})$

(b) $Tem(\vec{\omega}^1) \leq \mathcal{T}_\Delta(p, \widehat{\mathrm{T}})$

where $\widehat{\mathrm{T}}$ is defined as above so that $Tem(\vec{\omega}^2) \in \widehat{\mathrm{T}}$. The first requirement is immediate from Lemma 4.7. The second can be satisfied by applying Lemma 4.8 and Lemma 4.9. □

In order to improve efficiency, the source timestamp $\vec{\tau}^1$ can be restricted to be a constant offset from the lower-bound sink timestamp $\vec{\tau}^2$. This can be expressed as follows:

$$\mathcal{T}_\Delta(p, \widehat{\mathrm{T}}) = [\vec{\tau}^2 \uparrow c - \vec{d}(p)] \parallel \langle \infty, \ldots, \infty \rangle$$

where c is the number of outermost common sequential loops of S_1 and S_2. The notation $\parallel$ applied to the ∞ terms represent the concatenation of sequential loop indices surrounding S_1 that do not enclose S_2. Condition (4.8)above is then modified as follows: The multiplicative factors of both linear functions must be the same and the two loop indices must be the same ($j = k$).

4.8.6 An example

The above derivations can be illustrated by considering their application to the code in Figure 4-16, a variant of Figure 4-4b. Although other dependences exist, we focus our attention on the flow dependence Δ involving array `a` from `S1` to `S2`. For both assignment statements, the instance can be represented as a 3-tuple $\langle i, j, k \rangle$. The relation `S1`$\langle i_1, j_1, k_1 \rangle <$ `S2`$\langle i_2, j_2, k_2 \rangle$ holds if and only if $i_1 < i_2$ or $i_1 = i_2$ and $j_1 < j_2$. By assuming that the target machine has 100 processors, each `DOALL` loop is partitioned one iteration per processor and the processor number is interchangeable with the spatial instance. Since there are no nested `DOALL` loops, processor partitions are interchangeable with processors and spatial instances. Assume that the variable `X` has an unknown value.

```
do (i=1,100) {
  do (j=1,100) {
    doall (k=1,100)
      a[i,i,k+1] = ...;         /* S1 */
    ...
    doall (k=1,100)
      ... = a[i-3,x,k-1];       /* S2 */
  }
}
```

Figure 4-16

The processor target of the dependence $\mathcal{P}_\Delta(p, \langle i_2, j_2\rangle)$ can be computed by considering each instance `S2`$(\langle i_2, j_2, p\rangle)$. For each array reference coordinate i, the filters χ_i^j are defined as follows:

$$\chi_1^1 = \text{all processors}$$
$$\chi_2^1 = \text{all processors}$$
$$\chi_3^1 = p - 2$$

Since the intersection of the filters yields $p-2$, the spatial target is thus $\mathcal{P}_\Delta(p, \langle i_2, j_2\rangle) = p - 2$. Each processor p needs to synchronize with processor $p-2$.

The temporal target $\mathcal{T}_\Delta(p, \langle i_2, j_2\rangle)$ can be computed by considering the temporal filters ξ_i^j for each array reference coordinate i and each sequential loop coordinate j:

$$\xi_1^1 = i_2 - 3 \qquad \xi_1^2 = \text{all integers}$$
$$\xi_2^1 = \text{all integers} \qquad \xi_2^2 = \text{all integers}$$
$$\xi_3^1 = \text{all integers} \qquad \xi_3^2 = \text{all integers}$$

The temporal target can be computed by taking intersections of the filters for each sequential loop, yielding $\mathcal{T}_\Delta(p, \langle i_2, j_2\rangle) = \langle i_2 - 3, \infty\rangle$. This implies that before executing statement `S2` of iteration `i` and `j`, one must wait for the completion of the entire loop `j` of iteration `i-3`. As an interesting observation, note that if the above access of `a` in `S2` were `a[i,x,k-1]` instead of `a[i-3,x,k-1]`, then the temporal target function would yield $\langle i_2, j_2\rangle$. Also note that if $x = k$, then synchronization needs to be performed only when the first two array indices of `S2` are equal, which implies that $i_2 - 3 = p$. However, the separate computation of source instance coordinates does not allow us to readily take advantage of this fact.

4.9 Implementation issues

Recall that point-to-point synchronization is performed by the source processor writing a value to a synchronization variable and the sink processor spin-locking until the variable reaches a certain value. For a dependence Δ and a source processor p, the value written to the synchronization variable `sync[`p`]` is the timestamp $\vec{\tau}^1$ of the source statement. For a sink processor p, the set of source processors with which to synchronize is represented by the set $\mathcal{P}_\Delta(p, \widehat{\mathrm{T}})$. For each processor p' in that set, the sink processor spin-locks until `sync[`p'`]` $\geq \mathcal{T}_\Delta(p, \widehat{\mathrm{T}})$. In this section, we first give an example of an algorithm to compute static synchronizations, followed by a discussion on implementation of timestamps.

4.9.1 An algorithm

Following the above derivations, an algorithm can be presented for static computation of processor synchronization targets. To keep the presentation simple, the algorithm as given makes the assumption that processor targets are derived completely statically. For each processor at a dependence, the set of processors with which to synchronize is computed entire at compilation. Thus expressions are required to be functions of loop indices or compile-time constants. No allowance is presented here for non-constant loop-invariant expressions. In a real implementation, such constraints would of course be relaxed to allow for greater range of point-to-point synchronization support. We also assume that the value of θ is small and impose the constraint that the index of each parallel loop that encloses S_1 appear in at least one array index and is filtered by a parallel loop index or a constant. If this condition is not met, then some array location can be accessed by many partitions of the unrepresented loops. As a consequence, each sink processor would be required to synchronize with most processors in the partitions and likely exceed the limit θ of processors. This can be viewed as only providing support for the above cases (4.1) and (4.2) with constants.

The algorithm $staticSync(p, \Delta)$ in Figure 4-17 aims to compute the set of processors with which the sink processor p needs to synchronize for dependence Δ. Note that the entire algorithm can be run statically to produce a collection of source processors. If one were to allow for loop-invariant expressions, then some parts of the calculation would need to be performed dynamically, and provisions must be made for merging the static and dynamic results.

Observe that as long as each source parallel loop index is represented once in the source array reference, it does not matter how many other expressions in the array reference are unknown. This can be explained by pointing out that the expressions in each array dimension can be viewed as filters on the space of source instances with which to synchronize. Unknown expressions merely imply that the respective array dimension does not filter out any source instances. As long as other dimensions have filtered out enough source instances to allow point-to-point synchronization to be done, the unknown dimensions can be ignored. This feature can prove to be very effective in applications where much is known about some array dimensions while little is known about other dimensions.

Algorithm $staticSync(p, \Delta)$:

Let $\vec{\kappa}$ be the partitions for p at S_2.
Initialize $\vec{\mathrm{K}}(p)$ to be the product of partitioning sets for S_1.

for each source and sink array index expression e_i^1 and e_i^2 do
  if $e_i^1 = f_1(I_j)$ and I_j is parallel then
    if $e_i^2 = f_2(I_k)$ and I_k is parallel then
      $\mathrm{K}_j(p) = \mathrm{K}_j(p) \cap \phi_j^1(f_1^{-1}(f_2(\phi_k^{2\ -1}(\kappa_k))))$
    else if $e_i^2 = C$ where C is constant then
      $\mathrm{K}_j(p) = \mathrm{K}_j(p) \cap \phi_j^1(f_1^{-1}(A))$
  else if $e_i^1 = C$ where C is constant then
    if $e_i^2 = f_2(I_k)$ and I_k is parallel and $C \notin f_2(\phi_k^{2\ -1}(\kappa_k))$ then
      return $\emptyset$
if any source parallel loop index is not filtered then
  implement barrier

return $\{p' : p' \text{ has partitions in } \vec{\mathrm{K}}(p)\}$

Figure 4-17: Static computation of processor targets

4.9.2 Implementation of synchronization primitives

The actual implementation of a point-to-point synchronization involves writing and reading timestamp tuples. Although supporting tuples requires the allocation of several words of memory for each synchronization variable, tuples are not the only reason for this requirement. On cache-coherent machines, the synchronization variables themselves need to occupy separate cache lines to avoid thrashing when other variables are written. Since cache lines on many machines are 4 to 8 words long, supporting timestamp tuples may not incur much additional memory costs.

Even though tuples may not require much extra memory, writing and reading the words that correspond to tuple values can require a large amount of additional time. However, tuple values can be written and read one coordinate at a time. In the critical innermost loops, tuple values can be written and checked by accessing only one word, as demonstrated by the sample code in Figure 4-18. When performing a tuple write, it is important that the value stored in memory never exceeds the actual tuple value. Hence the less significant coordinates are zeroed before a coordinate value is updated.

To check a tuple, one must ensure that spin-locking is done only on cases where the stored value is less than the desired tuple. In the example code, checks in outer loops ensure that tuple values of more significant coordinates are at least equal to the desired value. Therefore, it is only necessary to check for the current coordinate and for higher coordinates being higher than their values. In the typical case of a synchronization being satisfied, only the first test is required before the `WHILE` loop is exited. Further tests are done only during spin-locking or in the relatively rare case that a higher coordinate has been updated by processor $p-1$.

```
do (i=1,100) {
  sync[p][3] = 0;
  sync[p][2] = 0;
  sync[p][1] = i;
  while (sync[p-1][1]<i);
  ...
  do (j=1,100) {
    sync[p][3] = 0;
    sync[p][2] = j;
    while (sync[p-1][2]<j && sync[p-1][1]==i);
    ...
    do (k=1,100) {
      sync[p][3] = k;
      while (sync[p-1][3]<k && sync[p-1][2]==j && sync[p-1][1]==i);
      ...
    }
  }
}
```

Figure 4-18: Code to assert and check for tuple $\langle i,j,k \rangle$ of processor $p-1$

When overhead for tuple support becomes significant, one can abandon the entire tuple scheme in some cases. When all relevant loop bounds are constants or equal across all processors, then all processors always execute the same number of iterations. In such cases, the iteration space can be flattened to one dimension, and one can perform synchronization merely by maintaining a counter on each processor to represent the one-dimensional iteration number. A synchronization check then involves merely checking that the synchronization array value of other processors are not less than the current counter. Of course, this technique does not allow for synchronization with past timestamps since one is in effect always synchronizing with the most recent timestamp. Despite its disadvantages, this scheme is used for the current implementation of this thesis due to its efficiency and ease.

4.10 Deadlock avoidance

By inserting synchronization assertions and checks for every dependence in a program, its execution can be carried without performing barrier synchronizations between every statement as specified in the semantics. However, we need to ensure that no deadlocks are introduced due to point-to-point synchronization. This can be done by adding additional assertions in cases involving conditional execution.

As shown in section 4.2.3, naively inserting synchronization primitives can result in deadlock conditions when conditionals are present. If an assertion is done in a conditional, any checks of that assertion may deadlock if the assertion is not invoked. Intuitively, deadlocks can be avoided by ensuring that for any synchronization, if a control flow path between two points contain an assertion, then every control flow path between the two points must contain the assertion.

Branches in structured control flow occur due to two types of statements: conditionals and sequential loops. Thus assertions need to be inserted to account for branches due to these statements. The transformation $\mathcal{Z}[S]$ can be applied to all statements S in a program in a bottom-up fashion according to the following rules:

1. In a sequential loop, if an assertion of the timestamp $\langle \tau_1, \ldots, \tau_n \rangle$ appears in the loop body, then the assertion of timestamp $\langle \tau_1, \ldots, \tau_m, \infty, \ldots, \infty \rangle$ is added at the end of the loop where m is the number of sequential loops that enclose S.

2. In a conditional statement `if (`V`)` S_1 `else` S_2, if an assertion of the timestamp $\langle \tau_1, \ldots, \tau_n \rangle$ appears in S_1, then an assertion of $\langle \tau_1, \ldots, \tau_m, \infty, \ldots, \infty \rangle$ is inserted at the beginning of the body of S_2 where m is the number of sequential loops that enclose S. Assertions in S_2 are added to the beginning of S_1 in the same manner.

The first rule accounts for the case when the loop body is not executed at all or when loop limits are not known statically. The second rule specifies that any assertions that occur on one branch of the conditional must also be done before any code in the other branch of the conditional is executed. Note that the monotonicity of assertions is still maintained in both cases since any future assertions of the same synchronization is done in the context of a greater timestamp than the one asserted.

When applied to all statements in a program, the above rules serve to satisfy the requirement that any control flow path between two points contain the same assertions. In order to prove that the application of these rules produces a program with no synchronization-induced deadlocks, an ordering on statement instances is needed. However, the execution ordering is inappropriate in this case due to the fact that synchronization between two instances does not always imply that the source instance is less than the sink instance in the execution ordering. Instead, the temporal ordering satisfies the above characteristic as shown in Lemma 4.10. Its definition is repeated below:

$$S_1\vec{\omega}^1 \prec S_2\vec{\omega}^2 \iff \vec{\tau}^1{\uparrow}c < \vec{\tau}^2{\uparrow}c \text{ or}$$
$$\vec{\tau}^1{\uparrow}c = \vec{\tau}^2{\uparrow}c \text{ and } S_1 \text{ precedes } S_2$$
where $\vec{\tau}^1 = Tem(\vec{\omega}^1)$ and $\vec{\tau}^2 = Tem(\vec{\omega}^2)$

and c is the number of sequential loops that enclose S_1 and S_2

The following lemma shows that if an assertion appears in the text of a statement, then an equivalent or greater assertion is done on any execution of the transformed statement.

<u>Lemma 4.12</u>: For a statement S and a statement S' that is a descendant of S, if an assertion of $\vec{\tau}'$ appears after statement S' for instance $S\vec{\omega}$, then any execution of $\mathcal{Z}[S]\vec{\omega}$ produces an assertion of $\vec{\tau}'' \geq \vec{\tau}'$.

<u>Proof</u>: Let $Tem(\vec{\omega}) = \vec{\tau}$. For $S = S'$, the lemma clearly holds. The proof for $S \neq S'$ is by structural induction on the statement S. Let c be the number of sequential loops that enclose S. Note that $\vec{\tau}'{\uparrow}c = \vec{\tau}$.

$S = [V = E]$: $S = S'$.

S = [`if (`V`)` S_a `else` S_b]: S' is a descendant of either S_a or S_b. Without loss of generality, assume that S' is a descendant of S_a. On any execution of $\mathcal{Z}[S]\vec{\omega}$, if $\mathcal{Z}[S_a]\vec{\omega}$ is executed, then the lemma is true by induction. If $\mathcal{Z}[S_b]\vec{\omega}$ is executed, then rule 2 above specifies that an assertion of $\vec{\tau}\|\langle\infty\ldots,\infty\rangle \geq \vec{\tau}'$ is done.

S = [`while (`V`)` S_a] and S = [`do (`I=K_1`,`K_2`,`K_3`)` S_a]: S' is a descendant of S_a. On any execution of $\mathcal{Z}[S]\vec{\omega}$, if S_a is executed, then the lemma holds by induction. If S_a is not executed, then from rule 1 above, the assertion of $\vec{\tau}\|\langle\infty\ldots,\infty\rangle \geq \vec{\tau}'$ is done.

S = [`doall (`I=K_1`,`K_2`,`K_3`)` S_a]: True by induction.

S = [`{`S_a S_b`}`]: S' is a descendant of either S_a or S_b. Without loss of generality, assume that S' is a descendant of S_a. Then the lemma holds by induction. □

In deriving the proof of deadlock avoidance, we need to ensure that assertions are done in order and before any instances that follow the asserted value. In addition, assertions that are unordered with respect to an instance need to be accounted for. The following lemma shows that at each statement instance, any assertion at a timestamp that is not greater than the current timestamp has been done.

Lemma 4.13: For a processor p executing a statement instance $\mathcal{Z}[S]\vec{\omega}$ where $Tem(\vec{\omega}) = \vec{\tau}$, for any assertion by p of $\vec{\tau}'$ at another statement S' such that $S'\vec{\tau}' \not\succeq S\vec{\tau}$, an assertion of $\vec{\tau}'' \geq \vec{\tau}'$ has been done.

Proof: If $S'\,Tem^{-1}(\vec{\tau}') \not\succeq S\vec{\omega}$, then either $\vec{\tau}'\uparrow c \not> \vec{\tau}\uparrow c$ or S does not precede S' and $\vec{\tau}'\uparrow c = \vec{\tau}\uparrow c$. There are three cases:

(a) $\vec{\tau}'\uparrow c < \vec{\tau}\uparrow c$: Then for some $j \leq c$, $\tau'_j < \tau_j$. Let L be the j-th outermost sequential loop. the assertion of $\vec{\tau}'$ would have been done in a previous iteration of L. By Lemma 4.12, the previous execution of the body of L would have asserted $\vec{\tau}'' \geq \vec{\tau}'$.

(b) $\vec{\tau}'\uparrow c = \vec{\tau}\uparrow c$ and S' precedes S: Then let S'' be the sequence `{` S_a S_b `}` such that S' is a descendant of S_a and S'' is a descendant of S_b. Then by Lemma 4.12, the execution of $\mathcal{Z}[S_a]\vec{\omega}$ produces an assertion of $\vec{\tau}'' \geq \vec{\tau}'$.†

(c) $\vec{\tau}'\uparrow c = \vec{\tau}\uparrow c$ and no precedence relationship exists between S' and S. Then there exists a conditional statement S'' such that S' and S are in separate clauses of the conditional. Without loss of generality, let S'' = `if (`e`)` S_a `else` S_b such that S' is a descendant of S_a and S is a descendant of S_b. Then by rule 2 above, an assertion of $\vec{\tau}'' \geq \vec{\tau}'$ is done before the body of S_b is executed. Therefore an assertion of $\vec{\tau}'' \geq \vec{\tau}'$ is done before S is executed. □

Using the above lemmas, we can now show that the transformation $\mathcal{Z}$ prevents deadlock conditions due to synchronization from occurring. By contradiction, if a deadlock occurs, then each processor is waiting for some synchronization variable to reach a value. When a processor p_2 waits for an assertion from processor p_1, then Lemma 4.13 implies that p_2 is farther along in the program than p_1 in some intuitive sense. However, this waiting relationship eventually produces a cycle of processor relationships which then implies that some processor is farther along in the computation than itself. Hence, a contradiction arises.

† The predicates of conditionals and loops and be viewed as being in a sequence with the statement body.

Claim 4.14: No deadlocks occur due to synchronization in a transformed program. Formally, there does not exist a scenario such that each processor p_i is at an instance $S_2^i\vec{\omega}_2^i$ and waiting for the assertion of $S_1^i\vec{\tau}_1^i$ by some processor.

Proof: By contradiction, assume that such a scenario exists. For each processor p_i, let $Tem(\vec{\omega}_2^i) = \vec{\tau}_2^i$. Select a processor p_{i_0}. It is at instance $S_2^{i_0}\vec{\omega}_2^{i_0}$ and waiting for the assertion of $S_1^{i_0}\vec{\tau}_1^{i_0}$ by processor p_{i_1}. From Lemma 4.13, we know that $S_1^{i_0}\vec{\tau}_1^{i_0} \succeq S_2^{i_1}\vec{\tau}_2^{i_1}$ or else processor p_{i_1} would have asserted $S_1^{i_0}\vec{\tau}_1^{i_0}$. From Lemma 4.10, we have $S_2^{i_1}\vec{\tau}_2^{i_1} \succ S_1^{i_1}\vec{\tau}_1^{i_1}$. By the transitive property of $\succ$, we have $S_1^{i_0}\vec{\tau}_1^{i_0} \succ S_1^{i_1}\vec{\tau}_1^{i_1}$. Continuing on as in Figure 4-19, processor p_{i_1} is waiting for some processor p_{i_2}, and we get $S_1^{i_1}\vec{\tau}_1^{i_1} \succ S_1^{i_2}\vec{\tau}_1^{i_2}$. Thus for each j, processor p_{i_j} is waiting for processor $p_{i_{j+1}}$ and $S_1^{i_j}\vec{\tau}_1^{i_j} \succ S_1^{i_{j+1}}\vec{\tau}_1^{i_{j+1}}$. Since the number of processors is finite, there exists j and k such that $j < k$ and $i_j = i_k$. By transitivity of $\prec$, we have $S_1^{i_j}\vec{\tau}_1^{i_j} \succ S_1^{i_k}\vec{\tau}_1^{i_k}$, which is a contradiction. □

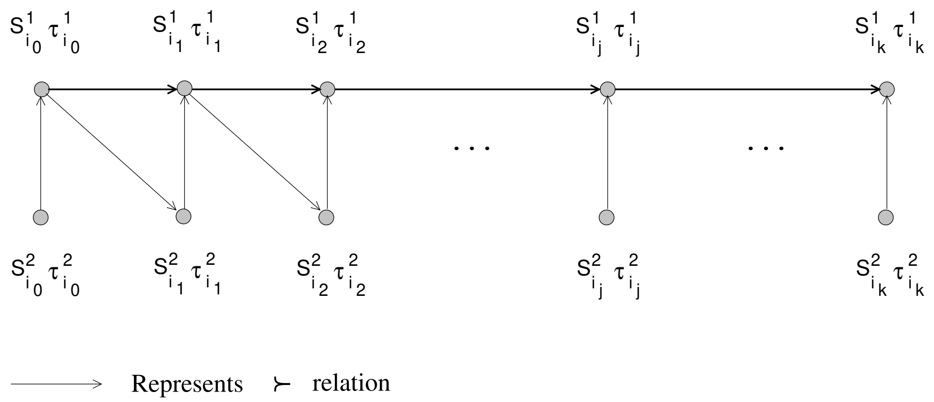


Figure 4-19: Deadlock scenario of Claim 4.14

In summary, the above proof implies that no cycles exist in the synchronization relationships among processors. This relies on ensuring two important criteria. First, processors must only wait for timestamps that are less than the current timestamp and thereby obey the temporal ordering on instances. Second, each instance must also assert synchronization to include other instances that are not related in the temporal ordering. Together, these constraints can be used derive a synchronization scheme that is free of deadlock conditions.

4.11 DOACROSS loops

The two loop constructs shown thus far in the thesis represent the purely sequential and purely parallel versions of loops. However, in some cases, one may wish for a loop construct that exhibits the behavior of both types. The `DOACROSS` loop construct commonly used in the literature [Cyt86] satisfies these characteristics. The semantics of `DOACROSS` loop execution follows that of sequential loops, but loop iterations can be partitioned among many processors. Consequently, data dependences can exist between iterations on different processors. Even though synchronization for `DOACROSS` loops has been studied by [MP87], a discussion is presented here to show how a synchronization scheme for such a loop fits into the current general framework that allows for arbitrary loop usage.

Semantically, iterations of a `DOACROSS` are executed in sequential order. Thus despite being partitioned into different processors, the processor executing an iteration must synchronize with the processor that executed the previous iteration. Within the execution model of loops defined in this chapter, one can satisfy the semantics by performing a barrier synchronization between each iteration of a `DOACROSS` loop. However, an actual implementation can depart somewhat from this expensive semantic specification. All iterations can be executed in parallel with point-to-point synchronization performed where dependences exist between iterations.

Whereas `DOALL` loop indices are viewed as temporal coordinates and `DO` loop indices as spatial coordinates, `DOACROSS` loop indices must be viewed as both temporal and spatial. Thus they affect the computation of both the processor and temporal target functions. Since `DOACROSS` loops are partitioned in the same manner as `DOALL` loops, source instance coordinates that correspond to `DOACROSS` loops are used in computing the processor target function. In addition `DOACROSS` instance coordinates need to be used to compute the temporal target function since the execution order of `DOACROSS` iterations on each processor must also be followed. Note that `DOACROSS` iterations are ordered similar to `DO` loop iterations, thus the ordering of timestamps remains unaffected.

At this point, one may object to the existence of so many different loop constructs in the language. Indeed, with an ideal compiler, there would be no need for separate specifications of sequential and parallel loops. All loops would be specified with the same construct, and the compiler would just optimally be able to partition the loop iterations onto processors. However, present-day compilers are far from ideal. The different loop constructs allow the programmer to give some hints to the compiler about dependence characteristics. In addition, this thesis takes the position that any dependence analysis for parallelism has been done in a previous phase. Hence, one can imagine the different constructs as information that has been deduced by the parallelization phase of a compiler.

4.12 Summary

Given the dependence relationships between statements computed in the previous chapter, we seek in this chapter to derive dependence information for processors. Unfortunately, the problem becomes very complex if one merely examines array indices of the dependent accesses. Instead, it is necessary to realize that dependences actually occur at program execution between particular invocations of statements. A statement instance can be defined as the lexical statement and a run-time context defined by the index values of all loops that enclose the statement. The above problem can then first be treated as one of finding dependence relationships between statement instances.

The task of relating the source and sink statement instance spaces can be solved by examining the source and sink array accesses. A dependence exists between two instances if the values of the array indices at those instances are equal. One can use each array index as a filter on the space of dependent statement instances. If nothing is known about an array reference, then no filtering is done. If enough instances can be filtered out, then point-to-point synchronization becomes realizable. Thus even though nothing may be known about some array indices, point-to-point synchronization can be used if enough instances have been filtered out by other indices.

When instance relationships are computed, one can then focus on deriving synchronization relationships between processors. By applying processor partitioning functions, one can make the transition between instances and processors. Likewise, sequential loop indices can be treated as timestamps to indicate temporal relationships. A requirement can then be imposed that synchronization must only be done with earlier instances to avoid cycles of synchronization. Even with this rule, deadlocks can still occur due to conditional execution. If a source statement is not executed, then a sink statement may be waiting indefinitely for the synchronization assertion. Thus one must transform a program to ensure that any control path contains a synchronization assertion. By following the above considerations, a provably deadlock-free synchronization scheme can be derived.

Chapter 5

Optimizations

5.1 Introduction

The algorithms presented in the previous chapter transform a program with barrier synchronization semantics into one that uses point-to-point synchronization wherever possible. However, since the predominant goal of this thesis involves producing improved performance over straightforward barrier synchronization schemes, optimizations must also be included in the transformation in order to increase efficiency. Whereas the previous chapter provided algorithms for general array references, this chapter focuses on providing optimizations for particular usage patterns. Even with such assumptions, the problems can be quite complex, and the task of integrating the optimizations into a general framework is a topic of further study.

We begin with a discussion of an alternate synchronization primitive, one that uses message-primitives rather than shared-memory accesses. The next section then focuses on eliminating dependences that are redundant due to compositions of other dependences. Finally, a novel technique for removing false dependences by replicating arrays is discussed.

5.2 Synchronization by message-passing

Since performing synchronization to support data dependences is most applicable in the shared-memory programming model, the point-to-point synchronization constructs presented in the previous chapter also make use of a shared-memory model. However, communication using a cache-coherent shared-memory model incurs significant overhead that can be alleviated by using explicit message-passing. Recall that synchronization is done in the shared-memory environment with the source processor setting a variable to some value and the sink processor spin-locking until the variable reaches a particular value. Because of cache support, spin-locking produces no network traffic and communication is done only after the source processor asserts the value, which causes an invalidation of the value in the cache of the sink processor.

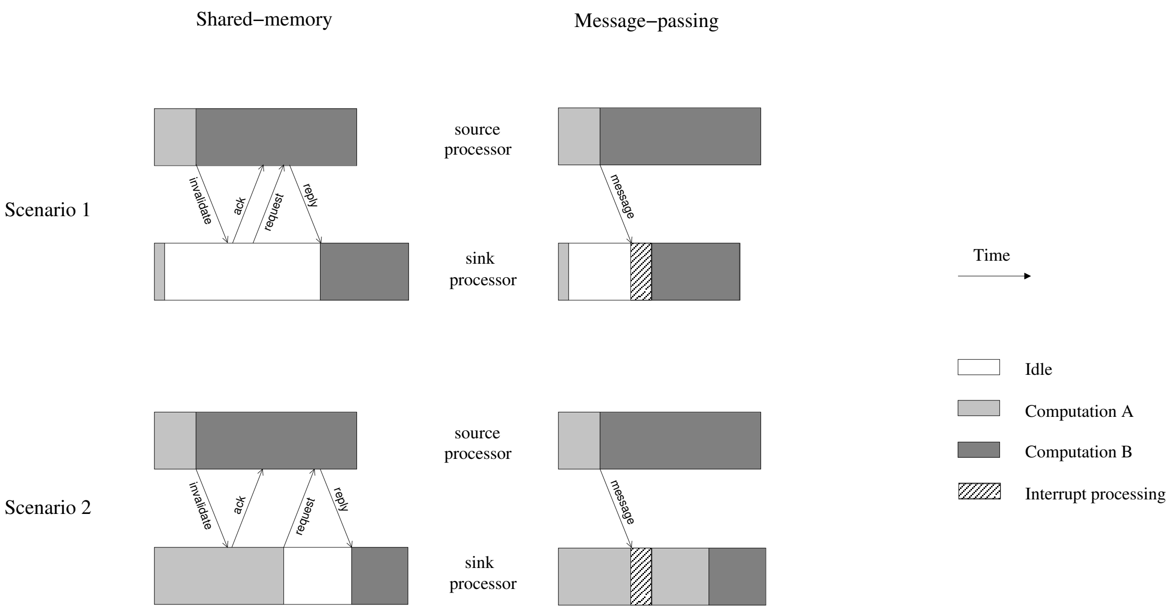


Figure 5-1: Message-passing vs. shared-memory

The run-time differences of the two models are illustrated in Figure 5-1. In scenario 1, the source processor asserts the synchronization after the sink processor begins checking for it. After the source asserts, four messages must be sent to update the caches before the sink processor sees the new value. Instead, synchronization through explicit sending of a message requires only the time for one message and additional overhead for message processing by the sink processor. Even when the source processor asserts much earlier than the sink begins checking as in scenario 2, using messages allows synchronization to be done with only the message-processing overhead rather than the request-reply round trip of the shared-memory paradigm. Figure 5-2 shows the difference between execution profiles of shared-memory and message-passing synchronization mechanisms. Note that the gaps representing idle synchronization intervals are smaller under the message passing scheme. However, the computation blocks also include extra time required for processing incoming messages.

The same disadvantages caused by the request-reply protocol of the shared-memory interface allows it to be more flexible than one-way message sends. In cases where processor synchronization targets are not known at compilation, one-way messages cannot be used as easily. For a particular synchronization, if the source processor is dependent on a run-time variable, the sink processor can determine the source processor at run time and then issue a memory request to check the synchronization variable. Ac-

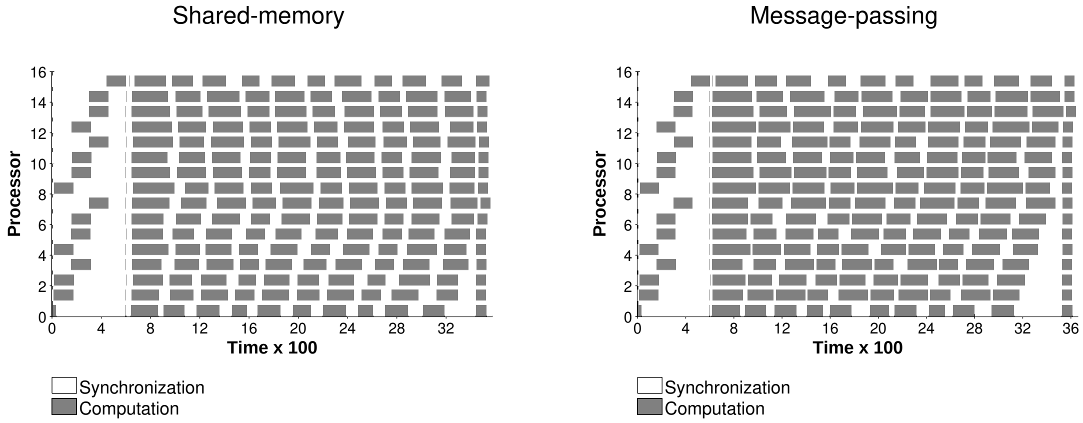


Figure 5-2: Execution of Figure 1-3 using different point-to-point synchronization schemes

complishing the same task using message-passing requires either the same request-reply scheme to be followed or computation by each potential source processor of whether it is the one that would have gotten the request. Although one can imagine cases where this computation can be done at reasonable cost, this section only focuses on implementing synchronization through message-passing when processor relationships can be statically determined.

In the shared-memory model, each synchronization is done through reading and writing to a variable that is shared by the source and sink processors. This same general technique can be supported in the message-passing model by maintaining a copy of the variable on the sink processor. To assert a synchronization, the source processor sends the new variable value to the sink processor. Upon the reception of each message, the sink processor updates its local variable to the new value stored in the message. To check for synchronization, the sink processor spins until its local variable reaches the desired value. These mechanisms assume a machine model where incoming messages are handled through processor interrupts. If messages must be explicitly received, then the sink processor merely spins until a message is received that contains the desired value. Note that the requirements for avoiding deadlocks in the shared-memory model also allow a message-passing scheme to be implemented without danger of deadlocks.

In the above scheme, sink processors are computed with respect to source processors as opposed to the relationship in the shared-memory model where source processors are computed with respect to sink processors. This is essentially equivalent to finding the inverse of the processor target function $\mathcal{P}_\Delta(p, \widehat{\mathrm{T}})$ of the last chapter. One can also imagine inverting the temporal target function $\mathcal{T}_\Delta(p, \widehat{\mathrm{T}})$ to compute the sink instance at which the synchronization is valid. Unfortunately, this inverse relationship does not completely generalize. As motivation, consider the case where the sink processor does not perform a synchronization in the shared-memory scheme. Even though the value asserted by the source processor is not read, the program still operates correctly. In the message-passing scenario, if the source processor does not send a message, but the sink processor requires one, then deadlock occurs. To be safe, the source processor must always send if there is a chance that the sink processor needs to check the result. Thus in cases where unknown expressions cause processor relationships to not be known, each source processor may be required to broadcast to all possible sink processors. Since these broadcasts may be very inefficient, the shared-memory interface provides a better solution in those cases.

Implementing synchronization through message-passing is only applicable to machines that provide support for both the shared-memory and message-passing models such as the MIT Alewife multiprocessor [Aga91]. On machines that only support message-passing, additional program transformations must be done to manage data sharing through explicit communication. Synchronization can be accomplished implicitly in such cases since processors are specifically aware of data sharing with other processors. Other shared-memory multiprocessors also contain mechanisms to overcome the inefficiencies of supporting cache-coherent protocols. The Stanford Dash multiprocessor [Len92] allows processors to write values directly to caches of other processors. Although it is less general, such a mechanism may support point-to-point synchronization even better than message-passing schemes since no message-processing overhead is required.

5.3 Redundant dependences

Whether synchronization is carried out through messages or shared-memory accesses, the execution of each synchronization primitive adds overhead to the total program running time. In many programs, not all synchronizations derived in the previous chapter need to be supported. An optimization phase can be included to remove redundant dependences and thereby minimize the number of synchronizations invoked at run time.

5.3.1 Motivation

Although dependences exist between individual instances, synchronizations are performed between processors. However, for simplicity, examples in the remainder of this chapter assume a one-to-one relationship between spatial coordinates and processors.

In the program of Figure 5-3, two flow dependences exist: $\Delta_1 = \texttt{S1}\langle i, j-1\rangle\ \delta^f\ \texttt{S2}\langle i, j\rangle$ and $\Delta_2 = \texttt{S1}\langle i-2, j-2\rangle\ \delta^f\ \texttt{S3}\langle i, j\rangle$. As illustrated, the gray arrows corresponding to dependence Δ_2 are redundant because they can be formed from the transitive closure of black arrows, which correspond to dependence Δ_1 and execution ordering of statements on each processor. In general, a dependence is redundant if it is automatically satisfied by the execution ordering that is implied by other dependences.

```
do (i=1,3) {
  doall (j=1,5)
    a[i,j] = ...;        /* S1 */
  doall (j=1,5)
    ... = a[i-2,j-2];   /* S2 */
  doall (j=1,5)
    ... = a[i,j-1];     /* S3 */
}
```

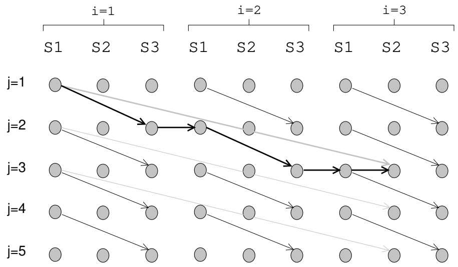


Figure 5-3: Redundant dependences

5.3.2 Problem definition

Formally, a dependence Δ between instances $S_1\vec{\omega}_0^1$ and $S_2\vec{\omega}_0^2$ is *redundant* if it is satisfied by a composition of other non-redundant dependences during any execution of a program. In other words, there exists a sequence of m non-redundant dependences $\{\Delta_i\}$ between instances $S_i^1\vec{\omega}_i^1$ and $S_i^2\vec{\omega}_i^2$ with the following properties:

$$\forall\, i\ \ S_{i+1}^1\vec{\omega}_{i+1}^1 \text{ is executed after } S_i^2\vec{\omega}_i^2 \text{ on each processor}$$
$$S_1^1\vec{\omega}_1^1 \text{ is executed after } S_1\vec{\omega}_0^1 \text{ on each processor}$$
$$S_m^2\vec{\omega}_m^2 \text{ is executed before } S_2\vec{\omega}_0^2 \text{ on each processor}$$

Note that the sequence of dependences that compose to cause Δ to be redundant must itself not include any redundant dependences. This condition is required when one considers two lone dependences that are identical. Since only one of those two dependences can be redundant, the determination of redundancy lies on the order of definition or algorithm traversal. In the algorithms that follow, dependences are checked in a well-defined order based on dependence distance. Dependences with identical distance are ordered arbitrarily depending on implementation.

Although dependences occur between instances of statements, this section focuses on removing redundant dependences between lexical statements. A *lexical dependence* between two statements represents the processor synchronization relationship for dependences between statement instances. Thus rather than focusing on dependences between individual instances, we instead study dependences between processors at particular statements. A more concrete definition of lexical dependences for particular situations will be given later. The word *lexical* will be omitted when the context is clear.

Unfortunately, removing all dependences that can be lexically redundant is an undecidable problem because it can require the knowledge of values that are only known at run time. As shown in Figure 5-4, the flow dependence between `S1` and `S2` is redundant only if the value of `x` is always 10 or greater. If `x` is computed from some undecidable function such as the halting problem, then establishing its value is also undecidable.

```
doall (j=1,100) b[j] = ...;      /* S1 */
do (i=1,x) {
  doall (j=1,100) a[j] = ...;
  doall (j=1,100) ... = a[j-1];
}
doall (j=1,100) ... = b[j-10];   /* S2 */
```

Figure 5-4

Furthermore, even in straight-line code without conditionals or sequential loops, the problem of finding redundant dependences is NP-hard, as shown in the following claim. As a side note, the problem is NP-complete since verification that a dependence is redundant can easily be done in polynomial time.†

Claim 5.1: Even with no sequential loops and conditionals, Finding redundant dependences is NP-hard.

Proof: The proof is based on reduction from the subset-sum problem: Given a set integers $U = \{u_1, \ldots, u_n\}$ and an integer b, the question of whether a subset $U' \subseteq U$ exists

† Midkiff and Padua [MP87] mention that finding redundant dependences in DOACROSS loops is NP-hard. Their proof is probably similar to the one given here.

such that $\sum_{u \in U'} u = b$ is NP-hard [GJ79][Kar72]. The program of Figure 5-5 can be created from the values of $\vec{y}$ and b where $m = \sum_i |u_i|$. Assume that the program is run on a machine with $3m + 1$ processors so that there is a one-to-one correspondence between loop iterations and processors. The problem then becomes one of whether the dependence Δ between statements S1 and S2 is redundant. If that is the case, then some composition of dependences of the ai arrays must have combined to satisfy Δ. Consequently, a solution exists to the subset-sum problem. Conversely, if no composition of dependences exist, then no solution exists to the subset-sum problem. □

```
doall (j=2m,3m) c[j] = ...;        /* S1 */

doall (j=m,4m)  a1[j] = ...;
doall (j=m,4m)  ... = a1[j-u1];
...
doall (j=m,4m)  an[j] = ...;
doall (j=m,4m)  ... = an[j-un];

doall (j=2m,3m) ... = c[j-b];    /* S2 */
```

Figure 5-5

Fortunately, the above problem is not NP-complete in the strong sense and can be computed in *pseudo-polynomial* time [GJ79]. Solutions exist for these problems whose running times are exponential in the length of the integers but polynomial in the value of the integers. The above subset-sum problem can be solved by a dynamic-programming algorithm that is polynomial in $\max(b, n, \log(\max u_i))$ [CLR90]. Furthermore, since the exponential growth in the example involves managing processor offsets, such quantities are limited by the number of processors on a machine. This leads one to be optimistic about the prospects of finding a polynomial-time algorithm to detect redundant dependences in straight-line code.

5.3.3 A solution for a simple problem domain

For simplicity, we first focus on programs **S** that take the form of a sequence $S_1, \ldots, S_n$ of non-nested DOALL statements. Data dependences between iterations in different DOALL loops give rise to dependences between processors assigned to those iterations. Since DOALL loops are not nested, the processor space can be viewed as a one-dimensional array. Thus synchronization relationships relative to a sink processor can be expressed as a linear function of the sink processor. In the following presentation, we focus on relationships that are purely integer offsets and relegate the treatment of general linear functions to a later discussion. The illustration of NP-hardness in Figure 5-5 represents a program with such assumptions.

Unfortunately, the subset-sum solution is not generally applicable to the above formulation due to the fact that address offsets can be positive as well as negative. Instead, the problem can be viewed as one of general integer linear programming which can also be solved in pseudo-polynomial time by dynamic programming [Sch86]. However, rather than computing whether each individual dependence is redundant, we seek to consolidate the intermediate dependence computations through an algorithm which computes redundancy information for all dependences at the same time.

With the above assumptions, a lexical dependence can be represented by the source and sink statements S_i and S_j and an integer offset d_Δ. A lexical dependence Δ is redundant under the following definition: For each sink processor p and source processor $p - d_\Delta$, where p executes S_j and $p - d_\Delta$ executes S_i, there are m non-redundant dependences $\{\Delta_k : 1 \leq k \leq m\}$ from source statements S_{i_k} to sink statements S_{j_k} with offsets d_k and processors p_k such that:

$$p_0 = p - d_\Delta \quad \text{and} \quad p_m = p \tag{5.1}$$

$$\text{Each processor } p_k \text{ executes } S_{j_k} \text{ and } S_{i_{k+1}} \tag{5.2}$$

$$\forall\, k \;\; p_{k-1} - p_k = d_k \tag{5.3}$$

$$\forall\, k \;\; j_k \leq i_{k+1} \tag{5.4}$$

$$i \leq i_1 \quad \text{and} \quad j_m \leq j \tag{5.5}$$

Due to some subtleties involving redundant dependences, an algorithm is first presented to find pseudo-redundant dependences, so-called because they possess only some characteristics of truly redundant dependences. Let the dependences between two statements S_i and S_j be represented by the function $\mathcal{D}(S_i, S_j)$. Each dependence $\Delta \in \mathcal{D}(S_i, S_j)$ is associated with an offset d_Δ which is computed with respect to the sink processors. A dependence Δ from S_i to S_j is *pseudo-redundant* if there are m non-pseudo-redundant dependences $\{\Delta_k \in \mathcal{D}(S_{i_k}, S_{j_k}) : 1 \leq k \leq m\}$ with offsets d_k such that the following are true:

$$\sum_k d_k = d_\Delta \tag{5.6}$$

$$\forall\, k \;\; j_k \leq i_{k+1} \tag{5.7}$$

$$i \leq i_1 \;\; \text{and} \;\; j_m \leq j \tag{5.8}$$

Since straight-line code implies that each $i_k < j_k$, a sequence of dependences that satisfy property (5.7) appear in lexical order in a program and is called a *cascade* of dependences. Observe that true redundancy implies pseudo-redundancy since the definition of pseudo-redundancy is identical to that of true redundancy without rules (5.1) and (5.2).

In the following presentation, the value $\mathcal{R}(S_i, S_j)$ represents the set of processor offsets whose dependences are satisfied by cascades of dependences involving statements S_i through S_j. The table represented by $\mathcal{R}(S_i, S_j)$ forms the basic update structure of the dynamic-programming algorithm. At each step j, the algorithm in Figure 5-6 computes $\mathcal{R}(S_i, S_j)$ for each source statement S_i. The value of $\mathcal{R}(S_i, S_j)$ can be derived from the previous value of $\mathcal{R}(S_i, S_{j-1})$ and any new dependences that include S_j as the sink statement.

Algorithm *delRedun1* $(S, \mathcal{D})$:

    Initialize all $\mathcal{R}(S_i, S_j)$ to $\{0\}$.

    for i from 1 to n do
        for j from $i+1$ to n do
            $\mathcal{R}(S_i, S_j) = \mathcal{R}(S_i, S_{j-1})$
            for each dependence Δ from S_k to S_j such that $k > i$ do
                for each $d' \in \mathcal{R}(S_i, S_k)$ do
                    $\mathcal{R}(S_i, S_j) = \mathcal{R}(S_i, S_j) \cup \{d' + d_\Delta\}$
            for each dependence Δ in $\mathcal{D}(S_i, S_j)$ do
                if $d_\Delta \in \mathcal{R}(S_i, S_j)$ then
                    Δ is a pseudo-redundant dependence
                $\mathcal{R}(S_i, S_j) = \mathcal{R}(S_i, S_j) \cup \{d_\Delta\}$

Figure 5-6: Finding pseudo-redundant dependences in straight-line code

Let d^+ and d^- be maximum and minimum processor offsets and define the offset size as $s = d^+ - d^- + 1$. The above algorithm requires n^2 steps for the outer two loops, b steps for the inner loop where b is the maximum number of dependences to any vertex, and s steps for updating $\mathcal{R}(S_i, S_j)$. The total running time is thus $O(n^2bs)$, which is near $O(n^3)$ if one assumes that s is constant and that b scales as n.

The following claims show that the above algorithm deletes exactly those lexical dependences that are pseudo-redundant. Since the algorithm follows an iterative structure, the proofs make heavy use of induction and related techniques to show correctness.

Lemma 5.2: $\mathcal{R}(S_i, S_j) \subseteq \mathcal{R}(S_{i'}, S_{j'})$ if $i' \leq i$ and $j' \geq j$.
Proof: For $i' = i$ and $j' \geq j$, the above is trivial since each computation of $\mathcal{R}(S_i, S_j)$ begins with the value of $\mathcal{R}(S_i, S_{j-1})$. If $i' \leq i$ and $j' = j$, the statement can be proven by induction on j: For $i = j$, the lemma is true since $\mathcal{R}(S_i, S_i) = \{0\}$. The inductive step is also straightforward since any sets unioned to $\mathcal{R}(S_i, S_j)$ are also unioned to $\mathcal{R}(S_{i'}, S_j)$. The general case can be shown by applying both of the above arguments. □

Claim 5.3: A dependence Δ is detected by *delRedun1* $\iff$ Δ is pseudo-redundant.
Proof: The proof is done for each direction of the claim individually.

($\Leftarrow$) We first claim that each cascade of dependences $\{\Delta_1, \ldots, \Delta_m\}$ is represented by the set of processor offsets $\mathcal{R}(S_{i_1}, S_{j_m})$, or equivalently, $\sum_{1 \leq j \leq m} d_j \in \mathcal{R}(S_{i_1}, S_{j_m})$. By contradiction, assume that there are cascades that are not represented by $\mathcal{R}(S_{i_1}, S_{j_m})$. Let $\{\Delta_1, \ldots, \Delta_m\}$ be the shortest cascade that is not represented. If $m = 1$, then a contradiction arises since $d_1 \in \mathcal{R}(S_{i_1}, S_{j_1})$. If $m > 1$, then the cascade $\{\Delta_1, \ldots, \Delta_{m-1}\}$ is represented and $\sum_{1 \leq j \leq m-1} d_j \in \mathcal{R}(S_{i_1}, S_{j_{m-1}})$ which also implies that $\sum_{1 \leq j \leq m-1} d_j \in \mathcal{R}(S_{i_1}, S_{i_m})$ by the lemma. However, at algorithm step $i = i_1$ and $j = j_m$, the dependence Δ_m is found for $k = i_m$. Therefore $d_m + \sum_{1 \leq j \leq m-1} d_j \in \mathcal{R}(S_{i_1}, S_{j_m})$. Also by the above lemma, any cascade of dependences is thus found by the algorithm and all redundant dependences are removed.

($\Rightarrow$) We need to show that each offset in $\mathcal{R}(S_i, S_j)$ corresponds to a cascade of dependences between S_i and S_j. By contradiction, assume otherwise. Let i and j be the respective loop values for the first violation of the above. Then the violation must have happened when considering dependences from S_k to S_j. However, since this the first violation, we know that each $\mathcal{R}(S_i, S_k)$ is correct and consequently that the resulting $\mathcal{R}(S_i, S_j)$ computation is correct, which leads to a contradiction. □

One can also imagine a different algorithm for removing redundant dependences which views statements as nodes in a graph and dependences as edges in the graph with weights equal to sets of offset values. A transitive closure can be formed by applying the Floyd-Warshall algorithm to the graph, which results in a running time of $O(n^3 s)$. However, as additional language constructs are considered, the more direct treatment of program structure in *delRedun1* allows easier incorporation of these constructs.

```
doall (i=1,5)
  a[i] = ...;       /* S1 */
doall (i=1,5)
  b[i] = ...;       /* S2 */
doall (i=1,5)
  ... = b[i-3];     /* S3 */
doall (i=1,5)
  c[i] = ...;       /* S4 */
doall (i=1,5)
  ... = c[i+2];     /* S5 */
doall (i=1,5)
  ... = a[i-1];     /* S6 */
```

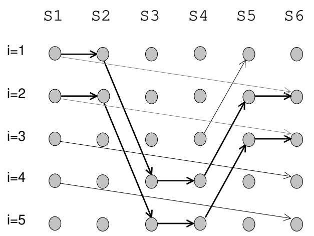


Figure 5-7: Redundant dependences and processor bounds

The above definition of pseudo-redundant dependences cannot be used to define redundant dependences since processor bounds have been ignored. In the program of Figure 5-7, the lexical dependence from `S1` to `S6` consists of four actual dependences between instances. However, only two of those dependences (drawn in gray) are redundant since cascades that would make the other dependences redundant are outside of the processor bounds of the loops. Consequently, the lexical dependence is not redundant. To accurately compute these cases, we associate with each processor offset d in $\mathcal{R}(S_i, S_j)$ a range of sink processors $\mathcal{E}(d, S_i, S_j)$ for which the offset d is effective. The range specified by $\mathcal{E}(d, S_i, S_j)$ cannot be outside of the processor bounds of the machine. For each dependence Δ, we introduce the notation $\mathcal{B}(\Delta)$ to represent sink processors affected by the dependence. Its value can be computed from the processor range of the sink statement intersected with the processor range of the source statement minus the dependence offset d_Δ. When a dependence Δ is added to a cascade to form a new offset, the new range of sink processors for the cascade is formed from the old range minus d_Δ and intersected with the processor range for the dependence. A pseudo-redundant dependence Δ is redundant only if its sink processor range $\mathcal{B}(\Delta)$ is within the processor range of the cascade. A new algorithm which incorporates the above computations is shown in Figure 5-8. Changes from *delRedun1* are denoted by the symbol "$\surd$". The running time of the algorithm is the same as that of *delRedun1* if processor ranges are specified efficiently.

Algorithm *delRedun2*$(S, \mathcal{D}, \mathcal{B})$:

Initialize all $\mathcal{R}(S_i, S_j)$ to $\{0\}$.
Initialize all $\mathcal{E}(d, S_i, S_j)$ to $\emptyset$. $\surd$
Initialize all $\mathcal{E}(0, S_i, S_j)$ to all processors. $\surd$

for i from 1 to n do
  for j from $i+1$ to n do
    $\mathcal{R}(S_i, S_j) = \mathcal{R}(S_i, S_{j-1})$
    for each dependence Δ from S_k to S_j such that $k > i$ do
      for each $d' \in \mathcal{R}(S_i, S_k)$ do
        $\mathcal{R}(S_i, S_j) = \mathcal{R}(S_i, S_j) \cup \{d' + d_\Delta\}$
        $\mathcal{E}(d' + d_\Delta, S_i, S_j) = \mathcal{E}(d' + d_\Delta, S_i, S_j) \cup [(\mathcal{E}(d', S_i, S_k) - d_\Delta) \cap \mathcal{B}(\Delta)]$ $\surd$
    for each dependence Δ in $\mathcal{D}(S_i, S_j)$ do
      if $d_\Delta \in \mathcal{R}(S_i, S_j)$ and $\mathcal{B}(\Delta) \subseteq \mathcal{E}(d_\Delta, S_i, S_j)$ then $\surd$
        delete the redundant dependence Δ
      $\mathcal{R}(S_i, S_j) = \mathcal{R}(S_i, S_j) \cup \{d_\Delta\}$
      $\mathcal{E}(d_\Delta, S_i, S_j) = \mathcal{E}(d_\Delta, S_i, S_j) \cup \mathcal{B}(\Delta)$ $\surd$

Figure 5-8: Deleting redundant dependences

Claim 5.4: A lexical dependence Δ is removed by *delRedun2* $\iff$ Δ is redundant.
Proof: The proof is done for each direction of the claim individually. Let $S_{i'}$ and $S_{j'}$ be the source and sink statements.

($\Leftarrow$) Since redundancy implies pseudo-redundancy, Δ is detected by *delRedun1*. Based on the observation that the computation of $\mathcal{R}(S_i, S_j)$ is identical in both algorithms, we only need to show the following: For any processor p such that p executes $S_{i'}$ and $p - d_\Delta$ executes $S_{j'}$ and there exists m dependences Δ_k and processors p_k such that rules (5.1) through (5.5) hold, then $p - d_\Delta \in \mathcal{E}(d_\Delta, S_{i'}, S_{j'})$. The above can be proven by showing that each $p_k \in \mathcal{E}(\sum_{1 \le k' \le k} d_{k'}, S_{i'}, S_{j_k})$ by induction on k. For each k, we know that $p_k \in \mathcal{B}(\Delta_k)$ otherwise p_k can't execute S_{j_k}. For $k = 1$, we know that at algorithm loop iteration $i = i'$ and $j = j_1$, dependence Δ_1 is considered and $p_1 \in \mathcal{E}(d_1, S_{i'}, S_{j_1})$ since $p_1 = p - d_1$. Inductively, for loop values $i = i'$ and $j = j_k$, dependence Δ_k is considered and $p_k \in \mathcal{E}(\sum_{1\le k'\le k} d_{k'}, S_{i'}, S_{j_k})$ since $p_k = p - d_k - \sum_{1\le k'\le k-1} d_{k'}$ and $p - \sum_{1\le k'\le k-1} d_{k'} \in \mathcal{E}(\sum_{1\le k'\le k-1} d_{k'}, S_{i'}, S_{j_{k-1}})$ by induction. Therefore, $p - d_\Delta \in \mathcal{E}(d_\Delta, S_{i'}, S_{j'})$ and Δ is redundant.

($\Rightarrow$) Suppose by contradiction that Δ is not redundant. Note that Δ must be pseudo-redundant since *delRedun2* only removes a subset of dependences removed by *delRedun1*. Thus there exists a source processor p such that p executes $S_{i'}$ and $p - d_\Delta$ executes $S_{j'}$ but there does not exist dependences Δ_k and processors p_k such that rules (5.1) and (5.2) hold. Since rule (5.1) is trivially satisfiable, it must be (5.2) that does not hold. Thus for dependences Δ_k and processors p_k that satisfy all the other rules, there exists some k' such that processor $p_{k'}$ does not exist or cannot execute $S_{j_{k'}}$ or $S_{i_{k'+1}}$. Let k be the smallest number such that the above is true. Then at algorithm loops $i = i'$ and $j = j_k$, dependence Δ_k is considered. There are three cases:
(a) If p_k does not exist, then it cannot possibly be in $\mathcal{E}(\sum_{1\le k'\le k} d_{k'}, S_{i'}, S_{j_k})$.
(b) If p_k cannot execute S_{j_k}, then $p_k \notin \mathcal{B}(\Delta_k)$.
(c) If p_k cannot execute $S_{i_{k+1}}$, then $p_{k+1} = p_k - d_{k+1} \notin \mathcal{B}(\Delta_{k+1})$ and is detected in the next iteration.

In all three cases, the dependence is not deleted. □

5.3.4 General removal of redundant dependences

Support for additional language features can be presented in order of complexity of modifications to the algorithm. First, we consider supporting synchronization relationships that are general linear functions of the sink processor address. Rather than merely adding offsets to compose the effects of two dependences, the linear functions themselves must be composed, with certain restrictions. Since the function domains involve integers, the composition of functions is not straightforward. For example, the composition of the functions $2p$ and $\lfloor \frac{1}{2}p \rfloor$ does not return p, but rather $2\lfloor \frac{1}{2}p \rfloor$. The task of managing these linear functions and deducing their inclusion relations can become expensive, and one may be forced to ask if these cases arise often enough in a program to justify the cost. In this thesis, the focus is on processor relationships that are integer offsets and absolute source processor addresses. The latter represents cases where all processors synchronize to one processor such as in a data broadcast, and can be supported by a straightforward extension to *delRedun1* to allow for absolute processor values as well as offsets in $\mathcal{R}(S_i, S_j)$. A dependence that requires barrier synchronization can be viewed as a synchronization with all absolute processor addresses.

When DOALL loops are allowed to be nested, the processor space must be viewed as being multi-dimensional rather than one-dimensional. In this context, dimensions refer exactly to processor partitioning sets in the previous chapter. When considering synchronization relationships between sets of loop nests with similar processor partitionings, processor relationships specify a linear function on each sink dimension as well as the source dimension to which the linear function maps. If linear functions are restricted to be address offsets or absolute addresses as above, then composition and inclusion of processor relationships can still be computed efficiently. Unfortunately, when different loop nests have different processor partitions or different numbers of nested DOALL loops, then finding all redundant dependences is too inefficient. Instead, a heuristic can be used which treats the partition space of each set of nested loops separately without regard for processor relationships that are not explicitly specified by the partition functions. For example, when relating a processor space that is one-dimensional to rows in a two-dimensional space, each row is analyzed separately without consideration for the fact that processors in the one-dimensional space correspond to many rows in the two-dimensional space.

Up to this point, the program structure has been assumed to be a sequence of DOALL loops. Now we remove this assumption and consider other control flow constructs. The presence of sequential loops extend the program flow graph to contain back edges as well as forward edges. Consequently, any scheme to detect redundant dependences must allow for the search path to traverse over the same node many times. In addition, temporal synchronization relationships must now be taken into account, as shown in Figure 5-9. For simplicity, we consider only the flow dependences in the example. Although the dependence on variable a from S1 to S2 is not redundant using forward edges only, it is redundant when one uses the back edges from S3 to S1. Of course, this is only possible because the dependence spans two iterations of the sequential loop, as would be specified by the temporal target function. In the following discussion, we assume that a reasonable lower bound can be established on the number of iterations executed in any sequential loop. We also assume temporarily that sequential loops are not nested.

Since the program flow graph becomes cyclic with the addition of back edges, schemes to detect redundant dependences must now be able to guarantee termination.

```
do (i=1,100) {
  doall (j=1,7)
    a[i][j] = f(c[i-1][j-2]);  /* S1 */
  doall (j=1,7)
    b[i][j] = g(a[i-2][j-5]);  /* S2 */
  doall (j=1,7)
    c[i][j] = h(b[i-1][j-3]);  /* S3 */
}
```

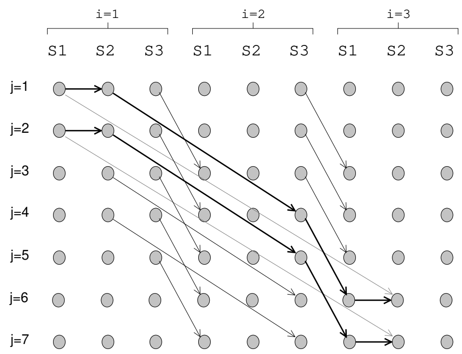


Figure 5-9: Redundant dependences and sequential loops

One solution can be to limit the number of back edges traversed to be equal to the lower bound of the number of loop iterations. However, this number can be very large in many programs, and the running time of an $O(n^3)$ algorithm where n includes the number of sequential loop iterations can cause programmers to turn off optimizations altogether. A second option involves fixing the number of back edges that the algorithm can traverse and give up on the goal of finding all redundant dependences. Fortunately, it is not always necessary to traverse such a large number of back edges to find all redundant dependences due to the fact that synchronization relationships typically span only a few sequential loop iterations, as shown in Figure 5-9. Since processor offsets correspond to processor synchronization targets, we introduce the notion of temporal offsets to represent temporal synchronization targets. A temporal offset t_Δ of a dependence Δ indicates the number of iterations of the sequential loop between the dependent sink and source instances. For a particular dependence, its temporal offset specifies the maximum number of back edges that one needs to traverse to decide whether the dependence is redundant.

The above idea of using temporal offsets is applicable only to dependences within a sequential loop. When a dependence Δ spans across a sequential loop as in Figure 5-5, it may still be necessary to traverse a large number of back edges. However, temporal offsets of dependences inside the loop can also be used to place an upper limit on the number of iterations needed to make Δ redundant. Let $d_1, \ldots, d_m$ be the processor offsets and $t_1, \ldots, t_m$ be the temporal offsets for dependences in the loop. The redundancy problem can be stated as the following integer linear programming problem:

$$\text{Find } \vec{x} \text{ to minimize } \{\vec{t} \cdot \vec{x} : x_i \geq 0 \text{ and } \vec{d} \cdot \vec{x} = d_\Delta\}$$

where $\vec{x}$ represents the number of times that each dependence is "used" in forming a cascade with processor offset d_Δ. From [Sch86], each component of $\vec{x}$ is bound by ms where s is the maximum absolute value of d_i and d_Δ. Consequently, one only needs to traverse $m^2 s$ back edges to find all redundant dependences even with very large sequential loop bounds.

A dynamic programming algorithm for finding pseudo-redundant dependences is shown in Figure 5-10. The offsets resulting from cascades of dependences $\mathcal{R}(S_{i'}, S_i, \ell)$ now include a third dimension to represent the number of back edges that have been traversed. The limit of back edges L can either be set to a small constant or to the maximum value of $m^2 s$ and temporal offsets for all loops to find all redundant dependences. Note that if $L = 1$, then we recover the algorithm of Figure 5-6. Any dependence Δ whose source and sink statements are outside of a loop is given temporal offset $t_\Delta = \infty$. The outer loop iterates over the number of back edges that a cascade can possess. New cascades are formed from current dependences combined with older cascades. These combinations take into account the temporal offsets of each dependence and uses cascades of the appropriate iteration. Recalling that n is the number of statements in a program, the running time of this algorithm is near $O(Ln^3)$.

Algorithm $\mathit{delRedun3}(S, \mathcal{D})$:

Initialize all $\mathcal{R}(S_1, S_2, \ell)$ to $\{0\}$.

for ℓ from 1 to L
  for i from 1 to n do
    for j from 1 to n do
      $\mathcal{R}(S_i, S_j, \ell) = \mathcal{R}(S_i, S_{j-1}, \ell) \cup \mathcal{R}(S_i, S_j, \ell - 1)$
      if a back edge exists from S_h to S_j then
        $\mathcal{R}(S_i, S_j, \ell) = \mathcal{R}(S_i, S_j, \ell) \cup \mathcal{R}(S_i, S_h, \ell - 1)$
      for each dependence Δ from S_k to S_j do
        for each $d' \in \mathcal{R}(S_i, S_k, \ell - t_\Delta)$ do
          $\mathcal{R}(S_i, S_j, \ell) = \mathcal{R}(S_i, S_j, \ell) \cup \{d' + d_\Delta\}$
      for each dependence Δ in $\mathcal{D}(S_i, S_j)$ do
        if $d_\Delta \in \mathcal{R}(S_i, S_j, \ell)$ and $t_\Delta \geq \ell$ then
          Δ is a pseudo-redundant dependence
        $\mathcal{R}(S_i, S_j, \ell) = \mathcal{R}(S_i, S_j, \ell) \cup \{d_\Delta\}$

Figure 5-10: Finding pseudo-redundant dependences with sequential loops

When sequential loops are nested, the algorithm must be modified to represent temporal offsets as tuples rather than integers. If L is the limit of back edges that one can traverse for each loop, then the outer loop contains ϵL iterations where ϵ is the maximum sequential loop nesting level. Temporal dependence distances contain tuples whose length is determined by the number of outer sequential loops of the source and sink statements. The running total of dependence offsets $\mathcal{R}(S_i, S_j, \ell)$ is extended to allow for ϵ additional dimensions, one for each loop nesting. The running time of such an algorithm is thus $O(\epsilon L n^3)$.

Unlike sequential loops with a lower bound on iterations, conditionals in a program imply that there are some statements that may not be executed by any processor. Consequently, an algorithm for removing redundant dependences in programs with conditionals must pay more attention to program flow. Since the number of paths between a source and sink statement can potentially be exponential in program length, intermediate information must be somehow gathered at join points for later phases of the algorithm. Although a polynomial-time algorithm can be given to remove all redundant dependences in programs with conditionals, we instead recommend an approach based on program structure as given in the next section.

5.3.5 Redundant dependences in structured programs

The previous section presented algorithms with the goal of eliminating all redundant lexical dependences in a program. Unfortunately, the $O(n^3)$ running times of such approaches can result in very slow compiler execution, particularly for large procedures where n approaches 1000 or more statements. Instead, the problem can be alleviated by applying algorithms that do not remove all dependences, but possess the potential of being more efficient.

Consider for example the problem of removing redundant dependences in the presence of conditionals. As mentioned above, complex algorithms can be used to summarize information at join points and detect all redundant dependences. However, one can also take the view that the source and sink statements of a dependence usually appear at the same lexical level in a program. By focusing on such dependences, more intuitive algorithms can be developed. With each statement S, we associate a list of processor offsets $\mathcal{F}(S)$ that are satisfied by the statement. Redundant dependences are computed and detected recursively in a bottom-up manner. In the case of a conditional

S = `if` (P) S_1 `else` S_2, one can check for redundant dependences on each branch of the conditional individually. The resulting list of offsets for the conditional can be defined as the intersection of processor offsets for each branch: $\mathcal{F}(S) = \mathcal{F}(S_1) \cap \mathcal{F}(S_2)$. Although this scheme does not account for redundant dependences such that between `S1` and `S4` as illustrated in Figure 5-11, it does exhibit a more modular structure than the ones given previously.

```
doall (i=1,100) a[i] = ...;        /* S1 */
if (p) {
  doall (i=1,100) ... = a[i-2];    /* S2 */
  doall (i=1,100) ... = a[i-3];    /* S3 */
  doall (i=1,100) ... = a[i-5];    /* S4 */
}
```

Figure 5-11

One can begin the specification of the recursive algorithm by observing that each sequence of statements can be analyzed as in *delRedun1* and *delRedun2*. An additional feature must be added to these algorithms to allow for the fact that statements themselves can contain processor offsets. Thus each $\mathcal{R}(S, S)$ is be initialized to $\mathcal{F}(S)$ rather than just $\{0\}$. The resulting processor offsets is then the processor offsets of the first and last statements in the sequence. For sequential loops, we can use the same strategy and obtain processor offsets for a certain number of iterations of the loop. Such a recursive algorithm is outlined in Figure 5-12. Although the order of growth in running time is not larger than the previous algorithms, the value of n can be much smaller since the dynamic programming is only applied to statements at the same lexical level rather than all statements in a program. Note that some details are omitted, particularly in the interface with previous algorithms. However, such modifications are straightforward if one is aware of the spirit of the above algorithms.

5.4 Eliminating false dependences

Although one must provide synchronization for all dependences that arise in a program, it is also useful to examine whether all such dependences are indeed necessary. Flow dependences represent actual transaction of information from the writing processor to the reading processor and consequently cannot be eliminated easily. However, output and anti-dependences are false dependences in the sense that they occur only

Algorithm $\mathit{delRedun4}\,(S, \mathcal{D})$:

For different cases of statement S:

$S = [\![V = E]\!]$
    return $\{0\}$
$S = [\![$`if` (P) S_1 `else` $S_2]\!]$
    $t_1 = \mathit{delRedun4}\,(S_1, \mathcal{D})$
    $t_2 = \mathit{delRedun4}\,(S_2, \mathcal{D})$
    return $t_1 \cap t_2$
$S = [\![$`while` (P) $S']\!]$
    $\mathit{delRedun4}\,(S', \mathcal{D})$
    return $\{0\}$
$S = [\![$`do (`V`=`K`,`K`,`K`)` $S']\!]$
    $\mathit{delRedun3}\,(S', \mathcal{D})$
    return result for highest ℓ
$S = [\![$`doall (`V`=`K`,`K`,`K`)` $S']\!]$
    return $\mathit{delRedun4}\,(S', \mathcal{D})$
$S = [\![$`{`S_1`,...,`S_n `}`$]\!]$
    $\mathit{delRedun1}\,(\{S_1, \ldots, S_n\}, \mathcal{D})$
    return $\mathcal{R}(S_1, S_n)$

Figure 5-12: Finding pseudo-redundant dependences recursively

because memory locations are being overwritten. In a single-assignment model, these dependences do not exist. Several works in the literature have introduced optimizations to remove such dependences by replicating arrays for every processor or loop iteration [Fea88][MAL93][Kum87]. While these techniques produce good results for the goal of parallelization, their application to the goal of reducing synchronization is not completely appropriate. First, we review the motivation for eliminating anti-dependences with an example.

In Chapter 1, an example is shown where anti-dependences can be eliminated by making two versions of an array. The example given here requires that an array be replicated into three copies before anti-dependences can be eliminated. Consider the

```
do (i=1,100) {
  doall (j=1,1000)
    b[j] = a[j];                  /* S1 */
  doall (j=1,1000)
    a[j] = (b[j-2]+b[j+1])*.5; /* S2 */
}
```

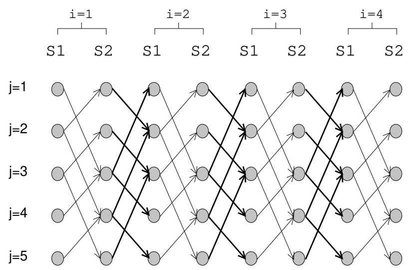


Figure 5-13: A candidate for anti-dependence elimination

program in Figure 5-13. The following flow dependences exist between S1 and S2:

$$\texttt{S1}\langle\texttt{i, j-2}\rangle\ \delta^f\ \texttt{S2}\langle\texttt{i, j}\rangle$$
$$\texttt{S1}\langle\texttt{i, j+1}\rangle\ \delta^f\ \texttt{S2}\langle\texttt{i, j}\rangle$$

In a sequential loop with index i, if a flow dependence occurs between two instances $S_1\langle i, j_1 + d_1, \ldots, j_n + d_n\rangle$ and $S_2\langle i, j_1, \ldots, j_n\rangle$ and j_k are all indices of DOALL loops inside the sequential loop and S_1 and S_2 operate on the same set of array elements, then an anti-dependence exists between $S_2\langle i-1, j_1 - d_1, \ldots, j_n - d_n\rangle$ and $S_1\langle i, j_1, \ldots, j_n\rangle$. This observation arises from the fact that both flow and anti-dependences are due to a write access and a read access. If a write must appear before a read in one iteration, then the read of the next iteration must appear after the write. In the example, the following anti-dependences exist between S2 and S1 and are highlighted in the illustration:

$$\texttt{S2}\langle\texttt{i, j+2}\rangle\ \overline{\delta}\ \texttt{S1}\langle\texttt{i, j}\rangle$$
$$\texttt{S2}\langle\texttt{i, j-1}\rangle\ \overline{\delta}\ \texttt{S1}\langle\texttt{i, j}\rangle$$

```
do (i=1,100) {
  k = i mod R;
  doall (j=1,1000)
    b[k][j] = a[j];                       /* S1 */
  doall (j=1,1000)
    a[j] = (b[k][j-2]+b[k][j+1])*.5; /* S2 */
}
```

Figure 5-14

Since all dependences involving array a are trivially satisfied, we focus on replicating array b. The program of Figure 5-14 shows a modification of the previous example to replicate array `b` into R copies. By increasing the number of copies of the array, the temporal distance of each anti-dependence is also increased. As illustrated in Figure 5-15, a replication factor of $R = 2$ does not result in any redundant dependences, but a replication factor of $R = 3$ causes all anti-dependences between `S2` and `S1` to be redundant. Thus by maintaining three copies of the array `b`, we have eliminated all synchronization requirements between the execution of statement `S2` and that of statement `S1`. The only remaining dependences that need to be supported are those flow dependences between `S1` and `S2`.

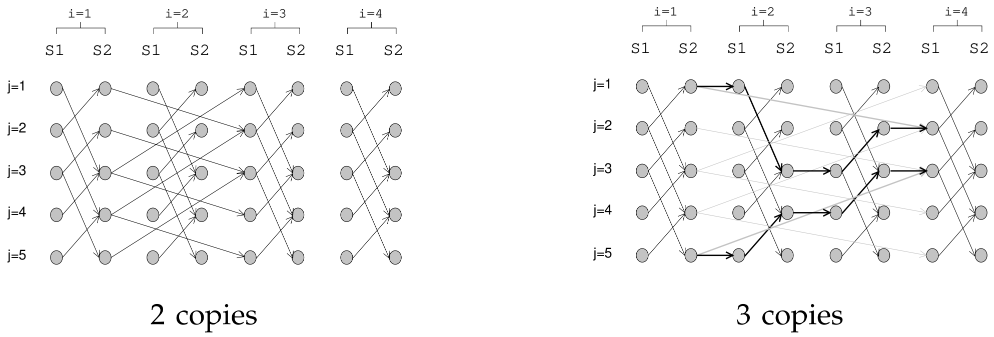


Figure 5-15: Dependences from replication of array `b`

From the above example, we see that the replication strategy makes use of shorter-distance dependences to eliminate anti-dependences with only a small number of array replications. This usage forms the primary difference between the elimination of false dependences to reduce synchronization and such elimination to increase parallelism. When one performs array replication for parallelism, the effort is only worthwhile if there are no other dependences across loop iterations. If that criterion is met, then an array can be "privatized" by being replicated across all processors, and all iterations of the loop can be executed in parallel. Instead, in the context of synchronization, one has no desire to try to execute the outer sequential loop in parallel due to the existence of flow dependences across iterations. However, it is still advantageous to try to eliminate false dependences in order to reduce synchronization overhead and to allow less restricted execution of iterations. When anti-dependences appear across iterations of a sequential loop, one can make use of the flow dependences that also arise to reduce the replication factor.

Note that although the above discussion focuses on dependences across instances, one can also apply the observations to dependences between processors. The dependence distances measured in `DOALL` loop iterations can just as easily be represented by distances between processor partitions. The algorithms for finding redundant dependences with sequential loops can then be adapted to find the minimum array replication factor needed to eliminate anti-dependences.

When detecting redundant dependences in sequential loops, cascades are formed from all dependences. If a dependence is contained a cascade and its temporal offset spans a range greater than the cascade, then it is redundant. This scheme can be altered to suit the current task by initially only considering flow dependences. Since the temporal offset of anti-dependences are dependent on the amount of replication, anti-dependences are initially checked for containment in cascades without consideration of temporal offsets. If an anti-dependence can be redundant, then its replication factor is the increase in temporal offset needed to make the anti-dependence redundant. An algorithm is shown in Figure 5-16 for pseudo-redundant elimination. A correct implementation must also consider processor bounds as in *delRedun2*. The symbol "$\surd$" is used to denote differences from *delRedun3*.

Algorithm $delAnti(S, \mathcal{D})$:

Initialize all $\mathcal{R}(S_1, S_2, \ell)$ to $\{0\}$.

for ℓ from 1 to L
  for i from 1 to n do
    for j from 1 to n do
      $\mathcal{R}(S_i, S_j, \ell) = \mathcal{R}(S_i, S_{j-1}, \ell) \cup \mathcal{R}(S_i, S_j, \ell - 1)$
      if a back edge exists from S_h to S_j then
        $\mathcal{R}(S_i, S_j, \ell) = \mathcal{R}(S_i, S_j, \ell) \cup \mathcal{R}(S_i, S_h, \ell - 1)$
      for each flow dependence Δ from S_k to S_j do $\surd$
        for each $d' \in \mathcal{R}(S_i, S_k, \ell - t_\Delta)$ do
          $\mathcal{R}(S_i, S_j, \ell) = \mathcal{R}(S_i, S_j, \ell) \cup \{d' + d_\Delta\}$
      for each anti-dependence Δ in $\mathcal{D}(S_i, S_j)$ do $\surd$
        if $d_\Delta \in \mathcal{R}(S_i, S_j, \ell)$ and Δ involves array a then $\surd$
          Δ is redundant if a is replicated by $\ell - t_\Delta$ $\surd$
        $\mathcal{R}(S_i, S_j, \ell) = \mathcal{R}(S_i, S_j, \ell) \cup \{d_\Delta\}$

Figure 5-16: Eliminating pseudo-redundant anti-dependences

The total replication factor R of an array a is equal to the sum of replications for each anti-dependence in a loop plus one for the current array copy. The loop can then be transformed to support the replication. A new array a' is formed from a with an additional dimension to allow for the use of a replication index. For each assignment to a' in the loop, the index is incremented by one modulo R. One must also supply additional code to copy from a to a' before the loop and from a' to a after the loop. Unfortunately, synchronization must be inserted to satisfy the data dependences introduced by the new copy statements. One can argue that since the new dependences are outside of the inner loop, barrier synchronization can be used without too much penalty. However, this argument relies on the fact that the inner loop is invoked a large number of times. In fact, if short compilation time were not an important issue, then point-to-point synchronization could actually be implemented by once again invoking all compiler passes on the new program.

Observe that the above scheme can also be used to eliminate output dependences in a loop. However, in our experience, most array elements in a loop are modified by the same processors. Consequently, output dependences that both require synchronization and can benefit from the above analysis are rare.

5.5 Summary

This chapter discusses several optimizations to improve efficiency of programs that use point-to-point synchronization. First, we focus on the synchronization mechanism itself. Although implementing synchronization primitives through cache-coherent shared-memory accesses is straightforward, the underlying support of cache coherence results in many message exchanges. Instead, messages can be sent directly from one processor to another to perform synchronization. This scheme provides a faster synchronization mechanism for cases where synchronization targets are known statically.

When a dependence between two processors is automatically satisfied by synchronization to support other dependences, then the dependence is redundant. Even though the general problem of eliminating all redundant dependences is undecidable, most programs exhibit characteristics that allow for many redundant dependences to be detected. Dynamic-programming algorithms can be employed to detect redundant dependences in time $O(n^3)$ in the size of the program.

Of the three types of data dependences, only flow dependences represent information exchange. Output and anti-dependences only occur in a program because variables are reused. Unlike analysis to remove false dependences to increase parallelism, the scheme used here does not require all dependences to be eliminated. Thus we can make use of existing flow dependences to cause false dependences to become redundant with only a small number of replications. The same dynamic programming structure used to detect redundant dependences can be employed to compute the number of replications needed to eliminate false dependences.

Chapter 6

Results

The developments in this thesis rely on the premise that replacing barrier synchronization with point-to-point synchronization produces an improvement in program execution. Recall that there are two disadvantages of barrier synchronization: the cost of propagating information globally, and the unnecessary idling of processors due to global synchrony. The high overhead of global propagation is clearly evident in the case of software-supported barrier schemes since $2\log(P)$ messages must be sent to collect and distribute information. However, hardware-assisted barrier schemes reduce this overhead to be more similar to that of a single message. In contrast, point-to-point synchronization often requires the transmission of several messages since each processor typically must synchronize with several other processors. Thus any advantage of point-to-point synchronization over hardware barrier schemes must be due to unnecessary idling. Since deriving models that can accurately predict and use such dynamic characteristics is very difficult, simulation results can instead be studied to evaluate the impact of the above claims on parallel programs.

In this chapter, the simulation results of a number of applications using various synchronization schemes are presented. First, we briefly discuss the implementation of the compiler and its performance. A detailed discussion of a particular application is then given, followed by results on the general set of benchmarks.

6.1 Applications

The benchmarks used here are selected due to the fact that they satisfy several important criteria. First, the parallel machine model used here is one that employs the shared-memory semantics rather than message-passing for interprocessor communication. Rather than being a limitation, this feature actually allows easier porting of sequential code to a parallel machine. However, some available benchmark suites [Hey91] that rely on message-passing semantics cannot be used. Second, the derivations of this thesis assume that the input program contains fine-grained data parallelism. In other words, although each program is meant to be executed on a multiprocessor, its top-level organization is sequential, with parallelism occurring at lower levels. This assumption eliminates applications with high-level coarse-grain parallelism such as those in the Splash benchmarks [SWG91]. Finally, since the analysis only performs optimization on array indices that are linear functions of loop indices, applications are chosen whose array accesses predominantly fit such characteristics. Consequently, sparse-matrix applications with many indirect array accesses are omitted, as are algorithms such as Fast Fourier Transform where array accesses are base-two exponential functions of loop indices. Although one can execute the compiler on such examples, the resulting code would be no different than if one were to employ a simplistic barrier scheme.

Application	Description	Stmts	Inlined
Jacobi	Jacobi algorithm for solving Laplace's equation on a 48×48 grid. [Fox88]	23	
Red-black SOR	Solution to Laplace's equation using a checkerboard 48×48 grid. [Fox88]	32	
Gaussian	Gaussian elimination on a $32{\times}32$ matrix.	25	
Median	Repeated 3×3 median filter on a 24×24 image. [Lim90]	42	
Doacross SOR	Successive over-relaxation on a 64×64 grid using `DOACROSS` loops.	19	
WaTor	Ecological simulation presented by Fox, et al on 32×32 array. [Fox88]	219	
Shallow	Weather prediction based on finite-difference models of the shallow-water equations [Sad75] on 32×20 array.	155	
Simple	Fluid flow simulation adapted to a 24×24 array. [Cro78]	829	991
MICCG3D	Preconditioned conjugate gradient using modified incomplete Cholesky factorization on an $8 \times 8 \times 8$ array. [YA93]	527	4270

A list of applications used to derive the results in this chapter is shown in the table above. The two right-hand-side columns contain the number of statements in the original application and the number statements in a version where procedure calls have been inlined. Although interprocedural support exists for array flow analysis, the implementation of processor dependence computation does not treat procedure calls very intelligently. Thus rather than tolerating barrier synchronizations before and after each call to a parallel procedure, we instead inline those calls and perform the entire analysis on the inlined program.

Of the above benchmarks, the first five are small code fragments that can form the kernel of a real application. The last two represent real programs that have been translated into the appropriate syntax for this thesis. Note that the problem sizes are small due to two reasons. First, since results are obtained through simulation and not on a real machine, small problem sizes allow data collection to be possible in a reasonable amount of time. Second, small problem sizes per processor increase the significance of synchronization costs since communication and synchronization overhead tends to grow more slowly as a problem scales. Indeed, if one uses a large enough problem size which allows large amounts of local computation, then efficiency is mostly affected only by parallelization success, and few other compiler optimizations matter. One can also consider future trends where many more processors are present in a machine than the 64 used here. As the machine size increases, the problem size per processor is likely to decrease. In addition, the results obtained here are based on the simulation of somewhat idealized hardware with very low communication costs. On a real machine, the actual communication overhead can be much higher and can in turn affect execution time much more drastically. This issue is discussed in more detail later in this chapter.

6.2 Simulation environment

The multiprocessor simulations for this thesis are done using Proteus [Bre91]. Although this simulation tool allows varying many architectural parameters, the figures here are obtained for a fixed hardware model. The imaginary machine is composed of 64 nodes arranged in a 8×8 mesh with bidirectional links between nearest neighbors on the mesh. Each node contains a processor, a memory unit, and a hardware-supported coherent cache. The simulation uses the Alewife [Aga91] cache coherence protocol and also allows for explicit message sending between processors. Although most communication is accomplished through the shared-memory interface, some operations such as software-supported barrier synchronization are implemented using explicit messages.

A compiler which generates point-to-point synchronization for parallel programs has been implemented in C. The compiler accepts as input the syntax as given in the examples of this thesis and emits the augmented C code expected by Proteus. The different phases of compilation are illustrated in Figure 6-1. Observe that only the top two phases are implemented by conventional sequential compilers. Later phases correspond to analysis steps that are derived in this thesis.

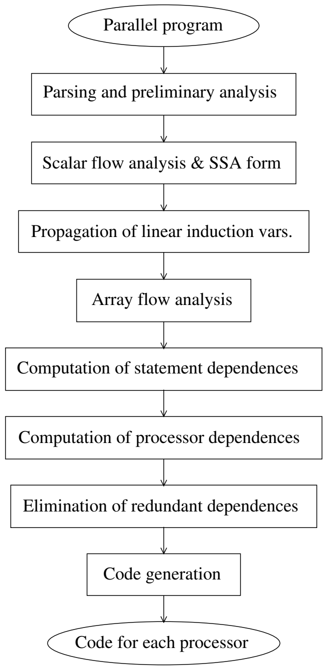


Figure 6-1: Compiler structure

The compilation time for several applications on a SparcStation IPC are shown in Figure 6-2. For smaller applications, the almost instantaneous compiler response did not allow for accurate measurement of individual phases. Illustrating the efficiency of algorithms presented here, the compiler finishes in under 35 seconds even on very large procedures. The expensive array flow analysis phase stems primarily from the fact that reaching sets are represented as linked lists. Instead, if one were to use hash tables or binary trees, then the time to search each set can be reduced from $O(n)$ to $O(log(n))$ or $O(1)$ and can significantly improve compiler running time. Note also that the time to compute processor dependences is significant despite the simplicity of the scheme presented here.

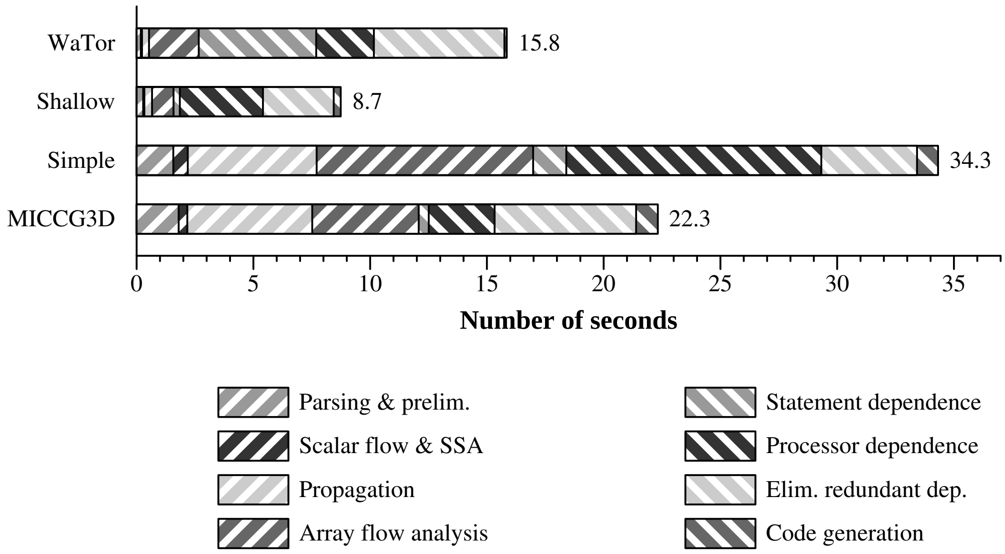


Figure 6-2: Compilation time for some applications

6.3 An example

In this section, we present an application which benefits greatly from point-to-point synchronization. Although this example is by no means representative of the benchmarks, it can be used to illustrate some strengths as well as weaknesses of the approach.

The WaTor program is adapted from an ecological simulation that appears in [Fox88]. Given a population of predators and prey with defined behavior, we wish to simulate the dynamics of the population in time. In this particular example, sharks form the predators and minnows form the prey. Both species inhabit a rectangular lake which is represented by a two-dimensional array. Each element in the array can either contain a shark or minnow or be empty. Each fish can move in one of four possible directions. On each time step, a minnow moves randomly to an adjacent empty array element and leaves an offspring if the minnow is older than a specified breeding age. A shark first searches for adjacent cells with minnows. If one exists, then it randomly moves to one such cell and eats the minnow. Otherwise, it moves as a minnow, but can die if it has not eaten for a certain time.

If one were to imagine a parallel simulation of the above lake, then potential update conflicts immediately arise. Imagine the situation where one processor p is updating an element containing a shark and another processor p' is updating an adjacent element

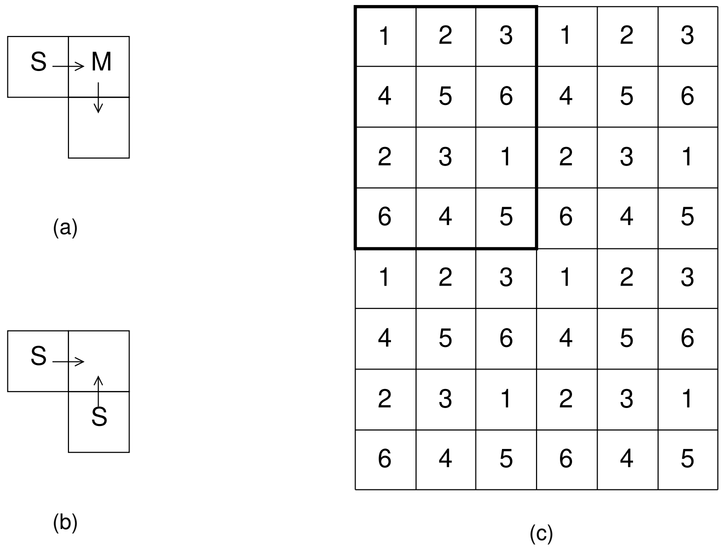


Figure 6-3: Eliminating update conflicts for the WaTor benchmark

containing a minnow as in Figure 6-3a. If p lets the shark consume the minnow and p' moves the minnow to another array element, then an inconsistency arises. Thus two processors cannot be updating two adjacent elements. Furthermore, a conflict also occurs when two processors try to deposit a fish into the same array element, as shown in Figure 6-3b. Consequently, a correct solution must ensure that at no time can two processors be updating two array elements that are separated by a Manhattan distance of 2 or less. The implementation considered here satisfies this constraint by tiling the array with a 6-color pattern as shown in Figure 6-3c.† On each phase of the update routine, only cells with a particular color are updated. In a 6-color scheme, an update iteration must contain six phases. Semantically, a barrier synchronization occurs between each color phase, ensuring that all updates are free of conflicts. Of course, the nearest-neighbor array usage of the application makes it a prime candidate for implementing point-to-point synchronization.

Using a block partitioning scheme, each processor is responsible for updating a block of the array. Array elements are shared at the boundary points of these blocks, and synchronization must be done to ensure that the accesses are performed in the correct order. If one imagines executing the code in Figure 6-4 on a 8×8 processor array, then the loop space can be partitioned into processors as illustrated. In order to ensure correct execution order, synchronization must be performed between each set of

† A 5-coloring can also be used to satisfy the constraints, but requires a larger tile pattern.

nested DOALL loops. Focusing on the dependences between colors 1 and 2, one observes that each processor (y, x) executing color 2 must synchronize with processors $(y, x-1)$, $(y, x+1)$, $(y+1, x)$, and $(y+1, x-1)$. Note that this analysis must be done between every set of loops and not just between consecutive loops. Fortunately, the farther apart the color phases are, the more likely it is that the dependences become redundant due to other dependences between the phases. From this example, one can see that the task of computing point-to-point synchronization can be very tedious and is best done by a compiler rather than a programmer.

```
doall (i=1,32,4)
  doall (j=1,32,3) {    /* color 1 */
    update(i, j);
    if (j<=30)
      update(i+2, j+2);
  }

doall (i=1,32,4)
  doall (j=2,32,3) {    /* color 2 */
    update(i, j);
    update(i+2, j-1);
  }

doall (i=1,32,4)
  doall (j=3,32,3) {    /* color 3 */
    update(i, j);
    update(i+2, j-1);
  }

...
```

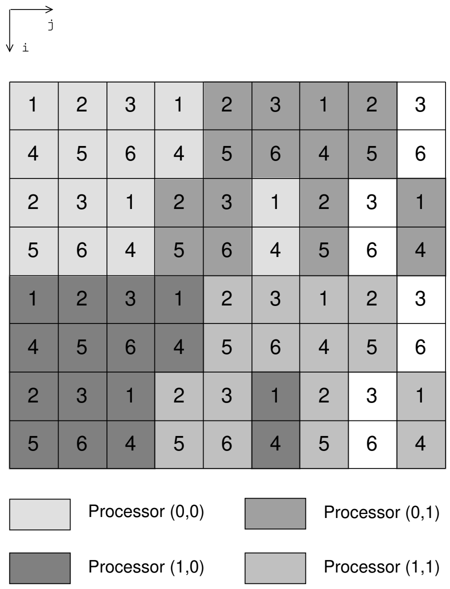


Figure 6-4: Partitioning a 32×32 WaTor array on a 8×8 processor array

In the actual code produced by the compiler, each processor must synchronize with 4 to 6 other processors before each color phase. In comparison against a software barrier scheme, the more local synchronization approach would clearly perform better. Instead, if one were to employ a hardware barrier scheme, then the cost of reading 4 remote memory locations is probably similar to that of executing a hardware barrier. At first glance, one may not expect any difference in performance between the two implementations. However, one has not considered the second disadvantage of barrier synchronization: unnecessary idling. In this particular example, the variance in execution of each iteration can be very large. Much more processing must be done for array elements that contain fish than those that are empty. This unbalanced loading also varies dynamically as fish move and regenerate. If global synchronization is performed between each phase, then the time to execute each phase is equal to the time of the busiest processor during the phase. Instead, if point-to-point synchronization were implemented, then the execution of different phases can overlap in time and the effects of busy processors can be minimized. This effect can be seen by comparing the execution profiles of the two schemes, as shown in Figure 6-5. In the no-cost barrier scheme, no time elapses between the last processor entering a barrier and the barrier exit by all processors. Even with such an ideal barrier, the illustration shows that the point-to-point synchronization scheme provides superior performance.

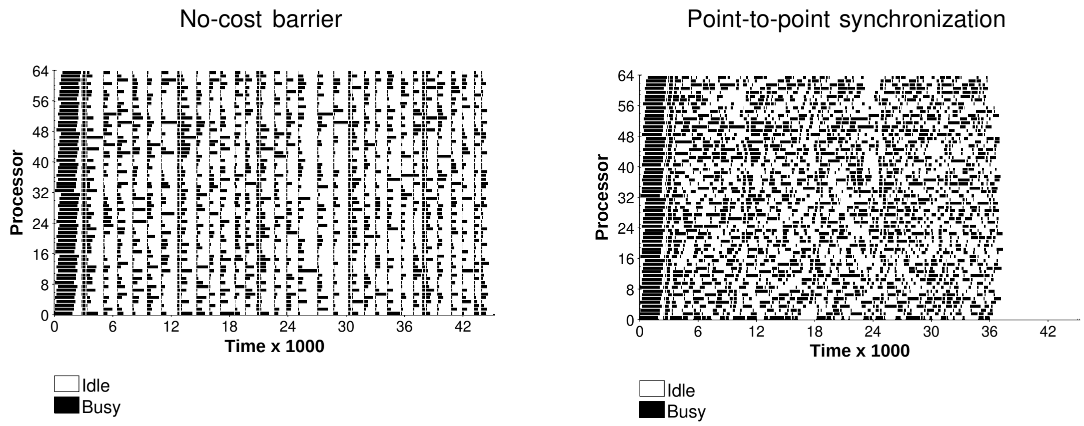


Figure 6-5: A comparison of synchronization schemes on the WaTor benchmark

One may wonder how the disadvantages of barrier synchronization are affected by problem size. As a problem becomes larger and each processor spends more time in each phase on computation, the constant overhead of software barriers becomes less significant. With very large problem sizes, one would also expect the variance in load on individual processors to decrease. In this particular example, the fish population on each processor can be represented by a binomial of n coefficients where n is the number of elements per processor. As n increases, the variance of the fish population decreases, which in turn reduces the penalty for global synchronization. Indeed, for any distribution, the standard deviation of the average of n identical events scales as $1/\sqrt{n}$ [Fel68]. One would expect the relative overhead due to unnecessary idling to be related to this ratio.

Figure 6-6

shows the effects of different synchronization schemes on varying problem size. Software barrier synchronization is accomplished by using a message-passing spanning tree which requires around 450 processor cycles to execute on a 64-processor machine. For both barrier schemes, synchronization is inserted only where necessary as computed by the flow analysis. No-cost point-to-point synchronization implies that no time elapses from a synchronization assertion by the source processor and the observation of that assertion by the sink processor. In studying the graph, one can view the difference between the software and no-cost barriers as the penalty due to global propagation. The difference between no-cost barriers and no-cost point-to-point synchronization can be viewed as the penalty due to unnecessary idling. All times are normalized with respect to the software barrier time. From the graph, we see that as problem size increases, the penalties due to both inefficiencies decrease when compared with overall execution time.

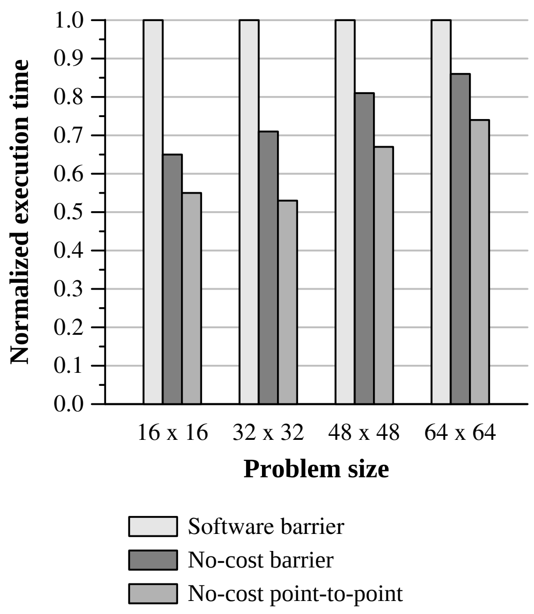


Figure 6-6: WaTor performance for varying problem sizes

Although the WaTor application possesses characteristics that enable point-to-point synchronization to be advantageous, such characteristics cannot be readily extracted from every representation of the program. In order to allow the compiler to provide significant results, the application had to be written in a particular way. As the first of several examples, consider the code in Figure 6-4. Each color phase is separated into its own set of nested loops, thus allowing for the compiler to be explicitly aware of the array elements that are active for each phase. This knowledge in turn enables processor synchronization targets to be computed intelligently and results in each processor only requiring synchronization with a few other processors between phases. Instead, if each phase is not represented by its own loop, but simply by an outer loop as in Figure 6-7a, then the phases are no longer lexically distinguished. Any knowledge about the structure of the colors within the array are hidden. The compiler must treat the loop body as executable by any phase and conclude that each processor must synchronize with all 8 of its neighbors.

```
do (c=1,6)
  do (i=1,32)
    do (j=1,32)
      if (color[i,j]==c)
        update(i,j);
```

(a)

```
if (dir==0)
  a[i-1,j] = ...;
if (dir==1)
  a[i+1,j] = ...;
if (dir==2)
  a[i,j-1] = ...;
if (dir==3)
  a[i,j+1] = ...;
```

(b)

```
i1 = i+dy[dir];
j1 = j+dx[dir];
a[i1,j1] = ...;
```

(c)

Figure 6-7

As another example, consider the code fragment in Figure 6-8b. For each of the four directions that a fish can move, a statement exists to modify the particular array element in that direction. This allows the compiler to deduce that the set of elements of a that can be changed for coordinate $(\mathtt{i}, \mathtt{j})$ are: $\{(\mathtt{i}-1, \mathtt{j}), (\mathtt{i}+1, \mathtt{j}), (\mathtt{i}, \mathtt{j}-1), (\mathtt{i}, \mathtt{j}+1)\}$. Now consider the more cleanly written version in Figure 6-8c where the update is done by one statement and arrays dx and dy represent the changes in i and j for each direction. In the current compiler, nothing is deduced about array values and the compiler must consequently assume that the update can happen to any possible array element. Any dependences with this statement can then only be satisfied by a barrier synchronization since the relationship between the processor and data spaces has been lost. Even if one makes the reasonable assumption that the compiler can deduce that every element in x and y are in the range $[-1, 1]$, this information only allows one to limit the range of updates to one of nine elements. In order to recover fully what the separate treatment of directions provided, the compiler must somehow realize the coupled relationship between elements of x and y. It must be able to infer that whenever the arrays are accessed together, only four possible values can result. However, this is too much to expect out of the analysis tools of today.

6.4 General application study

In a sense, the WaTor application is an ideal application for point-to-point synchronization. It contains regular array accesses which allow the proposed analysis to be effective and also possesses dynamic run-time behavior which penalizes global synchronization schemes. Unfortunately, such characteristics may not be representative of many other applications. In this section, we seek to compare the performance of various synchronization schemes on the benchmark applications.

The first comparison involves the same schemes used for the WaTor application. One would like to isolate the significance of each of the two disadvantages of global barriers. The cost to propagate information globally can be viewed as the difference between a software-implemented barrier and a no-cost barrier. The cost due to unnecessary processor idling can then be measured as the difference between a no-cost barrier scheme and that of a no-cost point-to-point scheme. First, we define the synchronization schemes more precisely.

Flow software barrier: A tree-based message-passing barrier is inserted only when synchronization is required between processors. The time elapsed between the last entrance into the barrier and the first exit is near 450 cycles. In addition, redundant barriers are removed whenever more than one barrier satisfy the required dependences. In a sense, this scheme represents the best performance that one can achieve with a global barrier mechanism.

Flow no-cost barrier: This technique is similar to the above, but the barrier synchronization does not incur any cost. In other words, no cycles elapse between the last entrance and the first exit from the barrier.

No-cost point-to-point: A no-cost point-to-point synchronization primitive is used wherever possible. However, when not enough information is available to compute synchronization targets, a no-cost barrier is invoked.

Real point-to-point: A shared-memory point-to-point synchronization primitive is used wherever possible. A software barrier is used when not enough information is available to compute synchronization targets.

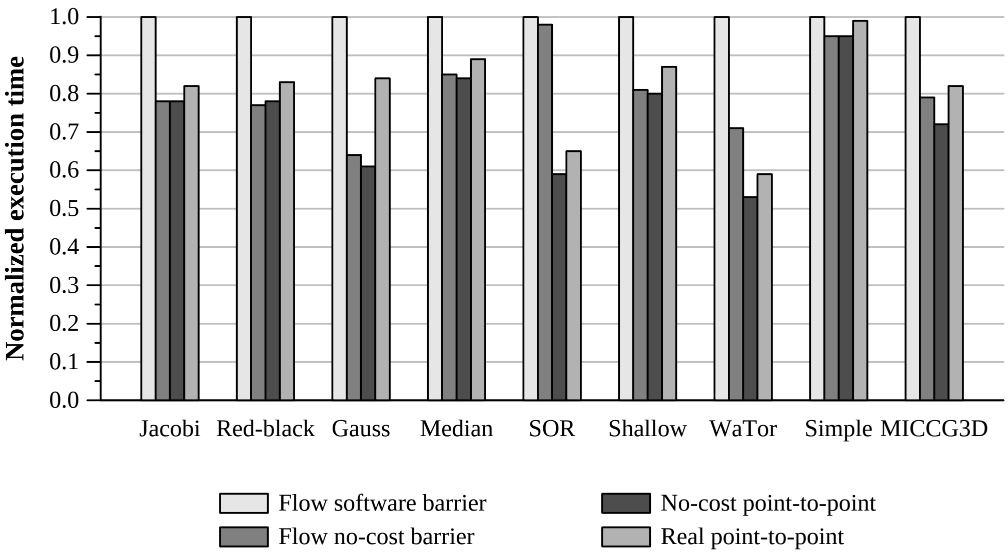


Figure 6-8: Comparing flow-analyzed synchronization schemes

Figure 6-8

illustrates the execution times of such synchronization schemes normalized with respect to the software barrier scheme. Observe that the cost due to information propagation is significant in almost every application. For larger problem sizes, this overhead is expected to become less important as computation costs begin to dominate.

The difference in execution time due to unnecessary idling appears to be insignificant in most applications other than WaTor and Doacross SOR. As discussed previously, the idling in WaTor stems from processors having varying loads on different coloring phases. In the case of Doacross SOR, idling is instead due to skewed execution among processors. Because of the nature of `DOACROSS` loops, processors responsible for later iterations of the loop are required to execute after previous iterations have been completed. As shown in Figure 6-9, this feature produces a skew in finishing times. If the `DOACROSS` loop is then re-invoked due to an outer `DO` loop, then using a barrier requires all processors to wait for the last processor to finish the `DOACROSS` loop. Instead, using point-to-point synchronization allows the first processors to begin the next iteration before the last processors finish the previous iteration. Similar to the load variance of WaTor, the skew effect in Doacross SOR does become less significant as problem size increases. With a cyclic distribution, as more points in the iteration space are assigned to each processor, the amount of time each processor must wait only increases by the the square root of those number of points. This factor can be further decreased by employing a cyclic distribution. However, such an approach results in additional communication due to poor locality.

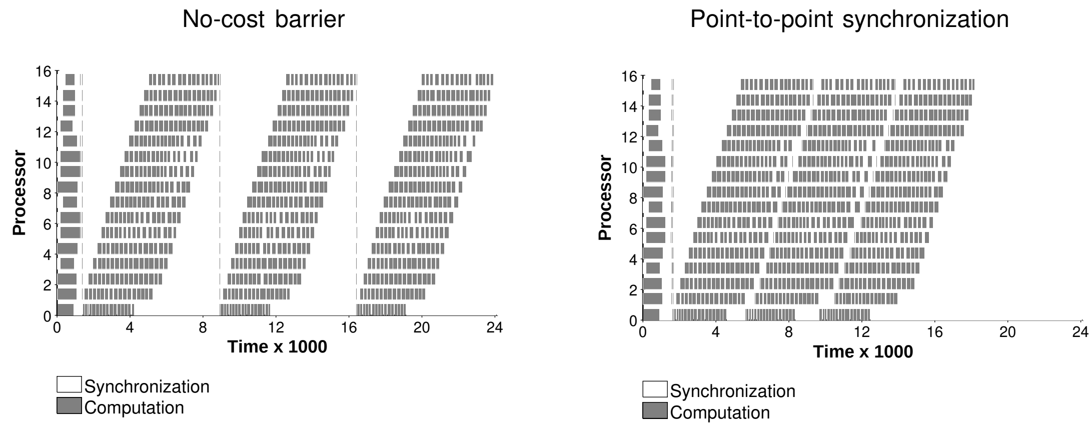


Figure 6-9: Synchronization schemes on the Doacross SOR benchmark

As one may assume from the discussion, the above results are obtained while always performing point-to-point synchronization between DOACROSS loop iterations, even for the software and no-cost barrier cases. Two explanations can be given for this approach. First, since this thesis focuses on providing synchronization support for dependences between parallel loops, any advantages due to providing point-to-point support for DOACROSS loops should be eliminated. Hence, all results presented in this chapter use the same scheme to synchronization between DOACROSS iterations. Second, if DOACROSS synchronization were not available, then the loops would be written differently to allow for iterating over hyperplanes instead of array axes. This requires representing array indices as functions of multiple loop indices, which cannot be recognized by the techniques of this thesis. From the perspective of barrier-based synchronization, there should be no real difference in execution performance. However, the point-to-point derivations presented here would not be able to take advantage of such a program.

Of the above benchmarks, the two that rely heavily on DOACROSS loops are Doacross SOR and MICCG3D. As shown above, Doacross SOR benefits greatly from point-to-point synchronization due to its skewed execution. MICCG3D, however, does not exhibit such improvements. This can be explained by studying a vital set of loops in the application where a matrix is being solved by forward and backward substitution. In the forward-substitution phase, values are propagated from one corner of the three-dimensional matrix towards the opposite corner. Immediately afterwards, the backward-substitution phase propagates values from that opposite corner back to the original corner. Processors responsible for the first corner cannot proceed until the last corner has finished and propagated its values through most of the matrix. Consequently, the skew introduced by `DOACROSS` loops cannot be exploited in this portion of the program.

The reader may be tempted to make comparisons between the no-cost barrier times and those of real point-to-point synchronization. However, one must remember that the no-cost barrier is an idealized version that does not exist in physical machines. To derive an estimate for performance on a machine with a more realistic hardware barrier, one merely needs to interpolate between the no-cost barrier and the 450-cycle software barrier. If one were interested in the figures for a 50-cycle barrier, then the additional barrier overhead can be viewed as 1/9 of the difference between no-cost and software barriers.

At this point, one may be interested in the performance comparison between point-to-point synchronization and that of a more naive barrier synchronization scheme. After all, if one were willing to perform all the flow analysis to insert barriers intelligently, one may as well use point-to-point synchronization to obtain an even higher improvement in performance. Shown in Figure 6-10 are the results for point-to-point synchronization compared to naive barrier schemes. Specifications of the schemes are given below:

Naive software barrier: Software barriers are inserted at the beginning and end of every set of nested parallel loops.

Naive no-cost barrier: No-cost barriers are inserted at the beginning and end of every set of nested parallel loops.

Point-to-point: A shared-memory point-to-point synchronization primitive is used wherever possible. A software barrier is used when not enough information is available to compute synchronization targets. Optimizations are performed to remove redundant dependences. This is the "real point-to-point" result of the previous figure.

Unoptimized point-to-point: This scheme is similar to the point-to-point scheme, but no optimizations are invoked to remove redundant synchronizations.

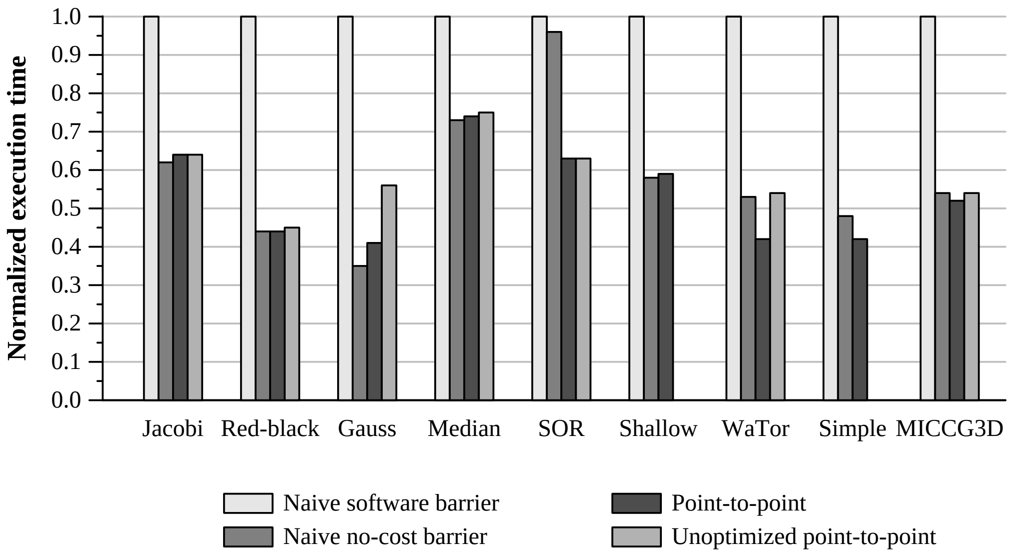


Figure 6-10: Comparing naive barrier and point-to-point schemes

By using the naive approach, the overhead for global propagation is magnified, as verified by analyzing the difference between software and no-cost barriers. Note that it may be possible to follow some simple heuristics to reduce the number of barrier synchronizations performed, especially when parallel loops immediately follow each other. However, this thesis does not explore such heuristics.

One can also observe from the graph that the performance of point-to-point synchronization approaches or exceeds that of the no-cost barrier for the given benchmarks. For the most part, the comparison of point-to-point synchronization to naive no-cost barriers is very similar to the comparison with intelligent no-cost barriers. If additional barriers do not increase overhead, then any additional cost can only be due to unnecessary idling introduced by barriers at new locations in the program. For the above applications, such situations do not arise, and the execution time of naive no-cost barriers is similar to that of intelligent no-cost barriers.

The above graph also illustrates the difference in performance when optimizations are performed to remove redundant dependences. Unfortunately, for some of the more significant cases, the program complexity due to redundant dependences exceeds the limits of the host compiler. Hence, pre-elimination simulation figures were not obtainable for the Shallow and Simple benchmarks. However, a lexical count of redundant dependences can be acquired. Figure 6-11 shows the percentage of dependences that are found to be redundant by the recursive algorithm presented in Chapter 5. For the larger applications, the high percentages represent a significant reduction in communication required for synchronization.

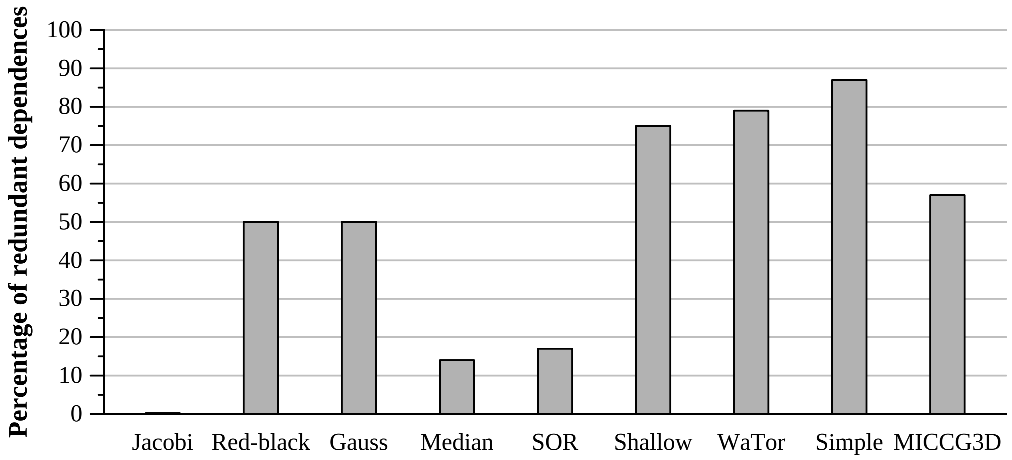


Figure 6-11: Percentage of redundant lexical dependences

6.5 Summary

The above results show that compiler analysis to support point-to-point synchronization can be done efficiently. The performance of resulting code display significant improvements over that of software barrier schemes, particularly when software barriers are naively inserted for all parallel loops. For programs with regular array usage such as the above benchmarks, these results illustrate that hardware support for barriers are unnecessary and in certain cases even inferior to point-to-point synchronization.

Chapter 7

Future work

7.1 Introduction

This thesis focuses on obtaining efficient algorithms to implement point-to-point synchronization for a large set of programs. However, with limited sources, one cannot possibly hope to provide optimally efficient algorithms for the set of all programs. Hence, many possibilities remain for improvements to be made to the current work. Some of these focus on providing support for more general programs such as performing analysis on multiple loop indices and computing dependence relationships across procedures. Others involve techniques to improve efficiency such as using synchronization groups and increasing awareness of synchronization in partitioning decisions.

7.2 Multiple loop indices

The analysis done in this thesis is limited to array elements that are linear functions of a single loop index. Although this restriction allows optimizations to be performed efficiently on a large class of programs, some array reference patterns that are typically supported in state-of-the-art compilers are not considered here. One such type of usage involves linear functions of multiple variables.

```
do (i=1,100)
  doall (j=1,i)
    a[j,i-j+1] = a[j-1,i-j+1]+a[j,i-j];
```

Figure 7-1

More general support for array references also implies that one consider array indices that are functions of multiple loop indices. The program of Figure 7-1 represents a wave-front computation which possesses dependence relationships similar to the Doacross SOR example in the previous chapter. If one assumes the constraint that each value of `j` is executed on processor j, then a processor j executes instances $\langle k, i-k+1 \rangle$ for all values of i. The array access `a[j-1,i-j+1]` is then executed by processor $j-1$, and the array access `a[j,i-j]` is executed by processor j. Thus each processor j must synchronize with processor $j-1$.

One can observe from the above example that the derivations to compute synchronization relationships must be changed to support such array references. In particular, the space of filtered instances is no longer necessarily orthogonal to the loop index axes. One can apply the more general algorithms of Feautrier [Fea91] or Maydan [MAL93] to compute the needed results. In adapting these algorithms, however, it is important to remember the desired goal. To perform synchronization, we only need to compute the processors represented by the filtered space and some reasonable estimate of the upper bound of its timestamps. Deriving any extra information that requires more complex algorithms is merely a waste of compiler effort.

7.3 Interprocedural analysis

As mentioned in the chapter on flow analysis, some simple interprocedural analysis is performed in the implementation of this thesis to support dependences across procedures. However, such a simple approach produces many inefficiencies that can be addressed by more intelligent schemes.

Fundamental to the simple technique is the assumption that the output program contains only one version of each procedure. As shown in Chapter 3, this assumption requires one to be overly pessimistic in generating code for the procedure. Any possible dependences that can arise within the procedure body must be supported without any attention to the actual values that are passed in as arguments. In addition, such an assumption also does not permit specialization of procedures for synchronization between the caller and the procedure.

In the program of Figure 7-2, both uses of `a` in statements `S1` and `S2` require synchronization with any definitions of `a` that occur before the call to `f`. By requiring only one version of `f`, the caller must be pessimistic and assume that either statement may be executed and synchronize accordingly. In this case, since nothing is known about the index `g(i)`, the dependence must be satisfied by a barrier synchronization. One can argue that the dependence can be supported by performing an assertion before the call to `f` and execution the checks inside the body of `f` in either branch of the conditional.

```
void f(int x)
{
  if (p(x))
    doall (i=1,100)
      ... = a[i];        /* S1 */
  else
    doall (i=1,100)
      ... = a[g(i)];     /* S2 */
}
```

Figure 7-2

However, the processor targets in the checks can vary depending on the definitions preceding the call to `f`. Thus such an approach can be accomplished only by allowing several different versions of the procedure to co-exist. As a side note, it should be mentioned that the above scenarios are supported in the current implementation by inlining the procedure call. However, a more intelligent mechanism should be provided than merely specializing every call to a procedure.

7.4 Synchronization groups

Although most of this thesis focuses on the distinction between the extremes of global barrier synchronization and local point-to-point synchronization, one should also observe that intermediate schemes do exist. For some cases, the lack of absolute information on processor relationships does not necessarily imply that one must rely on barrier synchronizations. Rather, techniques used to improve the performance of barriers can be applied to such cases to allow synchronization on groups of processors.

```
doall (i=1,100)
  doall (j=1,100)
    a[j,i] = ...;            /* S1 */
...
doall (i=1,100)
  doall (j=1,100) {
    ... = a[j,f(i)];         /* S2 */
    ... = a[j-1,g(i)];       /* S3 */
}
```

Figure 7-3

Consider the dependence between `S1` and `S2` in Figure 7-3. Assume that the behavior of the functions `f` and `g` are unknown. Although the first array indices match exactly, the second provides no filtering information on the loop index `i`. Thus each processor must synchronize with all other processors in the same partition of `j`. In other words, if the `i` loop partitions the processors into rows and the `j` loop partitions the processors into columns, then each processor must synchronize with all other processors in its column. If one were restricted to either pairwise point-to-point synchronization or barriers, then a barrier synchronization is probably more efficient than many pairwise synchronizations. However, a global barrier represents much more serialization than the dependences require. Ideally, only the processors within each column should be synchronized with each other. Consequently, one can introduce a ''mini-barrier'' which synchronizes only certain groups of processors. In this case, each column of the processor space forms such a group.

While the concept of a mini-barrier forms an effective solution for the above example, a more general mechanism is needed to support other cases. Consider the dependence between `S1` and `S3` in Figure 7-3. Assume that there are 100 columns in the processor space so that each iteration of `j` is partitioned to a separate column. The filters imply that processors in column j must synchronize with processors in column $j-1$ to preserve the dependences. Such a requirement cannot be satisfied by performing a mini-barrier on each column. Instead, one needs to divide the barrier mechanism into two phases: collection and distribution. The collection phase gathers signals from each processor that it has arrived at the barrier. Only after all processors have arrived does the distribution phase begin, which signals each processor that it can proceed with the execution. As applied to this example, one needs to collect signals from processors in column $j-1$ and then distribute that barrier signal to processors in column j. In general, the collection of barrier signals from processors in a group G can be distributed to several groups which may include G itself. Note that this decoupling of collection and distribution also allows the two phases to be done at different points in the program. In the above example, collection can be done immediately after the first loop nest, while distribution is not required until the beginning of the second loop nest. This separation forms the exact mechanism touted by the fuzzy barrier schemes [Gup89].

In summary, the above discussion shows that synchronization mechanisms other than purely global or local schemes may be useful. By viewing synchronization as a collection phase followed by a distribution phase among possibly different groups of processors, one can introduce a scheme that encompasses both barriers and point-to-point synchronization. Moreover, this scheme allows one to implement more efficient mechanisms for cases that are too ambiguous for point-to-point synchronization.

7.5 Partitioning

Processor partitioning can be defined as an optimization which maps computations to processors in order to maximize performance. Traditionally, such optimizations aim for this goal by striving to minimize communication across processors. In the language of this thesis, conventional partitioning schemes map statement instances to processors while minimizing flow dependences between instances on different processors. However, when synchronization costs are also considered, then other dependences become important as well.

```
doall (i=2,100)
  ... = a[i-1];        /* S1 */
doall (i=2,100)
  a[i] = ...;          /* S2 */
```

Figure 7-4

Consider the program in Figure 7-4. Although no communication exists between statements `S1` and `S2` as shown, there does exist an anti-dependence between the statements. If the two loops are partitioned identically, then synchronization is required to support the anti-dependence. Instead, if the partitioning function for the second loop is offset by 1 from that of the first loop, then no anti-dependences exist across processors, and no synchronization is required. Thus with all other factors being equal, a partitioning scheme that also pays attention to synchronization costs can produce better results. However, such partitioning and alignment decisions must frequently be weighed against other factors such as load-balancing. In this particular example, the synchronization cost would certainly be higher if one were forced to perform a software barrier rather than point-to-point synchronization, and partitioning algorithms must be aware of such details.

The use of point-to-point synchronization creates small changes in program behavior which in turn increases the factors that must be considered by partitioning algorithms. In particular, the existence of large skews between loop iterations imply that decisions that

```
do (j=1,10) {
  doacross (i=1,100) {
    a[i] = ...;
    ...
  }
  amax = max of array a
}
```

Figure 7-5

were made arbitrarily under barrier semantics become very important when skews are preserved across loops. In the program of Figure 7-5, each iteration of the outer DO loop consists of a DOACROSS loop followed by a reduction operation. A reduction operation typically maps a binary tree onto the processor space and propagates the results of an associative operation up the tree. When synchronization is performed using barriers, the skews at the end of the DOACROSS loop are eliminated and all processors begin executing the reduction simultaneously. With such semantics, the mapping of the reduction tree to processors does not have many implications. Specifically, the program performance is not drastically affected by whether the root of the tree is assigned to the first or last processor. However, with point-to-point synchronization, the preservation of skew across the outer sequential loop allows the execution of those loop iterations to be pipelined, as shown by the Doacross SOR application in the previous chapter. The assignment of reduction tree nodes to processors becomes very important since the root node cannot be computed until all processors have completed. As shown in Figure 7-6, if the root of the reduction tree is assigned to the first processor, then the skews are lost across the sequential iterations. Instead, if the root is assigned to the last processor, then the skews are preserved.

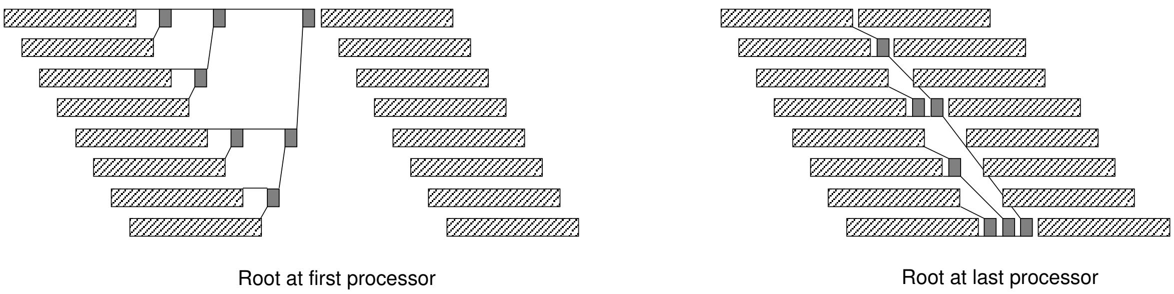


Figure 7-6: Alternate partitionings of a reduction

The issues involved in partitioning can become quite complex, and this discussion has no intention of solving them. Rather, these examples only serve to point out new factors that can arise when one considers synchronization in conjunction with partitioning.

Chapter 8

Conclusion

8.1 Summary

The shared-memory programming model requires that synchronization be performed in order to preserve data consistency. Traditionally, consistency is ensured by performing a global barrier synchronization between parallel sections of code. Although it provides a simple interface for the compiler or programmer, the barrier synchronization possesses several disadvantages. In order to synchronize globally, information must be collected from every processor which implies a latency of $O(\log n)$ on the number of processors. Furthermore, global synchronization forces the serialization of many tasks that do not contain dependences to each other and can thus increase total idle time. Instead of using global synchronization, this thesis seeks to reduce the above costs by performing local point-to-point synchronization between pairs of processors.

Compiler analysis to implement point-to-point synchronization requires that some assumptions be made about the input program. In this thesis, we focus on programs with explicitly-parallel loops and array references that are linear functions of loop indices. In addition, we assume that partitioning decisions have been made by a previous phase of the compiler and specified as mappings from the loop iteration spaces to the processor space.

The first analysis task involves deducing whether an array reference is a function of a loop index. By viewing this task as a propagation problem on a particular lattice, efficient existing propagation algorithms can be employed to generate a solution. The algorithm used here performs constant propagation by propagating over the static-single-assignment graph. While constant propagation makes use of a flat lattice, propagation of linear functions can be represented best by a lattice that allows for unions of functions. By limiting the height of this lattice, the propagation algorithm can be guaranteed to terminate.

Once array indices are determined, dependences between statements can be computed. Accurate dependence information requires that flow analysis be performed to compute reaching definitions and uses at each lexical point. Unfortunately, conventional scalar analysis is not sufficient due to their treatment of arrays as monolithic objects. Instead, array flow analysis must be employed to track the flow of individual elements of an array. A mapping from linear functions to array subsets enables efficient management of flow elements. However, such a mapping involves forming approximations and must be carefully designed to ensure that a superset of the real dependences will be detected. After the completion of array flow analysis, well-known dependence tests can be used to compute dependences between statements.

Statement dependences yield lexical dependence information which can be used to compute where synchronization primitives are placed in a program. However, nothing is as yet derived on dependence relationships between processors. Since such relationships require dynamic dependence information, we focus on dependences between dynamic statement instances rather than lexical statements. A statement instance is defined as the combination of the lexical statement and the values of loop indices of surrounding loops. A dependence exists between two statement instances if a dependence exists between the two lexical statements and if the array indices of each statements evaluate to the same values for the given instances. From this definition, dependences between statement instances can be computed.

In order to derive dependences between processors, one must consider the loop partitioning functions. For a given sink processor, the set of source processors with which it must synchronize can be computed from the source statement instances with dependences to the sink statement instances represented by the sink processor. In addition, one can focus on timestamps represented by sequential loop indices in each instance to compute temporal dependence information. Whereas each sink processor must synchronize with all dependent source processors, synchronization must only be performed with the highest timestamp since the timestamp ordering follows that of execution order.

Although one can show that the above derivations provide synchronization for every dependence, such claims are not enough to ensure correctness. In the presence of dynamic control flow, one must prove that each synchronization check can eventually be satisfied by an assertion on the proper processor. If a scenario can exist where all processors are checking for synchronization, then a deadlock condition arises. To avoid deadlock, the program must be transformed so that either branch of a conditional contains assertions that are equivalent to those of the other branch. By following this simple condition, a provably deadlock-free synchronization scheme can be derived.

Improving execution time of a parallel program represents the ultimate goal of these optimizations. However, a scheme derived from the above discussion can contain many redundant synchronizations that are automatically satisfied by combinations of other synchronizations. Since each synchronization operation incurs a certain cost, optimizations to eliminate redundant dependences can significantly improve running time. Unfortunately, removing all redundant dependences is an undecidable problem due to the lack of static knowledge of control flow. Even without the presence of dynamic control flow, the problem can be shown to be NP-hard. However, its integer-based characteristics allow it to be solvable by the application of dynamic programming techniques.

If one were to limit programs to sequences of non-nested parallel loops with offset-based array indices, then an algorithm can be introduced which removes all redundant dependences. The idea involves propagating the satisfied synchronization relationships from a source node to a sink node. If any synchronization between the two nodes is already satisfied, then it is redundant. Although this algorithm eliminates all redundant dependences and exhibits polynomial running time, its scope remains limited. As more general constructs are allowed in the problem domain, one must relax the constraint that the algorithm find all redundant dependences. This thesis employs a recursive algorithm which follows the program structure to eliminate redundant dependences.

The algorithms presented in this thesis have been implemented in a compiler which translates the source language into code for the Proteus simulator. Even on very large programs with up to 4000 statements, efficient algorithms enable the compiler to perform all optimizations in well under a minute. The simulated results on several benchmarks show that point-to-point synchronization produces significantly better running times than a naive scheme which insert software barriers before and after every parallel section. When compared to a no-cost hardware barrier, the performance of point-to-point synchronization approaches that of the no-cost scheme for most applications and even surpasses it for some applications.

8.2 Contributions of this thesis

The primary contribution of this thesis involves the creation of a scheme which automatically generates point-to-point synchronization to satisfy data dependences between parallel loops. However, in the course of pursuing such a goal, solutions to many other problems have required either the adaptation of known approaches or the invention of new ones. The principal contributions of this thesis include the following:

- The adaptation of existing constant propagation algorithms to enable propagation of symbolic functions by using a different lattice. In this thesis, the propagation lattice consists of linear functions of loop indices.

- A lattice-based treatment of array flow analysis which allows the preservation of linear functions for accurate dependence testing. Flow algorithms are presented for explicitly-parallel `DOALL` loops as well as common language constructs.

- The recognition that synchronization should be computed by considering dependences between dynamic statement instances. By using array references to derive filters, a general algorithm can be given for computing such dependence relationships.

- The use of a formal definition of loop partitioning functions to derive dependence relationships between processors.

- The employment of timestamps to support accurate synchronization relationships. In addition, transformations to maintain consistent timestamp assertions allow the derivation of a deadlock-free synchronization scheme.

- The separation of the task of computing dependence relationships between instances into two phases. The array flow analysis and dependence testing phase computes dependences between lexical statements, and the filtering phase computes dependences between dynamic instances of those statements.

- The introduction of a dynamic programming algorithm to eliminate redundant dependences. By reducing the problem to that of integer programming, limits can be placed on the answers in order to efficiently remove redundant dependences in similar domains.

Bibliography

[Aga91] Anant Agarwal, et al. The MIT alewife machine: A large-scale distributed-memory multiprocessor. Technical Report MIT/LCS/TM-454, MIT Laboratory for Computer Science, June 1991.

[AH91] Santosh G. Abraham and David E. Hudak. Compile-time partitioning of iterative parallel loops to reduce cache coherency traffic. *IEEE Transactions on Parallel and Distributed Systems*, 2(3):318–328, July 1991.

[AJ87] N. S. Arenstorf and H. F. Jordan. Comparing barrier algorithms. Technical Report ICASE 87-65, ICASE, Nasa Langley Research Center, September 1987.

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

[ASU86] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, Reading, Massachusetts, 1986.

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

[BC86] Michael Burke and Ron Cytron. Interprocedural dependence analysis and parallelization. *ACM Transactions on Programming Languages and Systems*, 8(3):162–175, June 1986.

[Bor90] Shekhar Borkar, et al. Supporting systolic and memory communication in iWarp. In *Proceedings of the International Symposium on Computer Architecture*, pages 70–81, 1990.

[Bou72] W. J. Bouknight, et al. The Illiac IV system. *Proceedings of the IEEE*, 60(4):369–388, April 1972.

[BP90] Micah Beck and Keshav Pingali. From control flow to dataflow. In *Proceedings of the International Conference on Parallel Processing, Volume II*, pages 43–52, 1990.

[Bre91] Eric A. Brewer, et al. Proteus: a high-performance parallel-architecture simulator. Technical Report MIT/LCS/TR-516, MIT Laboratory for Computer Science, September 1991.

[Cap87] Peter R. Cappello. Space time transformation of cellular algorithms. In Jr. Earl E. Swartzlander, editor, *Systolic Signal Processing Systems*, pages 161–207. Dekker, 1987.

[CC77] Patrick Cousot and Radhia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction and approximation of fixpoints. In *ACM Symposium on Principles of Programming Languages*, pages 238–252, 1977.

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

[CFKA90] David Chaiken, Craig Fields, Kiyoshi Kurihara, and Anant Agarwal. Directory based cache-coherence in large-scale multiprocessors. *Computer*, 23(6):49–58, June 1990.

[CH78] Patrick Cousot and Nicolas Halbwachs. Automatic discovery of linear restraints among variables of a program. In *ACM Symposium on Principles of Programming Languages*, pages 84–96, 1978.

[Che86] Marina C. Chen. A design methodology for synthesizing parallel algorithms and architectures. *Journal of Parallel and Distributed Computing*, 8:461–491, 1986.

[CHH89] Ron Cytron, Michael Hind, and Wilson Hsieh. Automatic generation of DAG parallelism. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 54–68, 1989.

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

[CLR90] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*. MIT Press, Cambridge, Massachusetts, 1990.

[Cro78] W. P. Crowley, et al. The SIMPLE code. Technical Report UCID 17715, Lawrence Livermore Laboratory, February 1978.

[Cyt86] Ron G. Cytron. Doacross: Beyond vectorization for multiprocessors. In *Proceedings of the International Conference on Parallel Processing*, pages 836–844, 1986.

[Cyt91] Ron Cytron, et al. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, October 1991.

[Dal92] William J. Dally, et al. The message-driven processor: A multicomputer processing node with efficient mechanisms. *IEEE Micro*, 12(2):23–39, April 1992.

[DGS93] Evelyn Duesterwald, Rajiv Gupta, and Mary Lou Soffa. A practical data flow framework for array reference analysis and its use in optimizations. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 68–77, 1993.

[DH88] Helen Davis and John Hennessy. Characterizing the synchronization behavior of parallel programs. In *Proceedings of Parallel Programming: Experience with Applications, Languages, and Systems*, pages 198–211, 1988.

[DSB88] Michel Dubois, Christoph Scheurich, and Faye A. Briggs. Synchronization, coherence, and event ordering in multiprocessors. *Computer*, 21(2):9–21, February 1988.

[EHLP91] R. Eigenmann, J. Hoeflinger, Z. Li, and D. Padua. Experience in the automatic parallelization of four Perfect-Benchmark programs. In U. Banerjee, D. Gelernter, A. Nicolau, and D. Padua, editors, *Languages and Compilers for Parallel Computing–Fourth International Workshop*, pages 65–83. Springer-Verlag, 1991.

[Ell85] John R. Ellis. Bulldog: A compiler for VLIW architectures. Technical Report YALEU/DCS RR # 364, Yale University Department of Computer Science, February 1985.

[Fea88] Paul Featrier. Array expansion. In *International Conference on Supercomputing*, pages 429–441, 1988.

[Fea91] Paul Feautrier. Dataflow analysis of array and scalar references. *International Journal of Parallel Programming*, 20(1):23–53, 1991.

[Fel68] William Feller. *An Introduction to Probability Theory and Its Applications*. Wiley, New York, 1968.

[FERN84] Joseph A. Fisher, John R. Ellis, John C. Ruttenberg, and Alexandru Nicolau. Parallel processing: A smart compiler and a dumb machine. In *ACM SIGPLAN Symposium on Compiler Construction*, pages 37–47, 1984.

[FOW87] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems*, 9(3):319–349, July 1987.

[Fox88] Geoffrey C. Fox, et al. *Solving Problems on Concurrent Processors*. Prentice Hall, Englewood Cliffs, New Jersey, 1988.

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

[GJ79] Michael R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman and Company, New York, New York, 1979.

[Gra91] Philippe Granger. Static analysis of linear congruence equalities among variables of a program. In *Proceedings of the Internal Joint Conference on Theory and Practice of Software Development*, pages 169–192, 1991.

[GS90] Thomas Gross and Peter Steenkiste. Structured dataflow analysis for arrays and its use in an optimizing compiler. *Software–Practice and Experience*, 20(2):135–155, February 1990.

[Gup89] Rajiv Gupta. The fuzzy barrier: a mechanism for high-speed synchronization of processors. In *Third International Conference on Architecture Support for Programming Languages and Operating Systems*, pages 54–63, 1989.

[Gup90] Rajiv Gupta. A fresh look at optimizing array bound checking. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages

272–282, 1990.

[GV91] Elana D. Granston and Alexander V. Veidenbaum. Detecting redundant accesses to array data. In *Supercomputing '91*, pages 854–865, 1991.

[Har77] William H. Harrison. Compiler analysis of the value ranges for variables. *IEEE Transactions on Software Engineering*, 3(3):243–250, May 1977.

[HBCM] Michael Hind, Michael Burke, Paul Carini, and Sam Midkiff. Interprocedural array analysis: how much precision do we need? In *Proceedings of the Third Workshop on Compilers for parallel computers, volume 2*.

[Hey91] A.J.G. Hey. The GENESIS distributed memory benchmarks. *Parallel Computing*, 17(10 & 11):1275–1283, 1991.

[HJ91] John L. Hennessy and Norman P. Jouppi. Computer technology and architecture: An evolving interaction. *Computer*, 24:18–29, 1991.

[HKT92] Seema Hiranandani, Ken Kennedy, and Chau-Wen Tseng. Compiling Fortran D for MIMD distributed-memory machines. *Communications of the ACM*, 35(8):66–80, August 1992.

[Jay88] Doddaballapur Narasimha-Murthy Jayasimha. Communication and synchronization in parallel computation. Technical Report CSRD-819, UIUC Center for Supercomputing Research & Development, September 1988.

[Jor78] Harry F. Jordan. A special purpose architecture for finite element analysis. In *Proceedings of the International Conference on Parallel Processing*, pages 263–266, 1978.

[Kar72] Richard M. Karp. Reducibility among combinatorial problems. In R. E. Miller and J. W. Thatcher, editors, *Complexity of Computer Computations*, pages 85–103. Plenum Press, 1972.

[Kar76] Michael Karr. Affine relationships among variables of a program. *Acta Informatica*, 6:133–151, 1976.

[KK91] A. Kallis and D. Klappholz. Extending conventional flow analysis to deal with array references. In U. Banerjee, D. Gelernter, A. Nicolau, and D. Padua, editors, *Languages and Compilers for Parallel Computing–Fourth International Workshop*, pages 251–265. Springer-Verlag, 1991.

[KLS90] Kathleen Knobe, Joan D. Lukas, and Guy L. Steele. Data optimization: Allocation of arrays to reduce communication on SIMD machines. *Journal of Parallel and Distributed Computing*, 8:102–118, 1990.

[Kra93] David Kranz, et al. Integrating message-passing and shared-memory: early experience. In *ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming*, pages 54–63, 1993.

[KS91] V. P. Krothapalli and P. Sadayappan. Removal of redundant dependences in DOACROSS loops with constant dependences. *IEEE Transactions on Parallel and Distributed Systems*, 2(3):281–289, July 1991.

[Kuc81] D. J. Kuck, et al. Dependence graphs and compiler optimizations. In *ACM Symposium on Principles of Programming Languages*, pages 207–218, 1981.

[Kum87] Manoj Kumar. Effect of storage allocation/reclamation methods on parallelism and storage requirements. In *Proceedings of the International Symposium on Computer Architecture*, pages 197–205, 1987.

[Kun82] H.T. Kung. Why systolic architectures? *Computer*, 15(1):37–46, January 1982.

[Lam87] Monica Sin-Ling Lam. A systolic array optimizing compiler. Technical Report CMU-CS-87-187, Carnegie-Mellon Univ. Computer Science Dept., May 1987.

[LAs85] Zhiyuan Li and Walid Abu-sufah. A technique for reducing synchronization overhead in large scale multiprocessors. In *Proceedings of the International Symposium on Computer Architecture*, pages 284–291, 1985.

[LC91] Jingke Li and Marina Chen. The data alignment phase in compiling programs for distributed-memory machines. *Journal of Parallel and Distributed Computing*, 13:213–221, 1991.

[Lei92] F. Thomson Leighton. *Introduction to Parallel Algorithms and Architectures*. Kaufmann, San Mateo, California, 1992.

[Len92] D. Lenoski, et al. The Stanford Dash multiprocessor. *IEEE Transactions on Computers*, 25(3):63–79, March 1992.

[Lim90] Jae S. Lim. *Two-Dimensional Signal and Image Processing*. Prentice Hall, Englewood Cliffs, New Jersey, 1990.

[MAL93] Dror E. Maydan, Saman P. Amarasinghe, and Monica S. Lam. Array data-flow analysis and its use in array privatization. In *ACM Symposium on Principles of Programming Languages*, pages 2–15, 1993.

[MCM82] Victoria Markstein, John Cocke, and Peter Markstein. Optimization of range checking. In *ACM SIGPLAN Symposium on Compiler Construction*, pages 114–119, 1982.

[MCS91] John M. Mellor-Crummey and Michael L. Scott. Synchronization without contention. In *Fourth International Conference on Architecture Support for Programming Languages and Operating Systems*, pages 269–278, 1991.

[MJ81] Steven S. Muchnick and Neil D. Jones. *Program Flow Analysis: Theory and Applications*. Prentice-Hall, Englewood Cliffs, New Jersey, 1981.

[MP87] Samuel P. Midkiff and David A. Padua. Compiler algorithms for synchronization. *IEEE Transactions on Computers*, 36(12):1485–1495, December 1987.

[MP91] Sam P. Midkiff and David A. Padua. A comparison of four synchronization optimization techniques. Technical Report CSRD-1135, UIUC Center for Supercomputing Research & Development, June 1991.

[OD90] Matthew T. O'Keefe and Henry G. Dietz. Hardware barrier synchronization: static barrier MIMD (SBM). In *Proceedings of the International Conference on Parallel Processing, Volume I*, pages 35–42, 1990.

[PKL80] D. A. Padua, D. J. Kuck, and D. H. Lawrie. High-speed multiprocessors and compilation techniques. *IEEE Transactions on Computers*, 29(9):763–776, September 1980.

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

[Pol89] Constantine D. Polychronopolous. Compiler optimizations for enhancing parallelism and their impact on architecture design. *IEEE Transactions on Computers*, 37(8):991–1004, August 1989.

[PW86] David A. Padua and Michael J. Wolfe. Advanced compiler optimizations for supercomputers. *Communications of the ACM*, 29(12):1184–1201, December 1986.

[Rau91] B. R. Rau. Data flow and dependence analysis for instruction level parallelism. In U. Banerjee, D. Gelernter, A. Nicolau, and D. Padua, editors, *Languages and Compilers for Parallel Computing–Fourth International Workshop*, pages 236–250. Springer-Verlag, 1991.

[RL86] John H. Reif and Harry R. Lewis. Efficient symbolic analysis of programs. *Journal of Computer and System Sciences*, 32:280–314, 1986.

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

[RWZ88] Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Global value numbers and redundant computations. In *ACM Symposium on Principles of Programming Languages*, pages 12–27, 1988.

[SA91] Shridhar B. Shukla and Dharma P. Agrawal. Scheduling pipelined communication in distributed memory multiprocessors for real-time applications. In *Proceedings of the International Symposium on Computer Architecture*, pages 222–231, 1991.

[Sad75] R. Sadourny. The dynamics of finite-difference models of the shallow-water equations. *Journal of Atmospheric Science*, 32(4), April 1975.

[Sal90] Joel Saltz, et al. Run-time scheduling and execution of loops on message passing machines. *Journal of Parallel and Distributed Computing*, 8:303–312, 1990.

[Sar87] Vivek Sarkar. Partitioning and scheduling parallel programs for execution on multiprocessors. Technical Report STANFORD CSL-TR-87-328, Stanford

Univ. Computer Systems Lab., April 1987.

[Sch86] Alexander Schrijver. *Theory of Linear and Integer Programming*. Wiley, Chichester, Great Britain, 1986.

[Sei85] Charles L. Seitz. The cosmic cube. *Communications of the ACM*, 28(1):22–33, January 1985.

[SLY89] Zhiyu Shen, Zhiyuan Li, and Pen-Chung Yew. An empirical study on array subscripts and data dependencies. In *Proceedings of the International Conference on Parallel Processing, Volume II*, pages 145–152, 1989.

[Smi78] Burton J. Smith. A pipelined, shared resource MIMD computer. In *Proceedings of the International Conference on Parallel Processing*, pages 6–8, 1978.

[SWG91] Jaswinder Pal Singh, Wolf-Dietrich Weber, and Anoop Gupta. Splash: Stanford parallel applications for shared-memory. Technical Report STANFORD CSL-TR-91-469, Stanford Univ. Computer Systems Lab., April 1991.

[SY89] Hong-Men Su and Pen-Chung Yew. On data synchronization for multiprocessors. In *Proceedings of the International Symposium on Computer Architecture*, pages 416–423, 1989.

[Thi91] Thinking Machines Corporation. The connection machine CM-5 technical summary. Technical report, Thinking Machines Corporation, 1991.

[War93] Steve Ward, et al. The NuMesh: A scalable, modular, 3D interconnect. In *International Conference on Supercomputing*, 1993.

[WF91] Min-You Wu and Geoffrey Fox. Compiling Fortran90 programs for distributed memory MIMD parallel computers. Technical Report CRPC-TR91126, Center for Research on Parallel Computation, January 1991.

[WM91] Ko-Yang Wang and Piyush Mehrotra. Optimizing data synchronizations on distributed memory architectures. In *Proceedings of the International Conference on Parallel Processing, Volume II*, pages 76–82, 1991.

[Wol89] Michael Wolfe. *Optimizing Supercompilers for Supercomputers*. MIT Press, Cambridge, Massachusetts, 1989.

[Wol92] Michael Wolfe. Beyond induction variables. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 162–174, 1992.

[WZ91] Mark N. Wegman and F. Kenneth Zadeck. Constant propagation with conditional branches. *ACM Transactions on Programming Languages and Systems*, 13(2):181–210, April 1991.

[YA93] Donald Yeung and Anant Agarwal. Experience with fine-grain synchronization on MIMD machines for preconditioned conjugate gradient. In *ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming*, pages 187–197, 1993.

[YTL87] P. C. Yew, N. F. Tzeng, and D. H. Lawrie. Distributing hot-spot addressing in large scale multiprocessors. *IEEE Transactions on Computers*, 36(4), April 1987.

[ZDO90] Abderrazek Zaafrani, Henry G. Dietz, and Matthew T. O'Keefe. Static scheduling for barrier MIMD architectures. In *Proceedings of the International Conference on Parallel Processing, Volume I*, pages 187–194, 1990.

Index of terms

ancestor 32
anti-dependence 19
back edge 31
cascade 121
child 32
composite partitioning function ... 82
composite partitioning set 82
control dependence 81
control flow graph 30
cross edge 31
data direction vectors 57
definition-use graph 40
descendant 32
DOACROSS loop 16
DOALL loop 15
dominance 31
enclose 32
execution before 86
execution ordering 85
flow dependence 19
forward edge 30
forward substitution 35
invariance 91
lattice 36
lexical dependence 118
linear integer sequence 44
loop iteration space 73
loop partition stride 94
loop partitioning function 80
loop-carried 57
loop-independent 57
multiple integer lattice 37
output-dependence 19
parent 32
partial ordering 32
partition 82
partitioning set 82
post-dominance 31
precedence 31
predecessor 31
private variable 61
processor partitioning function 82
processor target function 89
pseudo-polynomial 119
pseudo-redundance 120
redundance 117
relative dominance 31
single assignment 40
single integer lattice 36
sink 54
source 54
spatial coordinate 73
spatial relationship 67
statement instance 73
static single assignment 40
subarray 48
successor 31
synchronization assertion 70
synchronization check 70
temporal coordinate 73
temporal ordering 74
temporal relationship 68
temporal target function 89
timestamp 73
value set 36

Index of notation

$\mathcal{A}$ . . . 36
B . . . 123
$\mathcal{D}$ . . . 120
Dec . . . 51
$defGen$ . . . 49
$defIn$ . . . 49
$defKill$ . . . 49
$defOut$ . . . 49
Dom . . . 78
$\mathcal{E}$ . . . 123
$\mathcal{F}$ . . . 129
$\mathcal{L}_{\supset}$ . . . 46
$\mathcal{L}_{\subset}$ . . . 46
$\mathcal{L}'$ . . . 45
$\mathcal{L}'_{\supset}$ . . . 46
$\mathcal{L}'_{\subset}$ . . . 46
$\mathcal{M}_{\supset}$ . . . 49
$\mathcal{M}_{\subset}$ . . . 49
$markExt$ . . . 51
$\mathcal{R}$ . . . 121
$\mathcal{P}_{\Delta}$ . . . 89
$\mathcal{S}$ . . . 48
$\mathcal{T}_{\Delta}$ . . . 89
Tem . . . 73
$\widehat{\mathrm{T}}$ . . . 89
$\mathcal{Z}$ . . . 105
δ . . . 55
Δ . . . 89
$\overline{\delta}$ . . . 19
δ^f . . . 19
δ^o . . . 19
κ . . . 82
K . . . 82
$\mathbf{K}$ . . . 82
λ . . . 94
ξ . . . 97
ϕ . . . 80
Φ . . . 84
σ . . . 78
$\vec{\boldsymbol{\tau}}$ . . . 74
T . . . 97
τ^i_j . . . 90
θ . . . 91
χ . . . 93
Ψ . . . 82
ψ . . . 82
$\vec{\boldsymbol{\omega}}$ . . . 73
Ω . . . 77
ω^i_j . . . 76
$\circledin$ . . . 94
$\circledcap$ . . . 82
$\perp_c$ . . . 36
$\perp_{is}$ . . . 45
$\perp_{iv}$ . . . 38
$\top_c$ . . . 36
$\top_{is}$ . . . 45
$\top_{iv}$ . . . 38
$*$ . . . 77
$\uparrow$ . . . 74
$[\ldots]$ . . . 94
$\langle\ldots\rangle$ . . . 73
$\langle\!\langle\ldots\rangle\!\rangle$ . . . 44
$|$ . . . 76
$\|$ . . . 100
$\prec$ . . . 74
$\succ$ . . . 74