INFORMATION TO USERS

While the most advanced technology has been used to photograph and reproduce this manuscript, the quality of the reproduction is heavily dependent upon the quality of the material submitted. For example:

- Manuscript pages may have indistinct print. In such cases, the best available copy has been filmed.

- Manuscripts may not always be complete. In such cases, a note will indicate that it is not possible to obtain missing pages.

- Copyrighted material may have been removed from the manuscript. In such cases, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, and charts) are photographed by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps. Each oversize page is also filmed as one exposure and is available, for an additional charge, as a standard 35mm slide or as a 17"x 23" black and white photographic print.

Most photographs reproduce acceptably on positive microfilm or microfiche but lack the clarity on xerographic copies made from the microfilm. For an additional charge, 35mm slides of 6"x 9" black and white photographic prints are available for any photographs or illustrations that cannot be reproduced satisfactorily by xerography.
A global approach to detection of parallelism

Callahan, Charles David, II, Ph.D.

Rice University, 1987
RICE UNIVERSITY

A GLOBAL APPROACH TO DETECTION OF PARALLELISM

by

Charles David Callahan II

THESIS SUBMITTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE

DOCTOR OF PHILOSOPHY

APPROVED, THESIS COMMITTEE:

[Signatures and names of committee members]

Houston, Texas
March, 1987
A GLOBAL APPROACH TO
DETECTION OF PARALLELISM

Abstract

Modern computers will increasingly rely on parallelism to achieve high computation rates. Techniques to automatically detect and exploit parallelism have been shown effective for computers with vector capabilities. To employ similar techniques for asynchronous multiprocessor machines, the analysis and transformations used for vectorization must be extended to apply to entire programs rather than single loops. Three subproblems are addressed.

A sequential to parallel conversion techniques is presented. This algorithm, called a parallel code generator, is suitable for conversion of entire subroutines to parallel form. This algorithm is shown to be optimal for a restrictive form of the conversion problem. Additional transformations can be added to the basic parallel code generator. Loop interchange is added to the conversion problem but it is shown that finding the optimal solution is then NP-complete.

The presence of loop carried dependences results in less efficient parallel code. Loop alignment is a general tool for removing loop carried dependences and improving the effectiveness of the parallel code generator. Loop alignment is hampered by alignment conflicts. A transformation called code replication can be used to break alignment conflicts at the cost of additional computation in the output program. It is shown that minimizing the amount of code replication is NP-hard and potentially exponential.

Calls to external routines are also a problem for the parallel code generator. A method is developed to summarize the effects of external routines in such a way as to allow effective dependence analysis to be performed around the call site.

The ability to distribute shared arrays over local memories is important to mask long global memory access times. A general algorithm is presented based on data dependence analysis and alignment conflicts. This techniques is integrated into the parallel code generator.
Acknowledgements

I would like to thank the many people that have made this dissertation possible. My committee, especially Dr. Kennedy, for guidance and support. The IBM Corporation for its support of this research and other projects in the Computer Science Department. Both the IBM Corporation and AT&T Bell Labs for their direct support through graduate fellowships. The faculty and graduate students in LCSE, especially Randy Allen, and those associated with the PFC project, who have made the work enjoyable and provided the basis for the implementation of some of the work of this dissertation. My family for its support, and especially my wife Diana, for emotional support, editorial assistance, and patience.
# Table Of Contents

1 Introduction and Overview .............................................. 1
   1.1 Introduction .................................................. 1
   1.2 Dependence Analysis ........................................... 5
   1.3 Transformations to exploit parallelism ....................... 11
   1.4 Research Overview ............................................ 16

2 A Global Parallel Code Generator ................................. 21
   2.1 Introduction .................................................. 21
   2.2 Data Dependences ............................................. 25
   2.3 Greedy Code Generator ........................................ 26
   2.4 Modifications for the Multi-Loop Case. ...................... 31
   2.5 Loop Interchange .............................................. 35
   2.6 Using Loop Interchange ....................................... 42
   2.7 Loop Fusion with Loop Interchange .......................... 42
   2.8 Control Dependence ........................................... 43
   2.9 Summary ........................................................ 44

3 Loop Alignment and Code Replication .............................. 45
   3.1 Introduction .................................................. 45
   3.2 Loop Alignment ............................................... 47
   3.3 Code Replication .............................................. 59
   3.4 Minimal Alignable Path Covers ............................... 72
   3.5 Using Code Replication ....................................... 86
   3.6 Transformation Preventing Dependences ..................... 89
   3.7 Summary ........................................................ 92

4 Interprocedural Data Dependence Analysis ......................... 93
   4.1 Introduction .................................................. 93
   4.2 Summarizing Array Accesses ................................ 95
   4.3 Regular Sections .............................................. 102
   4.4 Merging Standard RSD's ...................................... 108
   4.5 Translation .................................................... 109
   4.6 A Data-flow Problem ......................................... 112
   4.7 Restrictions ................................................... 114
List of Figures

1 Fork-join Parallelism ........................................... 22
2 Pushing a serial region inside a loop. ......................... 24
3 Basic Translator .................................................. 26
4 Basic Translator Subroutines .................................... 26
5 Loop Interchange ................................................. 31
6 Fusibility test for Code Generator for Nested Loops .......... 32
7 Strongly connected regions with multiple parallel levels. .... 36
8 Example “clause subgraph” ....................................... 37
9 Example seven partition and eight partitions ................... 39
10 Test for alignment conflicts ..................................... 58
11 Test for valid fusion with loop interchange. ................. 59
12 Algorithm 1. Compute alignable path cover. ................ 70
13 Example of exponential growth during replication .......... 72
14 Example of quadratic growth during replication ............. 73
15 Example graph from class $S$ ................................... 75
16 Nodes $v_i,v_j$ and $w_{i,j}$ in $G_B$ ........................... 79
17 Nodes $v_i,v_j$ and $v_k$ and part of a factor node set in $G_B$ for $F_i = v_i + v_j + v_k$ 80
18 Subgraph of $G'$, path cover for $G_f$ ......................... 83
19 A subgraph of $G'$ ............................................. 83
20 A subgraph of $G'$, path cover of $G_B$ ....................... 85
21 Algorithm to eliminate alignment preventing dependences at level $k$. 90
22 Lattice of Array Access Information ........................ 94
23 Examples for Theorem 4.2: $\vec{b} \in A(R_2), \vec{c} \not\in A(R_1)$ ......................................................................................................................... 119
24 Examples for Theorem 4.3: $j = 1, k = 2$ ..................... 123
25 Examples of Distribution Functions for array $A$. ........ 133
26 Distribution Examples ............................................. 138
27 Sketch of Distribution Algorithm .............................. 143
28 Effects of merge for edge $e = (v,w): z = \text{weight}(e) - k + l$ 145
Chapter 1: 
Introduction and Overview

1.1 Introduction

Over the last decade, machine architectures designed to exploit parallelism have been introduced to meet the growing need for processing power. Early examples of this trend are the ILLIAC-IV array processor [BBKK 68], and the pipelined vector processor Star-100 built by CDC. More recent and more successful are pipelined machines, such as the Cray series [Russ 78]. Very recent architectures include a number of less expensive computers with multiple processors, such as the Sequent Balance 8000, the Elxsi System 6400, the Flexible Flex/32, the Intel iPSC, and the Alliant FX/8 [Elec 85]. Many of these machines also support pipelined vector operations within each processor. While it is still too early to judge the success of these machines, the number of commercial vendors suggests a large and growing market. As a part of this trend, prototype machines, such as IBM's RP3 [PBGH], are being constructed with a much larger number of processors.

Effective programming of parallel machines is significantly harder than effective programming of sequential machines. Not only are there all the efficiency concerns of the sequential machine, but additional problems unique to a multiprocessor environment are present. The programming task is further compounded when each processor in the machine is pipelined or supports high speed vector operations. Some programming concerns specific to multiprocessors are:

- **Efficient Use of Hardware** It is imperative that a problem be broken into enough subproblems so that every processor is busy most of the time. If large areas of a problem are serial, the advantages of a parallel machine are wasted. Therefore, finding parallelism in a problem in order to fully exploit parallelism in the architecture is an important programming concern.

- **Effective Use of Memory** Most modern computers have several tiers of memory — each outer tier is less expensive to build, but slower to access. Parallel architectures have similar tiers; in particular, there is often a major distinction between *local private* and *global shared* memory (the latter being significantly slower to access). When to transfer data between these two tiers is a problem. Data can be stored in shared memory for communications purposes, but that shared memory may be one to two
orders of magnitude slower to access [PBGH]: so it is desirable to cache as much data in local memory as possible. The programmer must determine when references to local copies of data will yield the same value as if the data had been kept in shared memory.

- **Correctness** Read and write access to shared memory presents a problem similar to the concurrent access problems discussed in the literature on concurrent processing [BenA 82]. It is not desirable to place every global memory access into some form of synchronization construct, such as a conditional critical region or monitor [BenA 82]. It would be prohibitively expensive to have a large number of accesses guarded by these constructs. Thus, a burden is put on the programmer to use explicit synchronization to avoid race conditions on the variables placed in shared memory.

- **Synchronization** The issue of synchronization is related to correctness; a synchronization point, where several processes are held at the same point in the program, is the key to avoiding critical regions. Synchronization points force a partial ordering on references to memory and so allow programmers and compilers to reason about the run-time behaviour of parallel programs. However, synchronization can cripple the smooth execution of a parallel machine. In fact, it ultimately limits the effective speedup of a parallel machine over a sequential machine [FlKe 85, Cytr 85]. The problem for the programmer is deciding when a synchronization point is necessary, which processors are involved, what data needs to be written before the synchronization point, and which data must be read after it.

These four areas illustrate some of the complexity involved in effectively utilizing modern parallel machines. When the entire program is considered, this complexity becomes enormous, underscoring the importance of developing analysis tools to collect and display relevant information for programmers. This point was stressed in the conclusions of [LFHM 85].

This research is aimed at this need: the result is a transformation system that can automatically translate programs written for a sequential machine into efficient programs for parallel computers, especially in the case of translating numerical and scientific programs for a multi-processor with 64-1024 processors, global shared memory, and local private memories. The only hardware synchronization assumed for this research is concurrent reads and writes provided by the global shared memory and the "fetch-and-add" operator
described in [GGKM 84]. This operation is not used explicitly, but is assumed so that higher level language constructs described later can be implemented efficiently.

Source programs written in sequential form have improved portability and the analysis needed, as compared with programs with explicit parallelism does not significantly increase. Regardless of how "high level" a programming language is used, a powerful system of static analysis will be needed to generate efficient and correct machine code. Low level constructs (e.g., semaphores [BenA 82], asynchronous variables [Jord 85], or "fetch-and-add" operators) are too complex for programmers to verify the program correctness (e.g., absence of race conditions and possible deadlock [BenA 82]) without an analysis tool. Further, low level constructs require more effort when porting the program to a machine with different characteristics.

The need for an analysis tool is not mitigated if slightly higher level constructs (e.g. parallel DO, serial blocks, or barrier synchronization points [LuOv 85] [SDNP 85]) are used, since these constructs also allow race conditions, though certain other problems (e.g., deadlock) are avoided. The use of intelligent compilers to help programmers verify correctness of programs is preceded in [FoOs 76].

Efficient implementation of high level constructs (e.g. monitors and conditional critical regions [BenA 82]) will require analysis of the program to determine when simple access is safe. For instance, if an array is shared so that accesses to the array must be protected inside a monitor, then it may be possible to analyze the program and determine that a simple read or write is safe for a particular reference. Programs discussed in the literature that utilize these constructs do not really need to have synchronization optimized because they invoke that synchronization relatively infrequently. However, processes working on numerical and scientific problems will need to be more tightly coupled; therefore, optimization of synchronization is an important issue. Cytron [Cytr 86] argues that:

... Finally, optimization and program transformations require dependence analysis and parallelization techniques to improve the performance of parallel programs... Rather than eliminating the need of parallelization and dependence analysis, the advent of parallel programming languages has amplified the importance of these techniques by creating these new areas of application.

It is not likely that a fully automatic transformer will be able to transform sequential programs into efficient machine code for parallel machines. A more probable system is
one in which the programmer and an interactive compiler work together to develop an efficient program. The arguments above suggest that such a system, even if it allows the programmer to explicitly describe parallelism, will need all the machinery used in the sequential to parallel transformer. Since no standard parallel constructs have been accepted, it seems reasonable to pursue the sequential to parallel transformation.

Extensive work has already been done in the area of program translation from sequential machines to parallel machines. This research has centered around the use of data dependence analysis to detect DO loops that can be executed either as a sequence of vector statements or with all iterations executed concurrently. Section 1.2 discusses the analysis used; section 1.3 provides examples of its use.

A primary characteristic of the analysis and transformational techniques developed before this dissertation is that they are local to a small section of a program (in general restricted to a single loop nest). The need for analysis to extend across procedure boundaries is clear in the case of a multiprocessor where synchronization needs to be minimized. We want to be able to execute the separate iterations of a loop in parallel even when the body of the loop contains an invocation of an external routine. The reduced synchronization will result in greater speedup. These problems will be discussed more fully in section 1.4.

The specific research reported here extends the local techniques to global (i.e., the entire program): development of a parallel code generator that is effective for entire subroutines including calls to other subroutines and can exploit a memory hierarchy provided on some parallel machines.
1.2 Dependence Analysis

1.2.1 Basics of Data Dependence

Data dependence analysis is a technique used to determine constraints on the execution order of statements in a program (based on the flow of data required during execution of the program). The result of data dependence analysis is a conservative set of dependences between statements in a program. These can be classified into three types of dependences: true dependences, anti-dependences, and output dependences[Kuck 78].

If statement $S_1$ computes a value used by statement $S_2$, then $S_2$ is said to have a true dependence on $S_1$. True dependences are related to definition-use chains of classical data flow analysis [Kenn 81]. They often are interpreted as a requirement that the value producing statement be executed before the value consuming statement. However, true dependences do not reveal all execution constraints. If statement $S_1$ uses a value stored in memory location $M$, and statement $S_2$, executed after $S_1$, modifies the value stored at $M$, then there is said to be an anti-dependence of $S_2$ on $S_1$. An output dependence occurs between two statements that modify the same memory location $M$. As for true dependences, if one statement has an output or anti-dependence on another statement, then the former must be executed after the latter to guarantee that all values used and defined are the same as in the original program. Both output and anti-dependences are generated by reuse of memory and can be broken safely by careful introduction of new variables.

Chapter 5 requires a notion of input dependence[Kuck 78] to complete this set; if statement $S_1$ uses a value stored in memory location $M$ and statement $S_2$ executed after $S_1$ also uses a value stored in that location, then there is an input dependence from $S_1$ to $S_2$. Input dependences are not required for the first 4 chapters and so the definitions given below explicitly exclude this case from consideration.

1.2.2 Data Dependences in Inductive Loops

To sharpen the precision of dependence analysis in the presence of subscripted variables and inductive loops, precise terminology has been developed to describe dependences between statements in loops. For example, in the loop:
DO I = 1, N
    A(I) = B(I)*2
    C(I) = A(I+1)
ENDDO

the value stored into A during each iteration is never used by the statement that assigns to C on any iteration of the loop, so there should not be a true dependence of the first statement on the second. However, since the memory location referenced by A in the second statement on a particular iteration is modified by the first statement on the next iteration, there should be an anti-dependence of the first statement on the second. Finally, no memory location is modified more than once so there are no output dependences.

Since statements in a loop represent a much larger set of statements indexed by the iterations of the loops, techniques and terminology have been developed to describe these sets of statements. For a given set of \( k \) nested inductive loops, numbered from outermost to innermost, if the loop control variable of the \( i^{th} \) loop assumes the values from 1 to \( N^i \) in that order, than the iteration space of the loop nest is the set:

\[
\{ \vec{i} = (j^1, \ldots, j^k) \mid 1 \leq j^i \leq N^i \text{ for } 1 \leq i \leq k \}.
\]

The elements of the iteration space correspond to the values of the loop control variables on successive executions of the body of the innermost loop. Any particular element of the iteration space is called an iteration vector. An execution ordering of a nest of inductive loops is a total order of the set \( \{ \langle \vec{i}, S_j \rangle \} \) of statement instances where \( \vec{i} \) is an element of the iteration space and \( S_j \) is a statement in the body of the loop nest. The initial execution order is given by lexically ordering this set based on the lexical order of the iteration space and the ordering of statements by the rule: \( S_i < S_j \) if \( S_i \) appears textually before \( S_j \) in the body of the loop.

A data dependence of statement \( S_k \) on statement \( S_j \) exists if and only if for some iteration vectors \( \vec{i}_1 \) and \( \vec{i}_2 \), \( \langle \vec{i}_1, S_j \rangle \leq \langle \vec{i}_2, S_k \rangle \) in the execution order of the enclosing loops, on iteration \( \vec{i}_1 \), \( S_j \) references some memory location \( M \), on iteration \( \vec{i}_2 \), \( S_k \) references the same memory location \( M \), and at least one of them modifies that location. When a dependence exists, we associate a distance vector \cite{Lamp74} defined as the pairwise difference of \( \vec{i}_2 \) and \( \vec{i}_1 \):

\[
\vec{i}_2 - \vec{i}_1 = (i^2_1 - i^1_1, \ldots, i^2_k - i^1_k)
\]
with that dependence. The direction vector [Wolf 82] can be constructed from the
distance vector by replacing each 0 component with '='; each negative component with
'>' and each positive component with '<'. Note that if the direction vector has any
components that are not equal to '=', then the first such component from the left must
be a '<' by definition of the initial execution order.

Data dependences can be classified as either loop carried or loop independent
[Alle 83]. A loop carried dependence is not affected by reordering transformations which
preserve the initial ordering of the iteration space, that is, for particular \( \vec{t}_1 \) and \( \vec{t}_2 \), if
\[ \langle \vec{t}_1, S_j \rangle < \langle \vec{t}_2, S_k \rangle \] for all \( S_j \) and \( S_k \) in the initial execution order, then the same holds
in the reordered execution order. Thus, loop carried dependences are preserved by any
transformation that simply rearranges the order in which statements appear in the body
of the loop. A loop carried dependence is characterized by a direction vector with at
least one component that is not '='. The value of the leftmost non-zero component of
the distance vector of a loop carried dependence is called the threshold [Kenn 80] of the
dependence.

On the other hand, a loop independent dependence is preserved by transformations
that preserve the original order of the statements in the loop body. In particular, transfor-
mations applied to a single loop that only change the ordering of the iteration space
(and hence to the execution order) will preserve loop independent dependences. A loop
independent dependence is characterized by a direction vector in which every component
is '='.

Each loop carried dependence is associated with a particular loop, which is said to
carry that dependence. Intuitively, a loop carried dependence exists because the body of
the loop is executed more than once: if the loop was not present, the dependence would
not exist. This is in contrast to the loop independent dependence which exists regardless
of the loops structure around the body of the loop. The level of a loop carried dependence
is equal to the position of the leftmost component of the distance vector that is not equal
to '='.

A data dependence is said to be backward if the sink of the statement appears
textually before the source of the dependence. This includes the case where the source
and the sink are the same statement. A dependence that is not backward is forward.
Note that loop independent dependences must be forward dependences.
1.2.3 Control Dependence

Data dependences do not capture all of the execution constraints of a program. In particular, the flow of control through a program affects which statements are actually executed, and the execution of one statement can affect whether or not another statement is executed. For instance, in the following code, the execution of the second statement depends on whether the goto in the first statement is taken.

\[
\begin{align*}
\text{IF (I .EQ. 0) GOTO 100} \\
\text{B = B/ I} \\
\text{100 CONTINUE}
\end{align*}
\]

The second statement is said to be control dependent [Kuck 78] on the first statement.

Ad hoc methods can be used to modify some of the transformations discussed in the next section to handle loops with internal branches. This approach can be seen in the transformations discussed in [Wolf 82]. Backward gotos form loops and Wolfe also discusses techniques to vectorize such loops as well as DO loops with exit branches (i.e., from inside the loop to outside of the loop).

A process called IF-Conversion[AKPW 83] systematically removes all control flow in a program by replacing forward branches by guards on the statements they affect and backward branches by guards and explicit WHILE loops. This process effectively converts control dependences into data dependences, making them amenable to the data dependence analysis described above. This technique is particularly appropriate for vectorization since vector operations controlled by a mask can be discovered easily. In [FeOt 83] and [FeOW 84] a technique to determine control dependences is given, based on the postdominator relation between nodes in the program flow graph.

If-Conversion is undesirable for a multiprocessor for two reasons: the process makes the dependence analysis phase more expensive, and the boolean guards cause a loss of efficiency during execution. Furthermore, If-Conversion is partially unnecessary for detection of parallel DO loops since forward branches introduce only loop independent dependences, and exit branches prevent parallel execution. If-Conversion might be useful if applied to a small area to allow more freedom for loop alignment (Chapter 3).
1.2.4 Determining Data Dependences

Much research has been done to accurately determine the data dependences between statements [Bane 76] [Bane 79] [Kenn 80] [Kuhn 80] [Wolf 82] [Alle 83] [Trio 86]. The basic question is: given two array references and information about the loops that contain them, do these references ever refer to the same storage location? The earliest approaches to this problem were ad hoc, and based primarily on pattern matching. For example: if the subscripts of the two references are identical and composed only of loop control variables and expressions that are invariant in all containing loops, then there are only loop independent dependences between the statements containing these references.

More analytical approaches treat the subscripts as functions of the loop iteration vectors into a linear set of memory locations. Two array references \( A(f(\vec{i})) \) and \( A(g(\vec{j})) \) generate a data dependence if, and only if, for some vectors \( \vec{i} \) and \( \vec{j} \) in the iteration space, we have:

\[
f(\vec{i}) - g(\vec{j}) = 0
\]  

(1)

This problem is undecidable [Bern 66] for arbitrary functions \( f \) and \( g \). However, Knuth [Knut 71] showed that expressions in general (and hence subscript expressions) are simple, making it is reasonable to place restrictions on \( f \) and \( g \). The common restriction is that they be affine functions in the loop control variables:

\[
f(\vec{i}) = a_0 + a_1 \cdot i^1 + \cdots + a_k \cdot i^k
\]

where each \( a_j \) is constant. When both references are enclosed in the same \( k \) loops, equation (1) translates to:

\[
(a_0 - b_0) + (a_1 \cdot i^1 - b_1 \cdot j^1) + \cdots + (a_k \cdot i^k - b_1 \cdot j^k) = 0
\]  

(2)

This restriction seems particularly acceptable for scientific and numerical programs where regular access is common.

With this restriction, powerful and accurate tests can be applied to determine if a dependence exists. Two important test are the gcd test [Coha 73] [Bane 76] and Banerjee's Inequality [Bane 76] [Kenn 80]. The former tests whether the greatest common divisor of certain combinations of the coefficients \( a_i \) and \( b_i \) and the differences \( a_i - b_i \) divides the constant difference \( a_0 - b_0 \). The latter uses information about the upper bounds of
the loops, which determine the iteration space, to test whether it is possible for values of $f(\tilde{r}) - g(\tilde{j})$ to be both positive and negative. If it is not, there is no dependence. Banerjee's inequality can be sharpened to determine direction and distance vectors [Wolf 82].

Another approach to testing for dependences between array references with affine subscripts is to describe the set of vectors that are solutions to equation (2) by a set of linear inequalities and determine whether that set is empty. Kuhn [Kuhn 80] and Triolet [Trio 86] describe data dependence analysis based on this approach.

1.2.5 The Dependence Graph.

We can now discuss a graph which summarizes the dependences in a given program.

**Definition 1.1** Let $D = (S, E)$ be the (layered) dependence graph of a program, where the set $S$ consists of the statements of the program and the set $E$ of directed edges represents the valid dependences between those statements. Each edge is a triplet consisting of the level associated with the dependence and the type of dependence, TRUE, ANTI, or OUTPUT. The endpoints of the edge are statements involved in the dependence such that the sink of the edge depends on the source of the edge.

Note that each edge is completely determined by the ordered pair of statements, the level and the type. Thus there may be only one edge between two statements which represents many solutions to equation (1), all with the same level and type, or there may be many edges between these two statements annotated with different information. Under this convention, we can refer to a data dependence by its level and type without mentioning the actual distance vectors.

If all solutions to the dependence equation associated with a dependence have the same distance vector, that dependence is said to be consistent; otherwise it is called inconsistent. This property will become important in Chapter 4.

Cycles in the dependence graph are important to the detection of parallelism.

**Definition 1.2** A cycle in the graph $D$ is called a recurrence and the level of the recurrence is the minimum of the levels of the edges in the cycle.
1.2.6 Valid Dependences and Program Transformations

The definitions given earlier for data dependences are asymmetric in the roles of the two statements or memory references. We can view dependences as constraints on the transformations that we perform. In particular, if the execution order of two statement instances is reversed, any data dependence between them will change direction in the dependence graph. In such a case, we say the dependence is reversed; otherwise it is preserved. This motivates the definition:

**Definition 1.3** A dependence is valid if it is a dependence in the initial execution order of the program; otherwise it is called invalid.

Valid and invalid are used interchangably with plausible and implausible[BuCy 86a].

A reordering transformation [Aile 83] is any transformation on the loop nest that changes only the execution order of the loop nest. When reordering transformations modify only the order of the iteration space or the order of the statements in the loop body, the sole requirement for the existence of a data dependence that can be affected is that \( \langle t_1, s_j \rangle \leq \langle t_2, s_k \rangle \) in the execution order of the loop.

**Definition 1.4** A reordering transformation is valid if valid data dependences are preserved.

A valid reordering transformation will not change the operational semantics of a program.

1.3 Transformations to exploit parallelism

This section describes some of the transformations and tools used to exploit parallelism in a program. Section 1.3.1 discusses the basic transformation for exploiting parallelism, section 1.3.2 describes techniques to extract parallelism from programs, and section 1.3.3 describes the impact of procedure calls and interprocedural information on these transformations.
1.3.1 Exploiting Parallelism

Lamport [Lamp 74] describes two techniques to transform a sequential program into a program with explicit parallelism. The hyperplane technique finds a linear transformation of the iteration space to a new iteration space such that all data dependences carried by some of the loops become loop independent. Those loop iterations can then be executed concurrently. The primary basis for this transformation is the following theorem.

**Theorem 1.1** If a loop at level k carries no data dependences, and there are no transfers of control from inside the loop to outside, then the separate iterations of the loop at level k can be executed concurrently.

This is assumed by Lamport and provides the primary goal of some of the transformations described in the next section. Lamport expressed the parallelism by replacing the \( n - k \) innermost loops with the following construct:

\[
\text{DO CONC FOR ALL } (J^{k+1}, \ldots, J^n) \in S \\
\text{loop body} \\
\text{ENDDO}
\]

where \( S \) was the resulting iteration space of the innermost loops after the transformation. A more direct application of the above theorem is to look for specific loops in any nest of loops that can be parallelized, and not to attempt the major transformations on the loop bounds that Lamport describes.

The second technique described by Lamport is called the coordinate method. Where the hyperplane method is appropriate for a multiprocessor machine without interprocessor communication, this method is appropriate for an array processor with the processing elements operating in lockstep. The basic parallelism construct Lamport used was the \( \text{DO SIM} \) loop which requires that all iterations of a particular statement in the loop body be executed simultaneously. The requirement for a loop to be a valid \( \text{DO SIM} \) loop is summarized in the following theorem.

**Theorem 1.2** An innermost set of loops can be executed as a \( \text{DO SIM} \) if all dependences carried by these loops are forward dependences and there are no transfers of control out of the loop or backward inside the loop.

where a forward dependence is one in which the source of the dependence appears textually before the sink of the dependence. Note that single statement true dependences are not forward dependences.
The realization of a DO SIM on a pipelined machine is as a sequence of vector operations. Wolfe [Wolf 82] notes that many of these vector machines have instructions to perform various forms of sum reductions and first order linear recurrences, and thus some single statement true dependences can be "vectorized" when translating for these machines.

1.3.2 Transformations to Enhance Parallelism

Both of the previous theorems indicate when a particular loop can be parallelized. In these sections, several techniques will be described to massage a loop that does not satisfy one of these theorems into one that does. Most of these transformations were developed for use in a vectorizing compiler but all will find application in the parallelizing transformer described in Chapter 2.

- **Statement Reordering** Backward dependences can be converted into forward dependences by rearranging the order of the statements inside the loop. Allen [Alle 83] shows that such reordering preserves loop carried dependences; as long as there are no cycles in the dependence graph, statement reordering can be used to convert all backward dependences into forward dependences.

- **Loop Distribution** Statements that can be vectorized can be separated from those that cannot by splitting the loop into two or more loops with the same loop header. This transformation is valid if both endpoints of any backward dependences end up in the same loop. Again, statement reordering can be used to improve the effectiveness of loop distribution. Since a cycle in the dependence graph must include a back edge, the strongly connected regions of the dependence graph (also called II-blocks) form the basic units that cannot be broken up via loop distribution.

- **Scalar Expansion** Definitions of scalar variables tend to inhibit vectorization by generating single statement loop carried output and anti-dependences. If the scalar is used in the loop, it also generates backward dependences which inhibit loop distribution. A scalar, \( T \), can be expanded into an array by replacing each occurrence of \( T \) after the first assignment to \( T \) in the loop with a reference to the new array subscripted by the normalized loop control variable. For uses that occur before the first assignment to \( T \)
in the loop, the array is subscripted by the normalized loop control variable minus one. This transformation causes assignments to the expanded variable on separate iterations to modify different memory locations thereby breaking anti and output dependences. If there is only one unguarded assignment to the scalar, then all output and anti-dependences carried by that scalar will be removed. This technique can be generalized to arrays, but the cost in additional memory is generally assumed to be prohibitive.

- **Variable Renaming** Output and anti-dependences are generated by the reuse of memory. Scalar expansion eliminates some reuse of memory and, accordingly, some output and anti-dependences. Replacing some references to a variable \( A \) by references to a new variable \( A' \), can also break additional output and anti-dependences. Kuck et al. [KKPL 80] give an algorithm for renaming scalars.

- **Variable Substitution** Instances of a variable can sometimes be replaced with the expression that most recently defined that variable. This transformation is used to reduce a multistatement cycle to a single statement. The cycle may then be vectorized if it consists of only output or anti-dependences, or if it forms a recurrence that the target architecture can efficiently execute [Wolf 82].

- **Node Splitting** The inverse of variable substitution is called node splitting [Wolf 82]. Node splitting involves breaking the original statement up by computing subexpressions in separate statements and storing their values in new temporary variables. The goal is to give more freedom to statement reordering and possibly to vectorize some of the operations that otherwise would be part of a cycle.

- **Loop Interchange** This powerful transformation is applied to a nest of loops by reordering the nesting levels of the loop headers [Alle 83]. For example in the following loop nest

```plaintext
DO I = 1,N
  DO J = 1,N
    DO K = 1,N
      \textit{loop body}
    ENDDO K
  ENDDO J
ENDDO I
```
if we were to interchange the loops at levels 1 and 3, i.e., the $I$ and $K$ loops, the result would be

\[
\begin{align*}
&\text{DO } K = 1,N \\
&\text{DO } J = 1,N \\
&\quad \text{DO } I = 1,N \\
&\quad \quad \textit{loop body} \\
&\text{ENDDO } I \\
&\text{ENDDO } J \\
&\text{ENDDO } K
\end{align*}
\]

where the body of the loop is unaffected. The primary effect of loop interchange is to change the level associated with the dependences: valid loop interchanges do not affect the shape of the dependence graph. Loop interchange can be used effectively to increase vectorization (by moving dependence to outer levels), or to increase the size of parallel regions (by moving dependences to inner levels).

- **Loop Fusion**  Loop fusion is the inverse of loop distribution. In loop fusion, two or more loops are fused into a single loop. Abu-Sufah [AbuS 78] discusses loop fusion as a technique to improve locality of reference for virtual memory machines. Wolfe [Wolf 82] discusses several uses for loop fusion in the context of vectorization as well as classical uses. Fusing parallel DO loops can have the important effect of eliminating the need for a synchronization point.

- **Summary**  This section has described some techniques used to squeeze parallelism out of sequential programs. For many of these techniques, their effectiveness is limited by two factors. One is that cycles of true dependences represent computations that have, in some sense, a serial nature. Cytron [Cytr 84] characterizes the serial nature of a cycle in terms of how much expected delay occurs between the successive iterations in a DOACROSS. While some improvement might be made in linear recurrences (for instance, replacing a linear time sum reduction with a logarithmic time sum reduction), it seems unlikely that general techniques will be found to handle more than a fraction of these cases. The other limiting factor is lack of accuracy in the dependence analysis. The primary obstacle to accuracy is invocation of external routines.
1.3.3 Interprocedural Information

A subroutine call represents a significant barrier to the precise determination of a data dependence graph, since there is no information at the point of call about how parameters or global variables are used and modified. In these cases, a parallelizing translator must make conservative estimates and assume that every parameter and every global variable is both used and modified by the execution of the external routine. Another aspect of external subroutines is the possibility that two variables in the called routine refer to the same storage location. This is called aliasing. Since a parameter variable could be aliased to any other parameter or to a common block, transformations involving those variables could be unsafe\(^1\). A final problem is the presence of symbolic terms in subscripts. At the local level, constant propagation and induction variable substitution can go a long way toward replacing symbolic terms with constants and affine functions of loop control variables. However, if a formal parameter or a global scalar appears inside a subscript, the accuracy of the dependence analysis may be severely hampered. Though it is often sufficient to know only the sign of a term, this may also be unknown if it depends on a parameter. Interprocedural summary information is discussed in Chapter 4.

1.4 Research Overview

The principal goal of this research was to extend the essentially single-loop nest techniques of parallelization, which are effective for pipelined vector machines, into global techniques suitable for a multiprocessor architecture. To accomplish this, the following areas are discussed in this dissertation:

- A parallel code generation technique suitable for computers with a large number of asynchronous parallel processors is developed in Chapter 2. This code generator uses data dependence analysis across entire subroutines and applies code transformations to a similar sized region. Some of the transformations of section 1.3, in particular loop interchange and loop fusion, are incorporated into the parallel code generator. New transformations, loop alignment and code replication, are developed in Chapter 3 to address potential inadequacies in this parallel code generator.

\(^1\) The Fortran standard forbids modification of an aliased variable, but it seems to be done with sufficient frequency to warrant the concern of ambitious parallelizing compilers.
• Interprocedural flow analysis techniques are presented in Chapter 4 to provide sufficient information for the efficient detection of potential parallel execution of subroutine calls.

• Chapter 5 presents an algorithm to map global data structures onto distributed memory. This algorithm allows the parallel code generator to make effective use of local, private memory when it is available.

1.4.1 A Parallel Code Generator

All of the transformations described in the previous section have focused on the problem of exploiting parallelism at the level of a DO loop. They do not address the problem of translating entire programs or subroutines into parallel form. Chapter 2 of this thesis describes a sequential-to-parallel code generator based on data dependence analysis which examines an entire subroutine to find the execution dependences between statements and loops and exploit any parallelism that is uncovered. For the code generation problem using only loop distribution and loop fusion, this algorithm is shown to minimize the number of barrier synchronization points.

Loop interchange can be employed to enhance the parallelism available in the program. Chapter 2 shows that adding the ability to interchange loops make the problem of finding optimal solutions to the parallel code generation problem NP-complete in the model of parallelism used in Chapter 2.

The algorithm of Chapter 2 can be improved by including additional transformations: loop alignment and code replication. The effectiveness, limitations and cost of these transformations are described in Chapter 3. Loop alignment, a method of reordering the computations in a loop, is shown to parallelize any loop that does not have an alignment conflict. Intuitively, an alignment conflict occurs whenever a value computed in a loop is needed by the same statement on two or more iterations. In such a case, code replication can be used; code replication creates multiple instances of the computation so that they can be individually aligned with each use of the value. Chapter 3 shows that under minor restrictions loop alignment used with code replication can parallelize any loop that does not carry a recurrence. However, code replication may increase the size of the loop body exponentially, and even when this increase is only linear, finding the minimal amount of replication to allow alignment is NP-hard. Chapter 3 shows how to incorporate these two transformations into the parallel code generator of Chapter 2 and suggests heuristics to guide the use of code replication.
1.4.2 Interprocedural Techniques

When the target architecture for a program can efficiently execute vector statements, it is profitable to look for parallelism in source programs at the statement level. In a multi-processor machine, where each processor follows an independent instruction sequence (and a primary bottleneck to efficient use of the machine is synchronization of the processors), it is important to find as large a region of parallelism as possible. Just as inductive loops form the most likely place to find parallelism, external subroutine calls inside such loops become a primary source of large grain parallelism.

The problem is to summarize the effects of executing the external routines in a way that allows parallelism to be detected at the point of call. For example:

```
DO 100 J = K+1,N
   T = A(I,J)
   IF (I.NE.K) THEN
      A(I,J) = A(K,J)
      A(K,J) = T
   ENDIF
   CALL UDAXPY(N-K,T,A(K+1,K),A(K+1,J))
100 CONTINUE
```

Where UDAXPY is similar to DAXPY from LINPACK:

```
SUBROUTINE UDAXPY(N,S,X,Y)
DO 10 I = 1,N
   Y(I) = S*X(I) + Y(I)
10 CONTINUE
RETURN
END
```

Local techniques can determine that the loop inside UDAXPY can be executed in parallel. It is much more desirable to determine that the loop that calls UDAXPY can be executed in parallel.

There are two benefits from finding call sites inside loops where the invocations of the external routine on separate iterations can be executed in parallel. The first is the larger granularity of parallelism that can be found. The speedup obtained by executing the separate instances of the loop inside UDAXPY would be much more dramatic than the speedup obtained by only executing the statements inside the loop of a single instance of the subroutine. The second benefit occurs when we have more processing resources
than iterations in the calling loop. In that case, we could exploit the parallelism inside
the separate instances of the subroutine as well.

To detect such parallelism, Chapter 4 develops methods for efficiently and effectively
summarizing procedures in such a way that allows both precise representation of the
majority of common access patterns (rows, columns, diagonals) and efficient analysis for
dependences. The basic idea is to limit the sections of arrays that will be considered so
as to allow construction of a lattice-based data flow framework suitable for use with one
of the standard interprocedural data flow techniques. The elements of this lattice will
contain information for use with a straightforward generalization of Banerjee's inequality,
thus avoiding the high cost associated with dependence analysis in Triolet's approach [Trio
86].

1.4.3 Distributed Data Structures

Consider the following example of multiplying three $n \times n$ arrays:

\[
\begin{align*}
\text{DO I = 1,N} \\
\text{DO J = 1,N} \\
\text{T(I,J) = 0} \\
\text{DO K = 1,N} \\
\text{T(I,J) = T(I,J) + B(I,K)*C(K,J)} \\
\text{ENDDO} \\
\text{ENDDO} \\
\text{DO I = 1,N} \\
\text{DO J = 1,N} \\
\text{D(I,J) = 0} \\
\text{DO K = 1,N} \\
\text{D(I,J) = D(I,J) + A(I,K)*T(K,J)} \\
\text{ENDDO} \\
\text{ENDDO} \\
\text{ENDDO}
\end{align*}
\]

This code can be parallelized in a straightforward manner by running the two outer loops in
parallel with a barrier synchronization point between them. A significant savings could be
had if we could avoid storing the intermediate array $T$ in global memory. If we parallelize
only the inner loop of the second nest, then an entire row of $T$ is needed by only a single
processor. This suggests that we should have the same processor compute an entire row
and store it in the local memory of that processor. The resulting code would look like:
LOCAL T(N)
DO ALL J = 1,N
  DO I = 1,N
    T(I) = 0
  DO K = 1,N
    T(I) = T(I) + B(I,K)*C(K,J)
  ENDDO
ENDDO
DO I = 1,N
  D(I,J) = 0
  DO K = 1,N
    D(I,J) = D(I,J) + A(I,K)*T(K)
  ENDDO
ENDDO
ENDDO

Here, the outer two loops have been interchanged in the first loop nest and then renamed to allow the loop fusion.

Note that removal of the array temporary would be profitable even for a scalar machine [Wolf 82]. On a parallel machine, the primary advantage may be the reduction in the number of global storage references. An additional advantage arises in removing the barrier synchronization point between the loops.

There are two related problems associated with the above example. The first is the question of when a variable can be held in local storage. In general, this will require that the live range of the variable be contained inside a single parallel DO. This condition can be relaxed if we can insure that the same local memory is associated with each processor, and if we can determine at compile time that iterations of distinct parallel loops will be assigned to processors in such a way that each processor references the same memory locations of an array in both of the loops.

Chapter 5 uses the notion of a distribution function to describe how arrays are to be distributed across local memories. The distribution of an array puts constraints on which processor can execute any particular iteration of a loop accessing that array. Chapter 5 develops a dependence based test for when a collection of variables can be simultaneously distributed over a set of loops without losing parallelism. This test uses the definition of alignment conflict presented in Chapter 3. It is also shown how to incorporate this test into the parallel code generator so that it can apply array distribution where profitable.
Chapter 2:
A Global Parallel Code Generator

2.1 Introduction

Programs for multi-processor jobs can be roughly categorized as homogeneous or non-homogeneous. In the first case, all processes execute exactly the same program and in the other they execute different programs. This notion is necessarily vague but the intent is that the non-homogeneous class contains the programs which follow a primarily client-server model and the homogeneous class contains programs such as numerical and scientific programs where the parallel processors are viewed as a single virtual processor that executes a single program. This chapter discusses the transformation of programs written in a sequential language into the same language augmented with explicit constructs to control parallelism in a homogeneous way: in particular, a program translation algorithm that translates programs written in a FORTRAN like sequential language to a FORTRAN like language with what I will call a medium grain parallel construct.

A medium grain parallel construct is basically a fork-join as illustrated in Figure 1. The circles represent either serial blocks of code that are executed by a single processor or parallel DO loops, each iteration of which is executed by a single processor. Each serial block and each iteration of a parallel DO loop is considered an independent task; tasks are assigned to processors non-deterministically. When a processor finishes one task, it competes for a second until all tasks have been assigned. After all tasks have been assigned, processors are blocked at a barrier synchronization point until all tasks have been completed.

This construct is called medium grain because it allows expressions of parallelism at a finer granularity than the subroutine level, but does not allow expression of parallelism at a finer granularity than a single statement. It also does not allow statements to be synchronized based on the availability of operands, unlike the fine grain parallelism of data flow machines. This construct is used because it allows expression of some very common forms of parallelism found in numerical and scientific programs and is similar to constructs advocated by Lusk and Overbeek [LuOv 85] and by Darema-Rogers, et al. [SDNP 85].
Execution of code that is outside one of these constructs can be interpreted in two ways. Under one interpretation, there is only one processor executing the program. When it reaches a parallel construct, more processors are made available to share the tasks. When these tasks are completed, the additional processors are released and only a single processor continues to execute the current program. This interpretation is suitable for a multiprocessor that time-shares real processors among different jobs and has operating system support for moving processing power from one job to another dynamically. Under this interpretation, the single processor executing outside of a parallel construct may modify both global memory and its own local memory.

Under another interpretation, every processor executes the code that is outside of the parallel constructs. In this case, the primary code that will be outside of parallel constructs is control flow: serial DO loops that contain parallelism, if-then-else branching and external subroutine invocation.

Chapter 4 will address the issue of how to use local memory; this chapter will assume that only global memory is used. The code generator developed in this chapter will output programs whose semantics are the same under both interpretations. This second interpretation will be used in that chapter.

To express the results of the code generation process, the medium grain parallel construct is realized using three lower level constructs:

- **Serial Block.** Indicates that the list of statements inside the serial block are to be executed by only a single processor. It is assumed that the first processor to reach
a serial block is assigned to execute that block, if more than one processor reaches
the block simultaneously, the block is assigned to a processor non-deterministically.
Processors not assigned to execute this block proceed to execute whatever code follows
it. An example of the syntax would be

```
SERIAL
  SUM = 0
  DO I = 1,N
    SUM = SUM + A(I)*SUM
  ENDDO
ENDSERIAL
```

- **Parallel DO.** Similar to the standard FORTRAN DO except that iterations are ex-
  ecuted concurrently and only one processor executes any particular iteration. The
  assignment of iterations to processors is made non-deterministically at execution time.
  When a particular processor finishes an iteration, it returns to the top of the loop and
  competes for any remaining iterations. When no further iterations remain, processors
  proceed to execute whatever code follows the loop. An example would be

```
DOALL I = 1,N
  IF (A(I) .NE. 0) THEN
    B(I) = B(I)/A(I)
    A(I) = 1
  ENDDO
ENDDO
```

- **Barrier Synchronization.** Indicates that all processors must reach this point before
  any processors are allowed to continue past it. An example would be

```
BARRIER
```

A fork-join as illustrated in Figure 1 is realized as a list of SERIAL-ENDSERIAL
blocks and DOALL’s terminated with a BARRIER statement.

The transformation discussed in this chapter can also be viewed as the replacement
of a serial block (initially the entire source routine) by a provably equivalent sequence of
serial blocks and parallel DO loops punctuated with barrier synchronization points. This
view allows the process to applied recursively inside DO loops that are found to be serial
in nature based on the transformation illustrated in Figure 2. This transformation will
be referred to as **pushing a serial region inside a loop.** Note that the post-transformation
loop control variable is assumed to be stored in the private memory associated with each
processor under the interpretation that all processors executed code outside of a SERIAL block or DOALL.

Our goal is to convert as many serial DO loops as possible into parallel loops (a.k.a, parallelizing the loop) to exploit the parallelism in the target machine. The primary constraint on this process is that certain accesses to shared variables must be explicitly serialized to ensure equivalence with the original source program. This requirement can be stated formally as

**(P1)** If two processes access a shared memory location and if at least one of those processes modifies that location, then accesses to that location must be serialized by inserting a barrier synchronization point such that access order of the original program is preserved.

Thus we have three problems: detection of loops that can be parallelized, detection of shared variable access that must be explicitly synchronized, and translation of the original program to parallel form. The approach taken here is to break the original program into serial blocks and parallel loops and then to insert barrier synchronization points as necessary. Whether a transformation is profitable depends strongly on the architecture of the target machine. Since parallel architectures vary considerably, we will adopt optimality conditions that independent of machine size and memory configuration. The initial optimality conditions are:

**(O1)** If a statement inside a DO loop can be executed in parallel, then some loop containing it will be parallelized.

**(O2)** Subject to (O1), the number of barrier synchronization points will be minimized.

These conditions are based on the premises that parallelism is always profitable to exploit and that almost all of the synchronization cost incurred during execution of the output
program is the result of synchronizing processors at barrier synchronization points. Neither of these premises are completely true, but they represent a reasonable ideal approximation to a real machine. The first condition is vague and in general is undecidable; a more precise and effective definition will be given in the next section.

Section 2.2 describes the use of data dependence analysis for this problem. Section 2.3 gives an algorithm based only on loop distribution and re-fusion for solving this problem; this algorithm is shown to be optimal for the single loop case. Section 2.4 to extends the algorithm of section 3 to nested loops. Section 2.5 examines how loop interchange can be used to further improve the transformation and proves that finding an optimal translation is NP-complete when using loop interchange. Section 2.6 describes the heuristic used in the current implementation. Section 2.7 describes how loop fusion, which used implicitly in the parallel code generator, interacts with loop interchange. Section 2.8 describes how control dependences can be handled by the parallel code generator.

2.2 Data Dependences

Data dependence analysis provides the answers to the first two problems. The first of these – determining which loops can be parallelized – is answered by the following theorem: a DO loop can be parallelized if it does not carry any dependences. As the second optimality condition implies, we are not interested in parallelizing loops but in parallelizing statements. When an entire loop cannot be parallelized, it is possible to distribute the loop around the strongly connected regions of the dependence graph. Thus the first problem is related to the notion of a parallel statement used in (O2).

**Definition 2.1** A statement $S$ is a parallel statement if it is enclosed in $k$ loops and for some $i \leq k$, $S$ is not part of a cycle in the dependence graph restricted to loop independent dependences and loop carried dependences carried at levels deeper than $i$. Such a statement is said to be parallel at level $i$.

If statement $S$ is parallel at level $i$, then if we execute the outer $i - 1$ loops serially, then distribute the level $i$ loop so that $S$ is in a loop at level $i$ by itself, then that loop does not carry any dependences and hence can be parallelized. Under this definition, optimality condition (O1) is precise and the optimality of the algorithm is made dependent on the accuracy of the dependence analysis.
We can decompose a program into parallel and serial statements at level 1 by finding the strongly connected regions of the program’s dependence graph. Loop distribution can be used to break loops up so that each strongly connected region is a loop by itself. Each such region either carries a dependence, and hence is serial, or doesn’t carry a dependence and is parallel. These regions are the basic blocks manipulated by the parallel code generator.

For the second problem – synchronizing shared memory accesses – the only memory accesses that must be synchronized are those between which there is a data dependence. In particular, it is not necessary to consider references related in the transitive closure of the dependence graph since the serialization process has a transitive effect. The second problem can be addressed based on where the endpoints of a data dependence are put after the program is broken into serial and parallel regions.

(P2) Both endpoints in the same serial or parallel region. Since only a single processor executes this region, condition (P1) cannot be violated. No synchronization is required.

(P3) One endpoint in a serial region and the other endpoint in a parallel region. In this case, a barrier synchronization point will always be needed between these two regions.

(P4) Endpoints are in different parallel regions. Here a barrier synchronization point will only be needed if the two loops associated with these parallel regions are not fused.

An implication of (P2) is that all dependences contained in a single region can be ignored by the algorithm which orders regions and inserts barrier synchronization points.

2.3 Greedy Code Generator

This section presents an algorithm for solving the parallel code generation problem. Distinguishing parallel from serial regions is based on the dependence analysis discussed in the previous section and Chapter 1. The primary task of the algorithm presented here is to find an ordering of the strongly connected region of the dependence graph that minimizes the number of barrier synchronization points needed between them. The algorithm is shown in Figure 3. The first step is to distinguish serial and parallel regions. From then on it is basically a greedy algorithm: beginning with a region that has no incoming dependences, build a maximal set of serial and parallel regions such that barrier
procedure codegen(S, D, k)
/\* S is a collection of statements and D is the level k de-
/\* pendence graph of S \*/
break S in to strongly connected regions \( P = \{P_1, \ldots P_k\} \)
let \( D_P \) be the dependence graph induced on \( P \) by \( D \)
/\*/ preds(p) is the set of direct predecessors of \( p \) in \( D_P \) \*/
\[ \text{nopreds} \leftarrow \{ p \in P \mid \text{preds}(p) = \emptyset \} \]
\[ \text{visited} \leftarrow \emptyset \]
while not empty(nopreds) do
\[ R \leftarrow \emptyset \]
\[ \text{NotOK} \leftarrow \emptyset \]
while nopreds - NotOK \( \neq \emptyset \) do
\[ \text{remove any node } p \text{ from } \text{nopreds} - \text{NotOK} \]
\[ \text{add } p \text{ to } \text{visited and } R \]
\[ \text{call check_sons}(p) \]
\[ \text{call generate}(R, k) \]
if not empty(nopreds)
then generate a barrier synchronization point

Figure 3. Basic Translator

synchronization points are needed between them. When a point is reached such that no additional region can be included, generate code for this set of regions, generate a barrier synchronization point and repeats.

The third parameter, \( k \), in procedure codegen indicates the nesting level for which the code generator has been called. All loops at nesting levels less than \( k \) will be executed serially and so the data dependences will be preserved by the barrier synchronization point at the bottom of each outer loop (see Figure 2). As discussed above, a cycle of dependences defines a basic set of statements that must be executed as a unit. If no dependence in this cycle is carried at level \( k \), the entire strongly connected region is a parallel region; otherwise it is a serial region. Scalar statements (i.e., statements enclosed in fewer than
procedure check_sons (v)
    for each outgoing dependence e = (v, w) in D_p do
        /* check if v and w must be serialized */
        if both v and w are parallel
            then if v and w cannot be fused or e is loop carried at level k
                then add w to NotOK
        else if either v or w is parallel
            then add w to NotOK
        if preds(w) ⊆ visited
            then add w to nopreds

procedure generate(region,k)
    for each connected region r in region do
        if nodes in r are serial then do
            let D_r be the level k + 1 dependence graph restricted to r
            call codegen(r, D_r, k + 1)
        else fuse nodes in r into a single parallel loop

Figure 4. Basic Translator Subroutines

k loops) are also serial. The main loop is basically a topological sort, modified to delay generating a barrier synchronization point as long as possible.

The set visited is the set of strongly connected regions for which code has already been generated. The set nopreds is the set of strongly connected regions for which code has not been generated but all of whose immediate predecessors in D_p have been visited. Finally, the set NotOk is the set of strongly connected regions that cannot be fused with the current region without violating (P2), (P3) or (P4). Note that this set is reset to empty each time a barrier synchronization point is generated. The subroutine check_sons

1 If the statement modifies only variables stored in the private memory associated with each processor, than the statement can be treated both as a serial and as a parallel region

2 Fusion may also be inhibited by fusion-preventing dependences. See section 2.7.
marks successors of a strongly connected region as NotOk when they can not be fused
with the node p being examined.

The process of generating code includes fusing together connected components so that
rule (P2) applies and recursing on serial regions to look for parallelism at deeper nesting
levels. Many details have been omitted for simplicity: the need to maintain the topological
order as connected components are fused, the mechanics of pushing a serial block inside
a loop as demonstrated in section 1, and correct handling of scalar statements. Also, if
no parallel loops are found deeper, the serial region should be popped back outside of the
loop.

The second optimality constraint, that every parallel statement be in a loop that is
parallelized, is achieved. If a statement S is parallel at level i, then either it is part of
a parallel region at a level less than i, or codegen will be called with a set of statements
including S and level equal to i. In this case, S will be found to be a parallel region and
a parallel loop will be generated around it.

The number of barrier synchronization points needed depends only on the order of
the parallel and serial regions in the sequence generated. The ordering generated by the
above algorithm is one containing the minimal number of barrier synchronization points.
To prove this, we first examine a related graph problem.

Definition 2.2 Let G = (V, E) by a directed acyclic graph and # a
relation called inconsistent defined on V x V. Thus, v#w is read, “v
is inconsistent with w”. A consistent partition of G is a partition
P = P_1,...,P_n of V such that the following two conditions hold for
each v \in P_i and w \in P_j:

- a. If (v, w) \in E then i \leq j
- b. If v#w then i \neq j

The mapping of the code generation problem to the problem of finding consistent
partitions is clear: V is the set of strongly connected regions of the dependence graph
restricted to loop independent edges and loop carried edges that are carried at levels
deeper than k. E is the set of inter-region edges of this restricted dependence graph.
Two nodes are inconsistent if they must be separated by a barrier synchronization point
according to rules (P3) and (P4). A consistent partition is a partial ordering on the
nodes in V that does not violate the partial ordering induced by E and that respects the
requirements of (P3) and (P4).
Any consistent partition for the code generation problem will induce a valid ordering of the parallel and serial regions, and clearly, the greedy algorithm above generates a consistent partition, from now on called the **greedy partition**. The minimal number of barrier synchronization points needed for a code generation problem is thus equal to one less than the number of elements in a consistent partition with the fewest number of elements. I intend to show that the greedy partition has the fewest number of elements and hence minimizes the number of barrier synchronization points.

**Theorem 2.1** The greedy partition of $G$ has a minimal number of elements.

**Proof:** Let $GP = G_1, \ldots, G_n$ be the greedy partition of $G = (V, E)$ and let $P = P_1, \ldots, P_m$ by any minimal consistent partition of $G$. I will show that $n \leq m$ by induction on $||V||$. If $||V|| = 1$, then $n = 1$ and the result is immediate since any partition must have at least one element. Assume that the greedy partition is minimal for all graphs with fewer than $k$ nodes and that $||V|| = k$.

An important property of the greedy algorithm is that $G_2, \ldots, G_n$ is the greedy partition of $G$ restricted to the nodes in $V - G_1$. This follows from the fact that the set $NotOK$ is reset to empty each time a new element of the partition is started: thus the algorithm is effectively applied recursively to the reduced graph and the partition generated for the reduced graph is also a greedy partition.

If $P_1 \subseteq G_1$, then $P_2 - G_1, \ldots, P_m - G_1$ is a consistent partition of $G$ restricted to $V - G_1$. By the induction hypothesis, we know that $G_2, \ldots, G_n$ is minimal for that restricted graph and so $n - 1 \leq m - 1$ and the theorem follows. Otherwise, select $w \in P_1 - G_1$ such that $preds(w) \subseteq G_1$. Since $w$ is not in $G_1$, at some point $w$ was put into the set $NotOK$. This occurs only if there exists $v \in preds(w)$ such that $w \neq v$. By property (a) of a consistent partition, $v$ must be in $P_1$ since $w$ is, but then we have a contradiction of property (b) which prohibits $v$ and $w$ being in the same element of $P$. Therefore $P_1 \subseteq G_1$ and the theorem follows.

**End of Proof.**

A key factor in this proof is the fact $preds(w) \subseteq G_1$ and $w \not\in G_1$ imply that $w$ is inconsistent with at least one of its predecessors. When we add the ability to interchange loop levels, we loose this property and the algorithm is no longer optimal (see section 2.5).
2.4 Modifications for the Multi-Loop Case.

In the presence of nested loops, the simple greedy algorithm is non-optimal due to the fact that recursive regions with deeper parallelism must be treated differently from recursive regions without deeper parallelism. The greedy algorithm minimizes the number of partitions at the outer level but does not minimize the expected number of barrier synchronization points. Consider the loop in Figure 5(a). At the outer level, the greedy algorithm will produce the code shown in Figure 5(b) and after recursing we get the code shown in Figure 5(c). This is non-optimal because the outer loop can be distributed, as shown in Figure 5(d), so that the first barrier synchronization is effectively moved out one nesting level to get a fragment with \( N \) barrier synchronization instead of \( 2N - 1 \) barrier synchronization points. This represents a general problem: fusing serial regions
procedure check_sons(v)
    for each outgoing dependence e = (v, w) in Dp do
        /* check if v and w must be serialized */
        if both v and w are parallel
            then if v and w cannot be fused or e is loop carried at level k
                then add w to NotOK
            else if either v or w is parallel
                then add w to NotOK
            else if neither v nor w is scalar and one is deeper
                then add w to NotOK
        if preds(w) ⊆ visited
            then add w to nopreds

Figure 6. Fusibility test for Code Generator for Nested Loops

together when at least one has parallelism at a deeper level will force a barrier synchronization point inside a loop instead of one at the outer level.

To correctly handle the multi-loop case, each serial region must be examined for deeper parallelism before the decision about which partition to put it in is made. Thus, the first step after breaking a loop into strongly connected regions is to recursively generate parallel code for each serial region that has statements at a deeper nesting level than k. If a serial region, r, has any parallelism deeper, then it is flagged as such: deeper(r) ← true if r has parallelism at a deeper level and deeper(r) ← false otherwise. With the additional information in the deeper flags, we modify the procedure check_sons to prevent fusing deeper regions with other regions (see Figure 6). Note that if a serial region has no statements at a deeper nesting level, it can be fused with a region with deeper parallelism, since the barrier synchronization between them is at the outermost nesting level. A serial region with no statements at a deeper nesting level is said to be scalar.

Finally, the code to fuse connected regions must be modified. If two deeper regions are in the same partition, they must be completely independent. They can be merged together to share barrier synchronization points. For example, the two loops
DO I = 1,N
    DOALL J = 1,N
        A(I,J) = A(I-1,J) + A(I,J)
    ENDDO
    IF (I.LT.N) BARRIER
ENDDO
DO I = 1,N
    DOALL J = 1,N
        B(I,J) = B(I-1,J) + B(I,J)
    ENDDO
    IF (I.LT.N) BARRIER
ENDDO

can be merged together into

DO I = 1,N
    DOALL J = 1,N
        A(I,J) = A(I-1,J) + A(I,J)
        B(I,J) = B(I-1,J) + B(I,J)
    ENDDO
    IF (I.LT.N) BARRIER
ENDDO

The algorithm to perform this merge is straightforward since the two regions are completely independent.

From now on, the greedy partition in the nested loop case will refer to the partition built by the greedy algorithm with the modifications described in this section. To discuss the nested loop case formally, the notion of a partition changes somewhat: a partition consists of a list of sets, each set consists of connected components of the original graph and partitions of subgraphs of the original graph. The subpartitions correspond to the partitions of regions with deeper parallelism.

The goal of the greedy code generator is to minimize the number of barrier synchronization points. Since subpartitions inside elements of a partition correspond to statements surrounded by a loop, and between each element in a subpartition there will be a barrier synchronization point, if we assume loop bounds are greater than one, then each barrier synchronization point in a subpartition corresponds to more than one barrier synchronization points to be executed.

**Definition 2.3** The number of barrier synchronization points of a partition is equal to the number of elements in the partition minus 1 plus twice the number of barrier synchronization points in each sub-partition.

Note that this definition is intended only to avoid the need to refer to loop bounds in the following theorem. It is based on the assumption that all loop bounds are at least two.
Theorem 2.2 The greedy partition of $G$ has a minimal number of barrier synchronization points.

Proof: If there are no nested loops, then the number of barrier synchronization points is equal to the number of elements in the outermost partition, and the theorem follows from Theorem 2.1. Assume that the theorem is true for all graphs $G$ corresponding to loops with maximum nesting level less than $k$. Let $G$ be a graph associated with a loop with maximum nesting level equal to $k$. Each serial region recursed on is a graph associated with a program with a maximum nesting level less than the current graph. If that region has deeper parallelism, then by the induction hypothesis, we have a partition for the statements inside that region that minimizes the number of barrier synchronization points.

Let $GP = G_1, \ldots, G_n$ be the greedy partition of $G = (V,E)$ and let $P = P_1, \ldots, P_m$ by a consistent partition of $G$ with a minimal number of barrier synchronization points. As in Theorem 2.1, it can be shown that either the number of barrier synchronization points in $P_1$ is more than in $G_1$, and so the theorem follows by induction on $n$, or we can show that $P$ is not minimal, in contradiction to its definition.

The proof of Theorem 2.1 holds for this generalized problem except that for $w \in P_1 - G_1$ such that $\text{preds}(w) \subset G_1$ may not be inconsistent with one of its predecessors. Instead it, or one of its predecessors, may have deeper parallelism, in which case we can show that $P$ is not minimal. Let $v \in \text{preds}(w)$. Since $w \in P_1$ we know that $w$ is consistent with $v$ and so $v \notin G_1$ implies both $w$ and $v$ serial regions, neither is scalar and at least one has deeper parallelism. Split $P_1$ into $P'_1$ and $P''_1$ where $P''_1$ consists of $w$ and all of its successors and $P'_1$ is the rest of $P_1$. The partition $P' = (P'_1, P''_1, P_2, \ldots, P_n)$ is consistent. Breaking $P_1$ reduces the number of barrier synchronization points by removing the barrier synchronization inside the subpartition containing $v$ and $w$, and replacing it with a barrier synchronization point between $P'_1$ and $P''_1$. Since a subpartition now has one fewer element and the main partition has one more, by definition of the number of barrier synchronization points for a partition, there is a net reduction of 1 barrier synchronization point. Hence $P'$ has fewer barrier synchronization points than $P$ which contradicts the selection of $P$.

We have shown that the nodes of $G$ in $G_1$ are a subset of the nodes in $P_1$. To show that $G_1$ has no more barrier synchronization points than $P_1$, let $r$ be the node in $G_1$ with deeper parallelism, such that the number of barrier synchronization points in the subpartition associated with $r$ is greater than the number of barrier synchronization points associated with any other deeper region in $G_1$. Let $k$ be the number of elements in the subpartition associated with $r$. The number of barrier synchronization points in $P_1$ is at least $2(k-1)$. Since all deeper regions in $G_1$ are merged together to share barrier synchronization points, the number of barrier synchronization points in $G_1$ is exactly $2(k-1)$, and hence $G_1$ has fewer barrier synchronization points than $P$.

The theorem follows by induction on the length of $GP$ as in Theorem 2.1.

End of Proof.
2.5 Loop Interchange

Loop interchange can be used to improve program performance by creating larger parallel regions. For example, in the following loop nest, the $I$ loop carries a recurrence and must be executed serially, but the $J$ loop can be parallelized. However, if we interchange the levels of the loops:

\[
\begin{align*}
\text{DO } I &= 1, M \\
\text{DO } J &= 1, N \\
A(I,J) &= A(I,J) + A(I-1,J) \\
\text{ENDDO} \\
\text{ENDDO}
\end{align*}
\]

\[
\begin{align*}
\text{DO } J &= 1, N \\
\text{DO } I &= 1, M \\
A(I,J) &= A(I,J) + A(I-1,J) \\
\text{ENDDO} \\
\text{ENDDO}
\end{align*}
\]

the outer loop can be parallelized, and the expected speedup will be greater because the amount of synchronization is reduced and the amount of useful work done between synchronization points is increased.

There may be more than one loop in a nest which could be parallelized when shifted to the outermost level. In the following loop nest, both the $I$ loop and the $J$ loop could be shifted to the outermost level and parallelized:

\[
\begin{align*}
\text{DO } I &= 1, N \\
\text{DO } J &= 1, N \\
X(I,J) &= X(I,J) + T*Y(I,J) \\
\text{ENDDO} \\
\text{ENDDO}
\end{align*}
\]

The choice of which loop to shift to the outermost position and parallelize is affected by the context of the loop nest. Consider the case where only the $J$ loop can be parallelized in the next loop nest (strongly connected region) of the source program, as in:

\[
\begin{align*}
\text{DO } I &= 1, N \\
\text{DO } J &= 1, N \\
X(I,J) &= X(I,J) + T*Y(I,J) \\
U(I,J) &= U(I-1,J) + X(I,J) \\
\text{ENDDO} \\
\text{ENDDO}
\end{align*}
\]

Note that if we distribute both of loops and then interchange loops only around the first strongly connected region, a fusion-preventing dependence (see section 2.7) is introduced. To avoid a barrier synchronization point between these loop nests, it is necessary (by rule (P4)) that the outer loops be fused together. To fuse these loops together, the $J$ loop must be in the outermost position for both strongly connected regions.
Adding loop interchange to the parallel code generator makes the problem of minimizing barrier synchronization points NP-complete. The difference which loop interchange adds to the consistent partition problem is that consistency of a member of a partition (which implies no barrier synchronization points are needed) is no longer implied by the pairwise consistency of the nodes in that partition. For example, consider the following loop nest:

```plaintext
DO I = 1, M
    DO J = 1, N
        A(I,J) = A(I,J) + A(I-1,J)
        B(I,J) = B(I,J) + B(I,J-1)
        C(I,J) = A(I,J) + B(I,J)
    ENDDO
ENDDO
```

This nest has three regions, each consisting of a single statement. The first region, consisting of the assignment to A, is serial at the outer level, but if loop interchange is applied, the J loop can be parallelized as the outermost loop. The second region consists of the assignment to B and can be parallelized as is; if the loops were interchanged, the J loop cannot be parallelized in the outermost position since it carries a recurrence. The final region consists of the assignment to C and can be parallelized regardless of which loop is in the outermost position.

Information about which loops can be shifted to the outermost position and parallelized can be added to the dependence graph as shown in Figure 7. The letters correspond to the variables that are modified by the statement in each region. The sets indicate which loop levels can be interchanged to the outermost position and parallelized, and the arrows represent the inter-region dependences. Observe that along each edge, the endpoints are consistent, in the sense that there is a loop which can be put in the outermost position and parallelized. The resulting two loops can be fused so that rule (P4) applies. In this example, if different loops are selected as outermost, then fusion-preventing dependences will exist between first and third statements, and likewise for
the second and third statements. Hence, to fuse these statements back together, they must have the same outermost loop. It is not possible, however, to select outermost loops for all three regions such that they can all be fused into a single parallel region. This leads to a generalized notion of consistency and to a generalized consistent partition problem. In the following definition, the function $\sigma$ corresponds to the mapping of strongly connected region into the set of levels which can be parallelized in the outermost position.

**Definition 2.4** Let $G = (V, E)$ be a graph and $\sigma$ a function mapping $V$ into the $2^{\{1, \ldots, k\}}$ for some integer $k$. A consistent partition of $(G, \sigma)$ is a partition $P = P_1, \ldots, P_n$ of $V$ such that the following two conditions hold:

a. If $(v, w) \in E$ then $i \leq j$

b. For each connected component $R$ of each member $P_i$, either $\sigma(v) = \emptyset$ for all $v \in R$ (every region is serial) or there exists an integer $j$ such that $j \in \sigma(v)$ for all $v \in R$ and that if the loop at level $j$ is shifted to the outermost level, then all of the regions in $R$ can be fused into a single parallel region.

Under this new definition, the problem of minimizing the number of barrier synchronization points will be shown to be NP-complete. This is not surprising since the question of whether a barrier synchronization point is needed between two particular regions can no longer be answered only with information about the two regions involved. The problem will be shown to be NP-hard by showing a related decision problem, based on a restriction of the general problem of code generation with loop interchange, is NP-hard. The restrictions assumed are that all regions were originally part of the same loop nest and that all inter-region edges were loop independent in the original loop nest. The significance of these restrictions is that two parallel regions can be fused if and only if they have the same loop at the outermost level. The above example satisfies these restrictions.

**Definition 2.5** An instance of the Restricted Consistent Partition Decision Problem (RCPDP) consists of a graph $G = (V, E)$, a function $\sigma$ mapping $V$ into $2^{\{1, \ldots, k\}}$ for some $k$ and a positive integer $N$. The question is: does there exist a partition, $P = \{P_1, \ldots, P_N\}$, of $V$ such that

a. If $(v, w) \in E$, $v \in P_i$, and $w \in P_j$, then $i \leq j$

b. For each connected component $R$ of each member $P_i$, either $\sigma(v) = \emptyset$ for all $v \in R$ or there exists an integer $j$ such that $j \in \sigma(v)$ for all $v \in R$.

Note that this definition differs from the last definition of consistent partition only in that the second condition is weaker; the additional constraints are imposed on the input programs.
Theorem 2.3 The Restricted Consistent Partition decision Problem is NP-Hard.

Proof: This theorem will be proved by demonstrating a polynomial reduction from the 3 Conjunctive Normal Form Satisfiability Problem (3SAT). An instance of the 3SAT consists of a boolean

---

3 Result proved independently by Joe Warren
expression in conjunctive normal form:

\[ H = \Pi_{i=1}^{m} F_i \]

where

\[ F_i = l_i^1 + l_i^2 + l_i^3 \]

and each literal \( l_i^j \) is a variable in the set \( \{v_1, \ldots, v_n\} \) or the negation of a variable in that set. The answer to a 3SAT problem is YES if there exists an assignment of logical values true and false to the variables that satisfies \( B \). A graph, \( G_B \), is constructed from an instance of 3SAT as follows: for each logical variable \( v_i \), there is a node labeled \( v_i \); none of these nodes have any incoming edges; for each clause of three literals, there is a sub-graph isomorphic to the one shown in Figure 8; and each node is annotated with a subset of \( \{1, 2\} \). The nodes labeled \( v_i \) are each labeled with \( \{1, 2\} \), and every node below the second row is labeled as in Figure 8. The labels of the nodes in the second row depend on the literals in the clause: if \( l_i^j \) is an unnegated instance of variable \( v_j \), then the node corresponding to that literal is annotated with the set \( \{2\} \). Otherwise, when the literal is a negated variable, the node is labeled with the set \( \{1\} \). Figure 8. illustrates for the example clause \( F_i = v_j + \bar{v}_k + v_l \).

The strategy behind this construction is that selecting an outermost loop for the "variable" nodes effectively selects a truth value for that node: \( 1 = \text{false} \) and \( 2 = \text{true} \). A node on the second row can be fused with its parent only if the associated variable is assigned a value which makes the literal associated with the child node true. The remaining nodes of the subgraph for each clause test that at least one literal is satisfied.

The sets annotating the nodes define a function \( \sigma_B \) from the nodes of \( G_B \) into \( \{1, 2\} \) and let \( N = 7 \), and so an instance of RCPDP has been constructed from an instance of 3SAT in polynomial time. The next step is to establish that the 3SAT instance is satisfiable if and only if \( \langle G_B, \sigma_B \rangle \) has a consistent partition with seven members. The only real choices in selecting a consistent partition for the graph \( G_B \) and function \( \sigma_B \) are which 'loops' to select as the outermost loops for the nodes labeled \( v_i \). These choices determine if any of the nodes in the second row of the subgraphs corresponding to the clauses (see Figure 8) can be part of the first member of the partition

If none of the nodes on the second row can be put in the first partition, the minimal consistent partition for the subgraph containing that node will have eight members. This is illustrated in Figure 9(b). However, if any of the nodes on the second row can be put in the first partition, the subgraph containing that node will have a consistent seven partition, as illustrated in Figure 9(a) (the other two cases are similar).
A node $l$ on the second row can be put in the first partition if and only if the loop chosen for the variable node that is its immediate ancestor is the same as the loop in the singleton set labeled $l$. If different loops are selected for a literal node and its ancestor, the two parallel regions they represent cannot be fused and so rule (P4) will prevent the nodes from being in the same partition.

Figure 9. Example seven partition and eight partitions
For example, in the case of regions $A$ and $C$ in Figure 7, where region $A$ must have the $J$ loop run in parallel and can only be fused with region $C$ if $C$ also has the $J$ loop in the outermost position. The restrictions on the problem ensure that if the outermost loops are the same, then the regions can be fused, otherwise they cannot be fused.

Let $T: \{v_1, \ldots, v_n\}$ be a truth assignment that satisfies $B$. This truth assignment is a guide to selecting outermost loops for the nodes in $G_B$ labeled with variables: if $T$ assigns true to $v_i$, then select loop 2 for the node $v_i$, otherwise, select loop 1. Let $F_i$ be a clause of $B$ and $G_{F_i}$ be the subgraph corresponding to $F_i$. $F_i$ is satisfied by $T$ and so some literal $l_i^j$ is satisfied. If $l_i^j$ is the unnegated variable $v_k$, then $T$ assigns true to $v_k$ and so loop 2 is selected for node $v_k$. Loop 2 is the only choice for the node corresponding to $l_i^j$, hence the node corresponding to $l_i^j$ can be consistently put in the first partition and the subgraph $G_{F_i}$ has a consistent seven partition. On the other hand, if $l_i^j$ is the negation of variable $v_k$, then $T$ assigns false to $v_k$ and so loop 1 is selected for node $v_k$. Loop 1 is the only choice for the node corresponding to $l_i^j$, therefore the node corresponding to $l_i^j$ can be consistently put in the first partition and again the subgraph $G_{F_i}$ has a consistent seven partition. Since each clause is satisfied, the entire graph as a consistent seven partition.

Assume that $G_B$ has a consistent seven partition: then in each clause subgraph, $G_{F_i}$, at least one node, $l_i^j$, on the second row is contained in the first partition. Define a truth assignment as follows: if the node $l_i^j$ corresponds to a literal consisting of the unnegated variable $v_k$, then assign true to $v_k$, otherwise, $l_i^j$ corresponds to a literal consisting of the negated variable $v_k$ and assign the value false to $v_k$. If any variable is not assigned a value by this rule, assign that variable true. This assignment is well defined, otherwise, for some variable $v_k$, nodes corresponding to literals consisting of both negated and unnegated instance of $v_k$ occur in the first partition and the first partition would not be consistent. To show that this truth assignment satisfies $B$, note that for each clause $F_i$, there is a literal $l_i^j$ that is true under the above truth assignment, therefore $F_i$ is true. Since each clause is satisfied, $B$ is satisfied.

The last two paragraphs have established that $B$ is satisfiable if and only if $G_B$ with $\sigma_B$ has a consistent seven partition.

End of Proof.

**Theorem 2.4** The Code Generation Problem with Loop Interchange is NP-Complete.

*Proof:* The previous theorem shows that the general problem has a subproblem that is NP-hard, this theorem follows if we can show that the general problem is in NP. This is straightforward: for each region that has more than one choice for outermost parallel loop, select one non-deterministically. Apply the algorithm of the Section 4 to obtain a minimal consistent partition in linear time based on this choice of outermost loops.

End of Proof.
2.6 Using Loop Interchange

Theorem 2.4 implies that minimizing the amount of barrier synchronization with loop interchange is too expensive to be practical for a compiler. Thus, we need a heuristic which will do a good job selecting outermost loops in a reasonable amount of time.

There are two aspects of loop interchange which must be included in the greedy algorithm. One is that parallel loops are consistent only if they have some parallel loop level in common. The other is that each parallel connected component of the current partition has an associated set of parallel loop levels. Thus, even if a node is consistent with each successor, it may not be consistent with the connected component of the partition it is added to. When an element is selected for processing from the set nopreds, it must be verified that is in fact "ok", meaning that if that node is added to the current partition, the resulting connected component which contains it still has at least one parallel level. A fast method for performing this check will be discussed in Section 3.5.

Of the nodes which are "ok" to add to the current partition, we want to give a preference to the ones which cannot be fused with some immediate successor. The rational behind this preference is that if we do not pick such a node, there is a greater chance that it will not be put in the current partition and one of its successors will not be put in the next partition. Therefore at least two barrier synchronization points will be needed, whereas if we select the node, then its successor is more likely to be put in the next partition and only one barrier synchronization point will be needed.

Thus, to handle the nested loop case: a set of parallel levels is determined for each region; the notion of not ok is modified to include the case where addition of a node to the current region results in no common parallel loop for the connected component containing it; and the selection of elements from the set nopreds is modified to give a preference to nodes that can not be fused with an immediate successor.

2.7 Loop Fusion with Loop Interchange

In procedure check.sons, it must be determined whether two strongly connected regions can be fused. As discussed in the restricted loop interchange problem, one sufficient condition is that the two strongly connected regions originally came from the same loop nest and currently have the same outermost loops. A more general problem involves the presences of fusion-preventing dependences. A loop independent dependence from one loop to another is fusion-preventing, if in the post fusion loop, the dependence becomes loop carried and reverses direction, indicating that the order of accesses to some memory location may have been reversed.

It is straightforward to test for fusion-preventing dependences using any dependence test by simply ignoring the fact that references occur in separate loops, and using the maximum of the two loop bounds where loop bound information is needed in the test. If any dependence is carried at a level deeper than the common nesting level of the references, and that dependence is backward (textually), then it prevents fusion of loops corresponding to the level it is carried at. Example:
DO I = 1,N  
DO J = 1,M  
   A(I,J) =  
ENDDO  
ENDDO  
DO I = 1,N  
DO J = 1,M  
   ... = A(I-1,J) + A(I,J+1)  
ENDDO  
ENDDO

Dependence analysis will indicate two dependences between these loops. Both are carried at levels deeper than the common nesting level (which is 0). One is carried by the outer loop, but is forward and so is not fusion-preventing. The other is carried by the inner loop and is backward, hence prevents fusion of the inner loops. As a result the dependence graph contains two loop independent dependences from the first statement to the second, one of which becomes loop carried at level 1 if the outer loops are fused; the other is fusion-preventing at level 2.

Note that if loops are interchanged in both nests, the post-fusion levels of the two edges change, but the directions do not. In this case, after loop interchange, the outer loops could not be fused since the fusion-preventing dependence between the loops is now associated with the outer loops. Standard tests do not provide sufficient information for determining which loops can be fused if loop interchange is applied to one loop nest but not the other. The tests could be modified to provide information for every possible ordering of loops around the individual references, but this is probably prohibitively expensive. If each reference is contained in $k$ loops, at least $k^2$ tests must be performed and perhaps as many as $k!k!$. If this information were available, the restriction that loops have the same outermost loop could be dropped.

2.8 Control Dependence

So far I have omitted control flow information from the discussions in this chapter. When control dependences are converted to data dependences via If-conversion[AKPW 83], programs do not have explicit control flow except for loops; cleanup passes attempt to restore explicit control flow after parallelism has been made explicit. When control dependences appear explicitly in the dependence graph, as in [FeOW 84], each transformation must handle control flow program constructs.

An important example of this is loop distribution. If the parallel code generator decides to separate two statements into separate loops, it is necessary to duplicate the control flow inside the original loop for each distributed part. This implies that the direction taken at each branch during
execution of the first of the resulting loops must be remembered so that it can be duplicated in the remaining loops. With If-Conversion, this is automatic since every statement is individually guarded. Without If-conversion, a certain amount of work must be performed after loop distribution choices have been made. This work is straightforward if backward goto's have been removed: at every point where a branched is taken, the direction is stored in a variable\(^4\). This variable must be scalar expanded: in subsequent loops, these stored values can be used to repeat the same path through the control flow graph. Exit branches can also be handled: the exit of the first loop jumps to the next loop, which exits to the next. The last loop exits to the original target of the exit branch. Backward branching control flow is more problematic but can be converted to iterative constructs in a straightforward manner, and subsequently ignored.

While control flow has been omitted from the discussion so far, it can be handled in one of several straightforward ways depending on how it is handled in general.

2.9 Summary

This chapter presented an algorithm for translating sequential Fortran programs into programs with explicit parallelism expressed with a simple fork-join construct. The cost of synchronization is assumed to be encapsulated in the form of barrier synchronization points and the greedy algorithm is shown to be optimal in minimizing the number of barrier synchronization points.

The important transformations of loop interchange and loop fusion are discussed in the context of parallel code generation. It is shown that minimizing the number of barrier synchronization is NP-complete in the presence of loop nests with multiple loops that can be executed in parallel as the outermost loop. A heuristic is proposed and modifications to the parallel code generator are given to use loop interchange. Loop fusion fits naturally into the parallel code generator and depends only on information that is easily derived from standard dependence tests.

Finally, the presence of forward branching control flow is discussed and seen not to affect the algorithm but may require extra work when loop distribution is done.

\(^4\) Logical for block if's, numeric for arithmetic if's and computed goto's
Chapter 3:
Loop Alignment and Code Replication

3.1 Introduction

Loop carried dependences are a major source of synchronization generated by the algorithm presented in the previous chapter. This chapter develops a method to convert loop carried dependences into dependences that are either carried at a deeper level in the loop nest or are loop independent. This method is called loop alignment; it can be viewed as moving the memory references of a statement from one iteration of the loop to another, and hence can be used to “align” all references to a memory location $M$ so that they occur in the same iteration of the loop. For example, in the loop:

$$
\begin{align*}
\text{DO } & I = 1,N \\
S_1 & : A(I) = C(I) + T \\
S_2 & : B(I) = A(I-1)
\end{align*}
$$

ENDDO

the value computed by statement $S_1$ is used by statement $S_2$ on the next iteration. As a result, this loop cannot be run in parallel since the value computed by $S_1$ may not be stored in $A$ by the time the memory fetch occurs for its use in $S_2$. If the second statement is “shifted” backward, the two references become aligned on the same iteration. A possible result is:

$$
\begin{align*}
\text{DO } & I = 0,N \\
S_1 & : \text{IF}(I \text{.GT.} 0) A(I) = C(I) + T \\
S_2 & : \text{IF}(I \text{.LT.} N) B(I+1) = A(I)
\end{align*}
$$

ENDDO

where the statement guards preserve semantics on the boundary iterations. This resulting loop can be executed as a DOALL. Note that each statement is executed in exactly one of the boundary iterations. An alternative to the explicit guards is to “fold” the first iteration onto the last as in:
DOALL I = 1,N
  IF (I.GE.N) THEN
    II = 1
  ELSE
    II = I+1
  ENDDIF
  A(I) = C(I) + T
  B(II) = A(II-1)
ENDDO

This folding also helps balance the work over all iterations. Examples in this paper will use guards or unrolling to make the results of the transformations clearer. Loop alignment was used by Padua [Padu 79] when partitioning loops.

Since any loop carried dependence between a pair of statements will force a barrier synchronization point to be generated by the parallel code generator, it is necessary to find an alignment of the statements that simultaneously converts all loop carried dependences into dependences that are carried by deeper loops or are loop independent. It is possible for a conflict to arise in the alignment requirements of a set of statements. Such an alignment conflict can occur when the output of statement $S_i$ is used by statement $S_j$ on two different iterations of the loop. In such a case, $S_i$ could be replicated and unique names could be generated for the outputs of each of the resulting statements. The uses would then be renamed using both the original and new names to associate uses with different alignment requirements with different copies of the replicated statement. This demonstrates the use of code replication to eliminate alignment conflicts. An example is:

\[
\begin{align*}
\text{DO I = 1,N} \\
S_1 & \quad A(I) = C(I) + T \\
S_2 & \quad B(I) = A(I-1) + A(I)
\end{align*}
\]

Statement $S_1$ computes a value which is used by $S_2$ both in the current and the next iterations, thus $S_1$ cannot be aligned to satisfy both of these requirements. If $S_1$ is replicated, and the uses partitioned, as in:

\[
\begin{align*}
\text{DO I = 1,N} \\
S_1 & \quad A1(I) = C(I) + T \\
S_2 & \quad A(I) = C(I) + T \\
S_3 & \quad B(I) = A(I-1) + A1(I)
\end{align*}
\]

then alignment becomes possible:
DOALL I = 1,N+1
   IF(I.GE.2) A(I) = C(I) + T
   IF(I.LE.N) A(I-1) = C(I-1) + T
   IF(I.LE.N) B(I) = A(I-1) + A(I)
ENDDO

Observe that A1 has no use previous or subsequent to this loop (except for the initialization) and therefore can be kept as a single scalar in the local memory associated with each processor.

The first part of this chapter (section 3.2) makes the notion of alignment conflict rigorous and develops a modified version of the parallel code generator that applies loop alignment when it will remove all loop carried dependences between statements in a set of statements. The second part of this chapter (section 3.3) develops code replication as a transformation and establishes that code replication can resolve any alignment conflict that is not also a recurrence.\(^1\) The third part of this chapter (section 3.4) examines the problem of minimizing the amount of code replication that is necessary to resolve all alignment conflicts. It is shown to be NP-hard. The remaining sections of this chapter discuss heuristics for using code replication in the parallel code generator of Chapter 2 and the impact of loop alignment on other transformation such as loop interchange and loop fusion.

### 3.2 Loop Alignment

#### 3.2.1 Textual Loop Alignment

Loop alignment is a reordering transformation. It can be viewed as moving instances of statements from one iteration to another. The following is a working definition of alignment and will be revised later.

**Definition 3.1** Let \( S \) be a statement inclosed by at least \( k \) loops. Assume that the loop at level \( k \) is has control variable \( I_k \), then an alignment of \( S \) by \( j \) with respect to level \( k \) is performed by replacing each occurrence of \( I_k \) in \( S \) by \( I_k - j \).

This definition is not complete because the iteration space must be modified as well as the guards on certain statements. I adopt the following notation to indicate the above

\(^1\) Definition 1.2, pg 9
replacement: if $S$ is a statement, then $S[I_k]$ represents the statement $S$ where all occurrences of the variable $I_K$ are considered as a parameter. Hence, $S[I_k - j]$ represents the statement $S$ after each occurrence if $I_k$ is replaced with the expression $I_k - j$. It should be clear from context, which variable is being replaced.

If statement $S_1$ has an array reference $A(b_0 + \vec{a} \cdot \vec{I})$ and $S_1$ is aligned by $j$ with respect to the loop at level $k$, then the new array reference is given by $A(b_0 - jb_k + \vec{b} \cdot \vec{I})$. Note that this is only different from the original array reference if $b_k \neq 0$, thus:

**Definition 3.2** A memory reference $A(a_0 + \vec{a} \cdot \vec{I})$ is alignment sensitive at level $k$ if $a_k \neq 0$. A data dependence is alignment sensitive at level $k$ if at least one of the references which generate the dependence is alignment sensitive at level $k$. If both of the generating references are alignment sensitive, then the data dependence is fully alignment sensitive.

From now on, we will assume that all data dependences are fully alignment sensitive. If neither endpoint varies in the loop, then array expansion [Wolf 78] can be applied and the resulting array references will be fully alignment sensitive.

Assume that $S_2$ contains the alignment sensitive array reference $A(a_0 + \vec{a} \cdot \vec{I})$ and $S_1$ contains the array reference $A(b_0 + \vec{b} \cdot \vec{I})$ and that $\vec{i}$ and $\vec{i} + \vec{d}$ satisfy equation (2) of Chapter 1:

$$(a_0 - b_0) + \vec{a} \cdot \vec{i} - \vec{b} \cdot (\vec{i} + \vec{d}) = 0$$

Now perform loop alignment by $j$ with respect to loop $k$ on statement $S_2$ and define a distance vector $\vec{d}'$ by:

$$d'_l = \begin{cases} 
  d_l, & \text{if } l \neq k; \\
  d_l + j, & \text{if } l = k.
\end{cases}$$

Substituting into equation (2) we see that after the alignment:

$$\begin{align*}
(a_0 - b_0 + jb_k) + \vec{a} \cdot \vec{i} - \vec{b} \cdot (\vec{i} + \vec{d}')
&= (a_0 - b_0 + \vec{a} \cdot \vec{i} - \vec{b} \cdot (\vec{i} + \vec{d})) + jb_k - \vec{b} \cdot (\vec{d}' - \vec{d}) \\
&= 0 + b_k j - b_k j \\
&= 0
\end{align*}$$

and hence $S_2$ has a data dependence on $S_1$ with distance vector $\vec{d}'$ if $\vec{i} + \vec{d}'$ is in the iteration space.

The latter condition implies the second half of the alignment processes: we alter the iteration space to ensure that $\vec{i} + \vec{d}'$ is in the iteration space of the aligned loop exactly
when $\vec{t} + \vec{d}$ is in the original iteration space. Since the vector $\vec{d} - \vec{d}'$ is $j$ in the $k^{th}$ coordinate and 0 elsewhere, the change in the iteration space is restricted to the bounds of the $k^{th}$ loop. Here we assume that the original loop bounds have been normalized to run from $L_k$ to $U_k$ by an increment of 1, so the new loop bounds are given by:

lower bound = $\min(L_k, I_k + j)$

upper bound = $\max(U_k, U_k + j)$

and we guard each statement as follows: if $S_i$ is not aligned, then replace $S_i$ with:

IF ($L_k \leq I_k \land I_k \leq U_k$) $S_i$

and replace the aligned statement, $S_1[I_k]$, with:

IF ($L_k \leq I_k - j \land I_k - j \leq U_k$) $S_1[I_k - j]$

Note that with these guards, even though we have enlarged the iteration space, each statement is executed the same number of times with essentially the same iteration vectors. The only change is the relative order of statement execution. Thus we have argued:

**Proposition 3.1** Loop alignment as described above is a reordering transformation.

If we align the second statement, $S_1$ by $j$ w.r.t. level $k$, and define the distance vector $d''$ by:

$$d''_l = \begin{cases} d_l, & \text{if } l \neq k; \\ d_k - j, & \text{if } l = k. \end{cases}$$

and define a new iteration vector $\vec{i}'$ by:

$$i'_l = \begin{cases} i_l, & \text{if } l \neq k; \\ i_k + j, & \text{if } l = k. \end{cases}$$

Note that $\vec{i}' + d'' = \vec{i} + \vec{d}$. Now we will return to equation (2) of Chapter 1 and note that after the alignment we have:

$$a_0 - ja_k - b_0 + \vec{a} \cdot \vec{i}' - \vec{b} \cdot (\vec{i} + \vec{d}'') = \left(a_0 - b_0 + \vec{a} \cdot \vec{i} - \vec{b} \cdot (\vec{i} + \vec{d})\right) - a_k j + \vec{a} \cdot (\vec{d} - d'')$$

$$= 0 - a_k j + a_k j$$

$$= 0$$

and hence, if $\vec{i}'$ is in the iteration space, then $S_2$ has a loop carried dependence on $S_1$ with distance vector $d'$. In general, loop alignment selects an integer for each statement in the loop body, and adjusts guards and loop bounds so that aligning each statement by the selected amount is a reordering transformation. The complete definition of loop alignment is:
Definition 3.3 Let \( L \) be a loop consisting of the statements \( S_1, \ldots, S_n \), and \( \sigma \) a function which assigns to each statement in the loop an integer. Let \( m \) and \( M \) be the lower and upper bounds of \( L \) respectively. Define:

\[
m' = m + \min\{\sigma(S_1), \ldots, \sigma(S_n)\}
\]
\[
M' = M + \max\{\sigma(S_1), \ldots, \sigma(S_n)\}
\]

then we perform a \( \sigma \)-alignment at level \( k \) by replacing each statement \( S_i[I_k] \) by:

\[
\text{IF } (L_k \leq I_k - \sigma(S_i) \land I_k - \sigma(S_i) \leq U_k) \text{ then } S_i[I_k - \sigma(S_i)]
\]

The lower bound of the level \( k \) loop is replaced by \( m' \) and the upper bound is replaced by \( M' \). We say that \( \sigma \) describes the alignment.

Finally, we can characterize the effects of loop alignment on data dependences subject to the restrictions mentioned above.

Theorem 3.1 Let \( \sigma \) describe an alignment of \( L \) and assume that before the alignment, there is a valid consistent data dependence generated by the alignment sensitive memory reference \( A(a_0 + \bar{a} \cdot \bar{i}) \) in statement \( S_2 \) and the alignment sensitive memory reference \( A(b_0 + \bar{b} \cdot \bar{j}) \) in statement \( S_1 \). Let \( \bar{d} \) be the distance vector of that dependence. Define \( \bar{d}' \) by:

\[
d'_{l} = \begin{cases} 
    d_l, & \text{if } l \neq k; \\
    d_l - \sigma(S_1) + \sigma(S_2), & \text{if } l = k.
\end{cases}
\]

Then after a \( \sigma \)-alignment, there is still a data dependence of \( S_2 \) on \( S_1 \), but its distance vector is now \( \bar{d}' \).

Proof: Assume vectors \( \bar{i} \) and \( \bar{j} \) in the iteration space satisfy:

\[
a_0 - b_0 + \bar{a} \cdot \bar{i} - \bar{b} \cdot \bar{j} = 0
\]

now define \( \bar{i}' \) and \( \bar{j}' \) by:

\[
i'_l = \begin{cases} 
    i_l, & \text{if } l \neq k; \\
    i_l + \sigma(S_2), & \text{if } l = k.
\end{cases}
\]
\[
j'_l = \begin{cases} 
    j_l, & \text{if } l \neq k; \\
    j_l + \sigma(S_1), & \text{if } l = k.
\end{cases}
\]

and note that:

\[
i'_l - j'_l = \begin{cases} 
    i_l - j_l = d_l = d'_l, & \text{if } l \neq k; \\
    i_l - j_l - \sigma(S_1) + \sigma(S_2) = d_l - \sigma(S_1) + \sigma(S_2) = d'_l, & \text{if } l = k.
\end{cases}
\]
and so \( \vec{d} = \vec{p} - \vec{j} \). By construction, observe that both \( \vec{p} \) and \( \vec{j} \) are in the post alignment iteration space and that the guard of statement \( S_1 \) is true on iteration \( \vec{p} \) and the guard of statement \( S_2 \) is true on iteration \( \vec{j} \).

Substituting into (2) of Chapter 1 after the \( \sigma \)-alignment, we get:

\[
\begin{align*}
a_0 - a_k \sigma(S_1) - b_0 + b_k \sigma(S_2) + \bar{a} \cdot \vec{p} - \bar{b} \cdot \vec{j} \\
= (a_0 - b_0 + \bar{a} \cdot \vec{t} - \bar{b} \cdot \vec{t}) + b_k \sigma(S_2) - a_k \sigma(S_1) + \bar{a} \cdot (\vec{p} - \vec{t}) - \bar{b} \cdot (\vec{j} - \vec{t}) \\
= 0 + b_k \sigma(S_2) - a_k \sigma(S_1) + a_k \sigma(S_1) - b_k \sigma(S_2) \\
= 0
\end{align*}
\]

and thus statement \( S_2 \) has a data dependence on \( S_1 \) with distance vector \( \vec{d} \) in the post alignment loop.

End of Proof.

**Corollary** \( S_1 \) and \( S_2 \) are aligned by the same constant with respect to the same loop, then any consistent dependences between them are preserved.

*Proof*: Immediate.

**Proposition 3.2** A level \( k \) loop alignment preserves all dependences at levels 1 through \( k - 1 \) inclusive.

*Proof*: Since loop alignment is a reordering transformation and a level \( k \) loop alignment affects only the \( k^{th} \) coordinate of the distance vector, the leftmost \( k - 1 \) components of the distance vector are 'preserved'.

End of Proof.

### 3.2.2 Alignment and The Dependence Graph.

Assume that the loop at level \( k \) for a program with dependence graph \( D = (S, E) \) carries no recurrences. In this case we want to align the statements so that it also has no level \( k \) dependences. Proposition 3.2 claims that loop alignment at level \( k \) preserves dependences carried by outer loops and Theorem 3.2 suggests that we need only look at edges with level \( k \) or greater to determine if the loop at level \( k \) can be run in parallel. Thus we need only look at a sub-graph of \( D \) which contains pertinent information.
**Definition 3.4** The level $k$ subgraph of the dependence graph $D = (S, E)$ is the graph $D_k = (S, E_k)$ where:

$$E_k = \{ e \in E \mid \text{level}(e) \geq k \}$$

At this point, our goal is to remove all level $k$ loop carried dependences. We can use Theorem 3.1 to compute the relative amount each statement should be aligned to accomplish this.

**Proposition 3.3** If $e \in E_k$ represents a level $k$ dependence and $\vec{d} = (d_1, \ldots, d_n)$ is the distance vector associated with $e$, $S_1$ is the source of $e$ and $S_2$ is the sink of $e$, then if $\sigma$ describes a loop alignment such that:

$$d_k = \sigma(S_1) - \sigma(S_2)$$

then after a $\sigma$-alignment of the $k^{th}$ loop, the $k^{th}$ component of the distance vector of $e$ will be zero and hence the level of the dependence will be greater than $k$.

**Proof:** Immediate from Theorem 3.1.

The $k^{th}$ component of the distance vector is important in determining an alignment for a loop at level $k$ so we will give it a name.

**Definition 3.5** The $k$-threshold of dependence edge $e \in E_k$ with distance vector $\vec{d}$, denoted $\tau_k(e)$, is defined by: $\tau_k(e) = d_k$ the $k$-th component of $\vec{d}$

When $k$ is clear from context, the $k$-threshold will also be called simply the threshold of the dependence. Based on Proposition 3.3, we define the type of alignment which we are after.

**Definition 3.6** An exact level $k$ alignment of $D$ is a function $\sigma$ mapping the statements of $D$ into the integers such that for all dependence edges, $e \in E_k$ we have:

$$\tau_k(e) = \sigma(\text{source}(e)) - \sigma(\text{sink}(e))$$

Assume that there exists an exact alignment, $\sigma$, for a loop $L$. Under what conditions is the corresponding $\sigma$-alignment of $L$ a valid\(^2\) transformation? There are only two cases in which alignment can generate invalid dependences. One case occurs when $S_2$ depends on $S_1$, the only non-zero component in the distance vector is the $k^{th}$ component, and $S_2$ proceeds $S_1$ in the program text. An example of this problem is:

---

\(^2\) In the sense of Section 1.2.6
DO I = 1, 100
    X(I) = A(I)
    A(I+1) = B(I)**2
ENDDO

The post alignment dependence would be loop independent and backward and therefore invalid. Reordering the statements in the post-alignment loop will make this transformation valid. If there are no cycles of dependences in $D_k$, after loop-alignment there were still be no cycles and so we can topologically sort the statements so that all dependences are textually forward. I will assume that this reordering is part of the alignment transformation from now on.

The other case occurs when the dependence is carried by the loop at level $k$ and the next non-zero coordinate (left to right) in the distance vector is negative. An example is given by:

DO I = 1, 100
    DO J = 1, 100
    S_1 A(I,J) = B(I,J) + 3
    S_2 C(I,J) = A(I-1,J+1) - 1
ENDDO ENDDO

The dependence of $S_2$ on $S_1$ has distance vector $\vec{d} = (1, -1)$ and so any exact level 1 alignment would transform that vector into $\vec{d}' = (0, -1)$ which is invalid. This problem is inherent in the dependence, not in the choice of alignment. Dependences with a distance vector in which the second non-zero component from the left is less than zero are called alignment preventing dependences, since an exact alignment will reverse these dependences and hence be invalid. We will return to this problem in a later section.

The following theorem establishes that exact alignments are the correct alignment to apply to obtain parallel loops.

**Theorem 3.2** If $L$ is a loop with dependence graph $D$, $\sigma$ an exact alignment for $D$ at level $k$, and $D$ contains no level $k$ alignment preventing dependences, and all dependences are consistent, then a $\sigma$-alignment is valid and the resulting dependence graph has no level $k$ carried dependences.

**Proof:** To show that the transformation is valid, let $D'$ be the post-alignment dependence graph and assume that there is an invalid edge $e'$ with distance vector $\vec{d}'$. Let $e$ be the
corresponding edge in $D$ and $\vec{d}$ be its distance vector. In both graphs, let $S_1$ be the source
of the dependences and $S_2$ the sink.

If $\vec{d} = \vec{0}$ then $S_2$ proceeds $S_1$, but then since $\vec{d}$ and $\vec{d}$ differ only in the $k^{th}$ component,
either $\tau_k(e) = 0$, in which case $e$ would be invalid in $D$ (which contradicts the definition
of the dependence graph), or $\tau_k(e) \neq 0$ in which case statements $S_2$ and $S_1$ would be
reordered by the alignment transformation so that $S_1$ proceeds $S_2$. If $\vec{d}^t < \vec{0}$ then either
$\tau_k(e) \leq 0$ and again $e$ is invalid or $\tau_k(e) > 0$ and $e$ is alignment preventing, either way we
have a contradiction. I conclude that $D'$ has no invalid edges and hence the $\sigma$-alignment
is valid.

If the level of $e \in D$ is less than $k$, then the level is still less than $k$ by Proposition
3.2. If the level of $e$ is greater than $k$, then $\tau_k(e) = 0 = \sigma(S_2) - \sigma(S_1)$, hence the $k^{th}$
component of $\vec{d}$ is zero by Theorem 3.1. If the level of $e$ is $k$, then:

$$
d'_k = d_k - \sigma(S_1) + \sigma(S_2)
$$

$$
= \tau_k(e) - \sigma(S_1) + \sigma(S_2)
$$

$$
= 0
$$

since $\sigma$ is an exact alignment. Thus the leftmost $k$ components of the $\vec{d}'$ are all 0 and so
the level of $e'$ is strictly greater than $k$. In each case, the level of $e'$ is not $k$, hence $D'$ has
no level $k$ dependences.

End of Proof.

3.2.3 Alignment Conflicts.

As was mentioned before, conflicts can arise in the alignment requirements of a pro-
gram. A canonical example is the following loop.

```
DO I = 2,100
   S_1 A(I) = C(I) + C(I+1)
   S_2 B(I) = A(I) + A(I-1)
ENDDO
```

The dependence graph has two true edges from $S_1$ to $S_2$. One of these edges has threshold
0 the other has threshold 1. Thus we have the conflicting requirements on any exact
alignment, $\sigma$, that:

$$
\sigma(S_2) - \sigma(S_1) = 0
$$

$$
\sigma(S_2) - \sigma(S_1) = 1
$$

This section will characterize alignment conflicts in terms of the dependence graph $D_k$.
The factors that enter into this characterization are the direction of edges and the $k^{th}$
component of the distance vector associated with each edge. For the rest of this section I
will talk only about weighted graphs.
Definition 3.7 A graph $G$ is weighted if every edge has an associated non-negative integer value.

The requirement that the weights be non-negative is satisfied for source dependence graphs since they consist only of valid dependences.

The notion of an exact alignment is the same with weight replacing the directed $k$-threshold, $\tau_k$. To talk about arbitrary paths in $G$, we make the following definition.

Definition 3.8 Let $G = (V, E)$ be a weighted, directed graph then define $U_G$, the undirected companion of $G$, by $U_G = (V, E \cup E^r)$ where:

$$E^r = \{e^r = (v, w) \mid e = (w, v) \in E\}$$

and the weights are given by $\text{weight}(e^r) = -\text{weight}(e)$.

First, observe that under this definition an exact alignment of $G$ is also an exact alignment of $U_G$. The weights on edges in $D$ give a rise to a requirement on an exact alignment that the endpoints differ by the weight of the edge, we can extend this to a similar notion for paths.

Definition 3.9 For a path $p = e_1 \ldots e_n$ in $U_G$, we define:

$$\text{weight}(p) = \sum_{i=1}^{n} \text{weight}(e_i).$$

Lemma 3.1 $\sigma$ is an exact alignment of $G$ if and only if for all paths $p = e_1 \ldots e_n$, in $U_G$ we have:

$$\text{weight}(p) = \sigma(\text{source}(p)) - \sigma(\text{sink}(p))$$

Proof: The 'if' part is immediate since every edge in $G$ corresponds to a path in $U_G$, satisfying the requirements of an exact alignment. If $\sigma$ is an exact alignment for $G$, then $\sigma$ is an exact alignment for $U_G$ and we have:

$$\text{weight}(p) = \sum_{i=1}^{n} \text{weight}(e_i)$$

$$= \sum_{i=1}^{n} \sigma(\text{source}(e_i)) - \sigma(\text{sink}(e_i))$$

$$= \sigma(\text{source}(e_1)) + \left(\sum_{i=1}^{n-1} \sigma(\text{source}(e_{i+1})) - \sigma(\text{sink}(e_i))\right) - \sigma(\text{sink}(e_n))$$

$$= \sigma(\text{source}(e_1)) + 0 - \sigma(\text{sink}(e_n))$$

$$= \sigma(\text{source}(p)) - \sigma(\text{sink}(p))$$

End of Proof.
We can define the values of an exact alignment on a connected component of $U_G$
by picking a value at any node and using the above lemma to determine the values of
the alignment at other nodes. An alignment conflict occurs when this method is not well
defined: there are two different values for a given node. This indicates two distinct paths
with different weights between the same endpoints. These paths define a cycle, hence:

**Definition 3.10** An alignment conflict in a weighted directed graph
$G$ is a cycle $p$ in the undirected companion of $G$ such that $\text{weight}(p) \neq 0$.

To show that this captures the notion, the following proposition makes the above
discussion rigorous.

**Proposition 3.4** A weighted directed graph has an exact alignment
if and only if it has no alignment conflicts.

**Proof:** If $G$ is a weighted directed graph and $G$ has an exact alignment $\sigma$, then for any
cycle $p$ in $U_G$, let $v$ be any node on $p$, then $p$ defines a path from $v$ to $v$ and hence:

$$\text{weight}(p) = \sigma(v) - \sigma(v) = 0.$$  

Now assume that $G$ has no alignment conflicts. Clearly we can align the connected
components of $G$ separately, so assume that $G$ is connected. Pick any vertex $v_0 \in V$ and
define $\sigma(v_0) = 0$. Since $G$ is connected, for each $v \in V - \{v_0\}$, there is a path from $v$ to
$v_0$ in $U_G$ so define $\sigma(v) = \text{weight}(p)$.

We can show that $\sigma$ is well defined as follows: let $p_1$ and $p_2 = e_1 \ldots e_n$ be two paths
from $v$ to $v_0$. If $p_2^r = e_n^r \ldots e_1^r$ is the path formed by reversing $p_2$, then $p_1p_2^r$
is a cycle in $U_G$. Since there are no alignment conflicts and $\text{weight}(p_2^r) = -\text{weight}(p_2)$:

$$0 = \text{weight}(p_1p_2^r)$$

$$= \text{weight}(p_1) + \text{weight}(p_2^r)$$

$$= \text{weight}(p_1) - \text{weight}(p_2)$$

hence $\sigma$ is well defined.

To show $\sigma$ is an exact alignment, let $e$ be an edge in $E$ and $p$ a path from $\text{sink}(e)$ to
$v_0$, then by construction:

$$\sigma(\text{source}(e)) = \text{weight}(ep)$$

$$= \text{weight}(e) + \text{weight}(p)$$

$$= \text{weight}(e) - \sigma(\text{sink}(e))$$

therefore $\text{weight}(e) = \sigma(\text{source}(e)) - \sigma(\text{sink}(e))$ and $\sigma$ is an exact alignment.

*End of Proof.*
A simple algorithm to determine an alignment for a loop would be based on depth first search. As the algorithm moves from node to node, it assigns an alignment value so that:

\[ \text{weight}(e) = \sigma(\text{source}(e)) - \sigma(\text{sink}(e)) \]

where \( e \) is the "tree" edge just traversed. When a "forward", "cross", or "back" edge is detected, a check can be made for alignment conflicts: the values of \( \sigma \) already assigned to the source and sink must satisfy the above equation where \( e \) is the edge being traversed.

### 3.2.4 Using Loop Alignment

This section shows how to add loop alignment to the algorithm in Chapter 2. The primary questions are how to efficiently detect alignment conflicts, and how to compute an exact alignment. As the parallel code generator visits strongly connected regions, let \( G \) be the subgraph of the dependence graph consisting of all nodes in the current partition and all of the edges between them carried at the current level or deeper. Recall from the proof of Proposition 3.4 that if a graph has no alignment conflicts then an exact alignment can be constructed by selecting a node from each connected component of the undirected companion, and using weights on paths from that node to each other node in the connected component to define the alignment values for the other nodes.

As the graph \( G \) grows, the connected components are sets of nodes, all of which have fixed relative alignments. The action of adding an edge to \( G \) either combines two previously unconnected nodes, or it specifies an alignment constraint between nodes whose relative alignment has already been fixed by earlier edges. To check whether adding an edge creates an alignment conflict, we maintain the connected components of \( G \) as disjoint sets of nodes. For each connected component, one particular node, the root, is assigned an alignment value of 0, and all other nodes in the component are assigned alignment values relative to this node.

Two primitives are needed: one to find the component containing a node and the nodes relative alignment value, and the other to combine two components into one by specifying an alignment value of one root relative to the other. These operations can be done quickly using a modified version of the fast disjoint set union-find algorithm [AhHU 74, sections 4.7-8]. So assume that a call to the function \textit{find}:
\((A, k) \leftarrow \text{find}(v)\)

returns both the connected component that contains node \(v\) and the relative alignment value of \(v\) with respect to the root of that component. Assume that a call to \text{union}:

\[
\text{union}(A, B, k)
\]

combines components \(A\) and \(B\) with the relative alignment of the root of \(A\) from the root of \(B\) equal to \(k\). The algorithms for these primitives given in [AhHU 74] are almost linear in worst case time complexity.

With these primitives, it is simple to test whether adding node \(v\) to the current partition will create an alignment conflict and add \(v\) and its edges to \(G\) if no alignment conflict is created. The code to do this is shown in Figure 10. The main inner loop in algorithm of Chapter 2 becomes:

\[
\text{while } \text{nopreds} - \text{NotOk} \neq \emptyset \text{ do }
\]

\[
\begin{align*}
\text{remove any node } p \text{ from } \text{nopreds} - \text{NotOk} \\
\text{if } \text{can_align}(p) \text{ then } \text{do} \\
\quad \text{add } p \text{ to } \text{visited} \text{ and } R \\
\quad \text{call } \text{check_sons}(p) \\
\text{else } \text{add } p \text{ to } \text{NotOk}
\end{align*}
\]

Finally the routine \text{generate_code} performs a final \text{find} on each node to get the final alignment values.

### 3.2.5 Loop Interchange

Recall from Section 2.6 that a set of parallel loop levels is maintained for each connected component of the current partition. We see that this information can be maintained using the same disjoint set abstraction as is used for detecting alignment conflicts. We associated with each disjoint set a bit vector indicating which loop levels are parallel for every strongly connected region that is part of that set. The \text{find} primitive returns an ordered triple consisting of the set the node is in, the relative offset of the node, and the parallel loop levels of the set. The \text{union} primitive updates this the set of parallel loop levels by \text{AND}ing together the bit-vectors associated with the sets to be unioned. The test to determine if adding a particular node to the current partition without losing parallelism is shown in Figure 11. The modified inner loop of the greedy algorithm is now:
function can_align(v : boolean
    /* Check for alignment conflict */
    align(*) = ⊥
    foreach edge e = (v, w) for w in current partition do
        (A, k) ← find(w)
        if align(A) = ⊥
            then align(A) = k
        else if align(A) ≠ k
            then { can_align ← False ; return }
    /* Now combine components */
    (A, k) ← find(v)
    foreach edge e = (v, w) for w in current partition do
        (B, l) ← find(w)
        if A ≠ B then do
            union(A, B, τ(e) + k - l)
        (A, k) ← find(v)
    can_align ← True
    return

Figure 10. Test for alignment conflicts

while nopreds − NotOk ≠ ∅ do
    remove any node p from nopreds − NotOk
    if can_fuse(p) then do
        if can_align(p) then do
            add p to visited and R
            call check_sons(p)
        else add p to NotOk
    else add p to NotOk

3.3 Code Replication

In the introduction to this chapter, the use of code replication was illustrated as a way of resolving alignment conflicts and allowing a loop to run in parallel. In this section we examine the virtues and limitations of this approach. Section 3.3.1 discusses the effects of code replication on the dependence graph, section 3.3.2 illustrates code replication on program text, section 3.3.3 establishes that code replication can parallelize any loop without cycles. Sections 3.3.4 and 3.3.5 extend section 3.3.3 to cyclic dependence graphs and show that no replication technique can parallelize a loop which carries a recurrence.
function can_fuse(v) : boolean

/* new is the set of loop levels that will be asso- */
/* ciated with the connected component that will */
/* contain v if v is added to the current partition */

let new be the set of loop levels of v that do not carry any
dependences.

/* If v is serial, then no check is needed. */

if new = ∅
    then { can_fuse ← True ; return }

foreach edge e = (v, w) for w in current partition do

/* old is the set of loop levels that are */
/* parallel for all nodes in set A */

(A,k,old) ← find(w)
new ← new ∩ old
if new = ∅
    then { can_fuse ← False ; return }

can_fuse ← True
return

Figure 11. Test for valid fusion with loop interchange.

3.3.1 Replicating Nodes in the Dependence Graph.

When code is replicated in the program text, new nodes are introduced into the
dependence graph. Edges are affected near the replicated statement: some new edges are
introduced and some old edges are removed. Unfortunately the effects of code replication
are not symmetric with respect to the various types of dependence edges. When an
assignment statement is replicated, one of the copies assigns into the original right hand
side while the rest of the copies assign into new variables. Thus, all output dependences
with either endpoint in the original statement will have that endpoint in the copy which
assigns into the original variable. Also, any anti-dependence with source in the statement
to be replicated, will be replicated so that a copy of the edge leaves every copy of the
statement. As a result, replication will not always be effective for alignment conflicts that
include output and anti-dependences.

However, it is the true dependence edges in which we are interested, because these
edges model the flow of data values through a program, in some sense. The other types
of dependences are generated due to memory usage, and we should be able to eliminate these dependences through selective renaming of variables, though this remains an open problem. Hence we will define code replication to model the effects of code replication only on true dependences. This will not affect negative results limiting the usefulness of node replication but will put extra requirements on the positive results.

When we replicate a statement and create \( i \) new copies of that statement, each of these new copies must have the same set of input values as the original statement, hence we copy to each new statement all of the incoming true dependences into the original statement. The purpose of code replication is to allow different uses of the value computed to have a different source; so we partition the outgoing true dependences of the original statement among the original statement and the new statements.

**Definition 3.11** Let \( G \) be a weighted, directed graph and \( v \) a distinguished node of \( G \). Form a new graph \( G' \) by deleting the node \( v \) and replacing it with \( n \) copies, \( v_1, \ldots, v_n \), of itself. For each edge, \( e \), incoming to \( v \) create an incoming edge \( e_i \) to each copy from the same source and with the same weight. For each outgoing edge \( e \) from \( v \), create one edge leaving one of the copies of \( v \) and going to the same sink and having the same weight as \( e \). The transformation of \( G \) into \( G' \) is called Node Replication.

Note that this definition is not accurate with respect to anti-dependences and output dependences.

In general, we may have to replicate more than one node to obtain a graph for which an exact alignment exists. The following property should hold for any graph derived from the original graph by any replication based transformation to guarantee that the new computed values are the same as the original computed values: if \( v' \) in the derived graph computes the same value as \( v \) in the original graph, then every edge into \( v' \) has a dual edge coming into \( v \), further the source of this edge computes the same value as the source of its dual. Graphs which satisfy this property are characterized by the following definition.

**Definition 3.12** A path cover of \( G = (V, E) \) is an ordered pair \( \langle G', \rho \rangle \) such that \( G' = (V', E') \) is a surjective \( (1:1) \) map from \( V' \) to \( V \). The map \( \rho \) extends to \( E' \) (i.e., \( \rho(e) = (\rho(w), \rho(v)) \) and \( \forall e \in E' \rho(e) \in E \)).

Further, if \( v' \in \rho^{-1}(v) \) for some \( v \in V \), then for each edge \( e = (w, v) \) in \( E \), there is a node \( w' \in V' \) and edge \( e' = (w', v') \).

The term path cover reflects the fact that if \( p = e_1 \ldots e_n \) is a path in \( G \), \( v_n = \text{sink}(e_n) \), and \( v'_n \in \rho^{-1}(v_n) \), there is a unique path \( p' = e'_1 \ldots e'_n \) in \( G' \) such that \( v'_n = \text{sink}(e'_n) \).
and for each \( i \), \( e_i = \rho(\epsilon_i) \). The path \( p' \) is called a lift of the path \( p' \). The role of \( \rho \) is to group together statements which perform the same computation. We induce weights on the edges in \( E' \) by the equation \( \text{weight}(e') = \text{weight}(\rho(e)) \).

Our next objectives are determining how to generate path covers for a given graph and how to find a path cover which has an exact alignment.

### 3.3.2 Textual Code Replication

The previous section discusses the effects of code replication on the data dependence graph. This section examines how replication affects the source program, in particular, when can a statement be replicated, what dependences exists between various copies of the new statement and what must be done to maintain the original semantics for iterations near the beginning of the loop.

A basic assumption for replicating a statement \( S \) is that executing that statement twice will produce the same values. A sufficient condition for this to be true is that no memory location modified by the statement is also used as an input. If we can rename all of the outputs of a statement to be unique variables, then this condition will be satisfied. We need to reuse the original outputs in one of the statements, thus we must generate anti-dependences from the other copies to this particular copy, indicating that it must be executed last.

In FORTRAN code without function and subroutine calls, each memory location modified by a statement is explicitly represented by a variable. Hence, it is possible to rename all of the outputs of a statement. This is no longer true when calls to external routines occur in the statement. In such cases: a parameter may be used as both an input parameter and an output parameter and a global common block location may be modified. Statements that perform I/O operations also can not be replicated since "hidden variables" (file pointers, buffers, etc.) are modified. In these cases, the outputs can not be renamed, therefore the statements can not be replicated. This situation is relatively easy to detect and we will assume that it does not occur for the remainder of this section.

The definition of true dependence is intended to capture an execution order constraint, rather than the flow of values through a loop. In particular, a true dependence between two statements does not mean that the second statement uses the value computed by the first; a third statement could modify the memory location that generates this dependence. This latter concept is the one which motivates the definition of path cover. Where this distinction is relevant is in renaming variable references that are the sinks of true dependences from statements that are being replicated. For example, in the following loop, the two definitions of array \( T \) generate two true dependences into each of the uses of \( T \) in the last statement.
DO I = 1,N  
  T(I) = A(I) + B(I)  
  C(I) = T(I)/2  
  T(I) = A(I)*4  
  D(I) = C(I) + T(I-1)  
ENDDO

Despite the apparent alignment conflict involving the first, second and last statements, no replication is required. The reason for this is that the first value assigned into T is dead after the next statement and so the first array T could be named to T1, thus moving the true dependence from the first statement to the second.

More complicated examples can be built using two-dimensional arrays or control flow inside the loop body. In a sense, these true dependences which do not reflect flow of values arise from reuse of memory in the same way that output and anti-dependences do, and therefore can be approached with the same tools: array renaming and array expansion.

Further, in cases like the previous example, the dependence graph can be pruned of these extra edges. In this pruning, the true dependence from the first statement to the last is seen to be transitive with respect to the output dependence between the two definitions of T and the true dependence from the second definition to the uses. When dependences are all consistent, an effective pruning algorithm can be developed. This pruning will help us recognize where array renaming can be applied. From now on we will assume that every reference node has, at most, one incoming true dependence.

Our final concern with textual code replication is that, even with only one incoming true dependence to a reference, the source of that dependence may not generate all of the values used by the sink. In the following loop, all instances of the second statement use values computed by the first except for the one with I = 1.

DO I = 1,N  
  T(I) = A(I)*4  
  D(I) = T(I) + T(I-1)  
ENDDO

In this case, and in general, the first few iterations will use old values of T, rather than the new ones computed in the loop. A correct replication of the first statement is:
DO I = 1,N
   IF (I.GE.2) THEN
      T1 = A(I-1)*4
   ELSE
      T1 = T(I-1)
   ENDIF
   T(I) = A(I)*4
   D(I) = T(I) + T1
ENDDO

The value 2 in the test is in general equal to 1 plus the threshold of the dependence associated with that copy of the replicated statement. Note that the trick of folding the initial iterations into the final ones used to remove guards for loop alignment cannot be used here.

### 3.3.3 Path Covers Revisited

Now we will establish code replication as a powerful tool, in that every graph obtained by node replication is a path cover and the class of alignable path covers can be obtained by replicating a sequence of nodes. In the next section, we will show that we can reduce an arbitrary directed graph to a directed acyclic graph and still retain all the necessary information to generate an alignment for the original graph. With this in mind, we will restrict our attention to directed acyclic graphs.

**Proposition 3.5** If $G' = (V', E')$ is derived from $G = (V, E)$, a directed, acyclic graph, by a single node replication transformation applied to vertex $v$, and $\rho$ maps every node in $G'$ which isn't a copy $v$ to the corresponding node in $G$ and maps every copy of $v$ to $v$, then $(G', \rho)$ is a path cover of $G$.

**Proof:** We simply check each condition of Definition 3.12. From the Definition 3.11 above:

$$V' = V - \{v\} \cup \{v_1, \ldots, v_n\}$$

and clearly $\rho$ is a surjective map from $V'$ to $V$. Further if $e = (u, w) \in E$ and neither $u$ nor $w$ is equal to $v$, then $e \in E'$ and $\rho(e) = e$. If $u = v$, then $e$ is replaced with a single edge $e' = (v, w)$ and then $\rho(e') = e$. Otherwise $w = v$ in which case, there is an edge $e_i = (w, v_i)$ for each new copy and each of these maps to $e$ under $\rho$. This exhausts the edges in $E'$ and so we conclude that $\rho$ extends to $E'$ as defined in Definition 3.12.

Let $w \in V$ be any node other than $v$, then $\rho^{-1}(v) = \{v\}$ If $e = (u, w)$ is an edge incoming to $v$, then if $u \neq v$, then $e \in E'$ and $u$ is a copy of $u$. If $u = v$, then an edge $e' = (v, w)$ was created and $e' \in E$ and $\rho(e') = e$. Thus paths lift and the conclusion follows.

*End of Proof.*
Proposition 3.6 If \( \langle G', \rho \rangle \) is a path cover of \( G = (V, E) \) and \( G \) is acyclic, then \( G' \) can be derived from \( G \) by a sequence no more than \( \|V\| \) node replications.

Proof: Let \( v_1, \ldots, v_n \) be a listing of the nodes of \( G \) in a reverse topological order induced by \( E \), (i.e., \( v_i \) is an ancestor of \( v_j \) only if \( j < i \)). This is the proposed order in which nodes will be replicated. For each \( i \), arbitrarily order the elements of \( \rho^{-1}(v_i) \) and call them \( v_i^1, \ldots, v_i^k \). Define the sequence of graphs, \( G'_n = (V'_n, E'_n) \), by:

\[
V_n = \{ v_i \in V \mid i \leq n \} \\
V'_n = (V - V_n) \cup \rho^{-1}(V_n) \subseteq V \cup V'
\]

and for the edges define:

\[
Int_n = \{ e = (w_i^j, w_k^i) \in E' \mid i \leq n, k \leq n \} \\
Ext_n = \{ e = (w_i, w_k) \in E \mid i > n, k > n \} \\
Bnd_n = \{ e = (w_i, w_k) \mid (w_i, w_k) \in E, i > n, k \leq n \} \\
E'_n = Int_n \cup Ext_n \cup Bnd_n
\]

We want to show that these graphs are the intermediary results of the sequence of node replications. \( V_n \) represents the set of nodes replicated so far, hence \( \rho^{-1}(V_n) \) are the copies of the splits so far. The edges are divided into three groups, the interior edges, \( Int_n \), which are in final form, both of their endpoints having been replicated; the exterior edges, \( Ext_n \), which are in original form, neither of their endpoints having been replicated; and the boundary edges, \( Bnd_n \), which are in an intermediate form having exactly one of their endpoints replicated.

Finally define for all \( v \in V'_n \):

\[
\rho_n(v) = \begin{cases} 
  v, & \text{if } v \in V \\
  \rho(v), & \text{if } v \in V'
\end{cases}
\]

Note that \( G_0, \rho_0 \) = \( G, id \) is trivially a path cover of \( G \) and that \( \langle G \| V \|, \rho \| V \| \rangle = \langle G', \rho \rangle \).

If we show that \( G_n \) can be derived from \( G_{n-1} \) by a single node replication applied to \( v_n \), the conclusion follows.

Consider the symmetric difference of \( V'_n \) and \( V'_{n-1} \):

\[
V'_n \oplus V'_{n-1} = \{ v_n \} \cup \{ v^1_n, \ldots, v^m_n \}
\]
where the last set is $\rho^{-1}(v_n)$. This tells us that we should replicate the node $v_n$ into $m$, copies $\{v^1_n, \ldots, v^m_n\}$.

For edges, let us further divide the edges into the near boundary edges:

$$NBnd_{n-1} = \{e = (w_i, w'_k) \in Bnd_{n-1} \mid i = n\}$$

and the near exterior edges:

$$NExt_{n-1} = \{e = (w_i, w_k) \in Ext_n \mid k = n\}.$$

replicating $v_n$ into $m$ copies which requires that every edge incoming to $n$ be replicated for each copy. The set of edges incoming to $v_n$ is exactly $NExt_{n-1}$ and so these edges become edges in $Bnd_n$. Call the new copies of edges $CNExt_n$. No other exterior edges are effected so the new set of exterior edges is $Ext_{n-1} - NExt_{n-1}$ which is exactly $Ext_n$.

Further, we must partition the outgoing edges of $v_n$ among its direct descendents. The set of outgoing edges is exactly $NBnd_{n-1}$, each of these edges becomes an interior edge and we let $\rho$ guide the partition: replace $(v_n, w'_k)$ with $(v^i_n, w'_k)$ if and only if $(v^i_n, w'_k) \in E'$. Since $(G', \rho)$ is a path cover of $G$, there is exactly one such edge in $E'$. Call these new edges $NInt_n$ and clearly the interior edges after the split are $Int_{n-1} \cup NInt_n$ which is exactly $Int_n$.

Finally, the new boundary edges are the old boundary edges except those into $v_n$ and the new copies of new exterior edges, $Bnd_{n-1} - NBnd_{n-1} \cup CNExt_n$ which is exactly $Bnd_n$.

Clearly the sets, $Int_n, Ext_n$, and $Bnd_n$ include all of the post-replication edges thus, $E'_n$ is precisely the set of post replication edges. Further, since $\rho_n$ is constructed exactly as in Proposition 3.5, we have that $(G'_n, \rho_n)$ is a path cover of $G'_{n-1}$. Thus $G'_n$ can be derived from $G'_{n-1}$ be a single node replication and so $G'$ can be derived from $G$ in at most $\|V\|$ node replications.

*End of Proof.*
3.3.4 Recurrences

The requirement that the graph $G$ be acyclic guarantees the existence of an alignable path cover. That condition is actually too strong; the exact condition tests for the existence of an alignment conflict that consists of edges all of which are in $E$.

**Definition 3.13** Let $G$ be a weighted directed graph. A recurrence is a cycle $p = e_1 \ldots e_n$ in $G$ such that $\text{weight}(p) \neq 0$.

Note that this definition of recurrence agrees with that given in Definition 1.2 of in that if $G$ is the level $k$ subgraph of the dependence graph of a program and the weights are specified by the $k$-threshold, then a cycle in $D_k$ is a level $k$ recurrence in the sense of Definition 1.2 if and only if it is a recurrence in the sense of Definition 3.13. Now we can formalize the intuition that a loop which carries a recurrence cannot be run in parallel.

**Proposition 3.7** A graph $G$ has an alignable path cover only if $G$ has no recurrences.

**Proof:** Assume that $(G', \rho)$ is a path cover for $G$ and $p = e_1 \ldots e_n$ is a recurrence in $G$. Let $v_1, \ldots, v_n$ be the nodes along the cycle such that $e_i = (v_i, v_{i+1})$ assuming $v_{n+1} = v_1$. We will construct a path $p'$ in $G'$ that is a sequence of one or more lifts of $p$ and is an alignment conflict. This shows that $G'$ is not alignable and establishes the result.

Arbitrarily order the elements of $\rho^{-1}(v_1)$ and label them $w_1^1, \ldots w_1^{m_1}$. Since $(G', \rho)$ is a path cover for $G$, there is a node in $\rho^{-1}(v_n)$, call it $w_n^1$, such that $e_n^1 = (w_n^1, w_1^1)$ and $\rho(e_n^1) = e_n$. Iterating, we can select $w_j^1$ so that $e_j^1 = (w_j^1, w_{j+1}^1)$ and $\rho(e_j^1) = e_j$. Thus we lift the path $p$ to a path $p'$. Note that we are going backwards through the cycle, tracing along inputs to nodes.

We cannot guarantee that the sink of $p'$ is $w_1^1$, but it is at least in $\rho^{-1}(v_1)$. If it is $w_1^1$, then we are done, since then $p'$ is a cycle and $\text{weight}(e_j^1) = \text{weight}(e_j)$ for each $i$ implies $\text{weight}(p') = \text{weight}(p) \neq 0$ and so $G'$ has an alignment conflict and hence is not alignable. Otherwise, label the endpoint $w_1^2$ and repeat the above process beginning at $w_1^2$ and lift $p$ to $p^2$. Repeating this process defines a sequence of paths $p^1, p^2, \ldots$ each of which is lift of $p$ and the sink of each $p^i$ is the source of $p^{i+1}$.

Since the sources of these lifts all lie in $\rho^{-1}(v_1)$, which is finite, eventually, the sequence $w_1^1, w_1^2, \ldots$ of endpoints must repeat. Let $w_1^i$ be the first occurrence of a repeated node
and \( \omega_1^{i+k} \) the second occurrence of that same node. The path \( p' = p^i p^{i+1} \ldots p^k \) is a cycle in \( G' \) and:

\[
weight(p') = \sum_{j=i}^{k} p^j = k \cdot weight(p) \neq 0
\]

and hence \( p' \) is an alignment conflict in \( G' \) and so \( G' \) is not alignable.

\textit{End of Proof.}

### 3.3.5 The \( \pi \)-Graph

The converse of Proposition 3.7, that a graph without recurrences has an alignable cover is true, a fact we will prove shortly. The basic problem is how to handle cycles in the dependence graph. We shall present a method which reduces a general graph to the acyclic case and then give an algorithm based on node replication that generates an alignable path cover. Proposition 3.6 is the guide to this basic algorithm, it not only guarantees that if an alignable path cover exists, then we can derive it with node replication, it also suggests the algorithm. First we give the important reduction.

**Definition 3.14** Let \( G \) be an arbitrary graph. Let \( \pi_1, \ldots, \pi_n \) be the maximal strongly connected regions of \( G \). Form a new graph, \( \Pi_G = (V_\pi, E_\pi) \), the vertices of which are the maximal strongly connected regions of \( G \). The edges are defined by \( e' = (\pi_i, \pi_j) \in E_\pi \) if and only if there exists an edge \( e = (v, w) \in E \) such that \( v \in \pi_i \), \( w \in \pi_j \) and if \( i = j \), \( weight(e) \neq 0 \). The weight of an edge is the weight of the edge in \( G \) which gave rise to it. This graph is called the \( \pi \)-graph of \( G \).

Note that we are considering an edge to be determined by its endpoints and its weight, so assume that each edge in \( E_\pi \) represents a set of edges in \( E \) that have source in the same strongly connected region and sink in the same strongly connected region and same weight. Further, we ignore edges contained in a single strongly connected region that have 0 weight. We do this because these edges give no information for purposes of alignment – all they imply is that a node must be assigned a single alignment value. Further, this extra condition allows the following proposition.
**Proposition 3.8** If \( G \) has no recurrences, then \( \Pi_G \) is acyclic.

*Proof:* If \( p \) is a cycle in \( \Pi_G \), then it must contain only a single node, otherwise there would be a cycle in \( G \) which includes nodes from more than one maximal strongly connected regions, contradicting the definition of maximal strongly connected region. Since \( p \) consists of only a single node, there exists an edge from that node to itself. By construction, there exists an edge \( e = (v, w) \in E \) with both endpoints in that region such that \( \text{weight}(e) \neq 0 \). By definition of strongly connected region, there exists a path \( p \) in \( G \) from \( w \) to \( v \). Since the weights on \( G \) are non-negative,\(^3\) we have:

\[
\text{weight}(ep) = \text{weight}(e) + \text{weight}(p) \geq \text{weight}(e) > 0
\]

thus \( ep \) is a cycle from \( v \) to \( v \) with non-zero weight and hence is a recurrence.

*End of Proof.*

Our intention, of course, is to generate an alignable cover for \( \Pi_G \) and use it to construct an alignable cover for \( G \). This second transformation is a straightforward but tedious process of replacing each node \( v' \) in the cover of \( \Pi_G \) with the associated strongly connected region of \( G \) associated with the node \( v \) in \( \Pi_G \) that \( v' \) covers, and patching the edges together. Thus I will assume the following proposition and omit the details of the construction.

**Proposition 3.9** If \( \Pi_G \) has an alignable cover, then \( G \) has an alignable cover.

### 3.3.6 Back to DAG's

Assuming that \( G \) is a directed, weighted and acyclic graph, we can find an alignable path cover using node replication. The algorithm resembles Proposition 3.4 in that we pick some nodes and assign them an alignment coefficient of 0, then define alignment coefficients for other nodes based on the paths from them to the special nodes, thus guaranteeing that the final graph is alignable. It also resembles Proposition 3.6 in that we replicate nodes in topological order so that the subgraph determined by the nodes replicated so far is a subgraph of the final path cover.

---

\(^3\) This is the only place that we have used the non-negative requirement on the weights of \( G \). The requirement is necessary otherwise we would be unable to distinguish between recurrences and simple alignment conflicts.
Algorithm 1. Computation of alignable path cover.

Input. A weighted, graph $G$.

Output. A graph $G'$, a function $\rho: G' \rightarrow G$ and a function $\sigma$ mapping the nodes of $G'$ into the integers.

Method. Apply the procedure $dfs\_align$ given in Figure 12 with $COUNT$ initialized to 1 and every node unmarked, until every node is marked.

Proposition 3.10 For $G, G', \rho$, and $\sigma$ as in Algorithm 1. $(G', \rho)$ is a path cover for $G$ and $\sigma$ is an exact alignment for $G'$.

Proof: In this proof, the nodes in $G$ will be referred to by the ordering given by $\delta$ (i.e., the node assigned number $i$ is referred to as $v_i$) and, likewise, the nodes in $G'$ will be referred to by the number assigned by $\rho$ and the value assigned by $\sigma$ (i.e., if $\rho(v) = v_i$ and $\sigma(v) = k$, then $v$ is called $v_i^k$). Let $\hat{G} = (\hat{V}, \hat{E})$ be the graph as it changes during execution of the program and likewise, $\hat{\rho}$ is the function $\rho$ as it changes.

The ordering also allows us to define the same sequence of graphs, $G'_{n} = (V'_n, E'_n)$ as in Proposition 3.6. Clearly, $G'_{n} = \hat{G}$ and $\rho_n = \hat{\rho}$ after the call to $dfs\_align$ on vertex $v_n$. Note that the transformation performed by one call to $dfs\_align$ is in fact a node replication, therefore, by Proposition 3.5, $(G', \rho)$ is a path cover of $G$.

All we have left to show is that $\sigma$ is in fact an exact alignment. Let $e = (v_i^k, v_j^l)$ be an edge in $G'$, then immediately before the replication of node $v_i$, there was an edge $e' = (v_i, v_j^l)$ in $G'_{n-1}$. By construction, we know that $k = \sigma(v_j^l) - weight(e')$, $\sigma(v_i^k) = k$, and $weight(e) = weight(e')$ hence $\sigma(v_i^k) = \sigma(v_j^l) - weight(e)$ or:

$$weight(e) = \sigma(sink(e)) - \sigma(source(e)).$$

Since this holds for all $e \in E'$, we have $\sigma$ is an exact alignment for $G'$.

End of Proof.

Corollary Every directed, weighted, acyclic graph $G$ has an alignable cover.

Proof: Apply Algorithm 1 to $G$.

Now we have all the pieces for the main result of this section.

Theorem 3.3 $G$ has an alignable path cover if and only if $G$ has no recurrences.

Proof: The 'only if' part is Proposition 3.7. If $G$ has no recurrences, then by Proposition 3.8, $\Pi_G$ has no recurrences. Hence by the corollary to Proposition 3.10, $\Pi_G$ has an alignable path cover and so by Proposition 3.9, $G$ also has an alignable path cover.

End of Proof.
procedure dfs_align(v)
    /* Basic Depth First Search code */
    mark(v)
    for each edge e = (v, w) do
        if not_marked(w)
            then dfs_align(w)
    /* Assign topological ordering */
    δ(v) ← COUNT
    COUNT ← COUNT + 1
    /* Build replication guide */
    I ← Ø
    for each edge e = (v, w) do
        I ← I ∪ {σ(w) − weight(e)}
    if I = Ø
        then I ← {0}
    /* Replicate node */
    for each i ∈ I do
        create node v^i
        mark(v^i)
    /* Duplicate Inputs */
    for each edge e = (w, v) do
        for each i ∈ I do
            create edge e' = (w, v^i) with the same weight as e
            delete edge e
    /* Partition Outputs */
    for each edge e = (v, w^j) do
        i ← σ(w^j) − weight(e)
        create edge e' = (v^i, w^j) with the same weight as e
        delete edge e
    delete node v
    /* Define alignment and covering functions */
    for each i ∈ I do
        σ(v^i) ← i
        ρ(v^i) ← v
end dfs_align

Figure 12. Algorithm 1. Compute alignable path cover.
Then, if we ignore certain technical difficulties, we can recast this result into terms about the dependence graph.

**Theorem 3.4** Let \( P \) be a program fragment consisting of \( n \) statements inside \( m \) common DO loops. Then we can apply node replication and loop alignment to obtain a semantically equivalent program fragment such that the separate iterations of the \( k^{th} \) outermost DO loop can be run concurrently if and only if there are no level \( k \) recurrences.

The "technical difficulties" include:

1. All dependences must be consistent.
2. All variable references must be alignment sensitive.
3. The dependence graph can be pruned so that no reference has more than one incoming true dependence.
4. All dependences must be true dependences.

The first two of these are not as significant as the last two; to satisfy the third and fourth condition may require explicit copy statements where sources of values cannot be resolved. This area requires further investigation.

### 3.4 Minimal Alignable Path Covers

Algorithm 1 generates an alignable path cover for any DAG but it does not guarantee that that path cover is the smallest alignable path cover with respect to the number of nodes in the graph. In this section it will be shown that finding minimal path covers for arbitrary DAG's is a hard problem. Here, "minimal" refers to the number of nodes in the path cover. Further, is will be shown that there exist DAG's with the property that the minimal alignable path cover is exponentially larger than the DAG itself. Finally, it will be shown that even in the case for which we can show a polynomial bound on the size of the minimal alignable path cover, finding that path cover is an NP-complete problem.

For the first of these two problems, we define the following family of DAG's:

\[
G_n = (V_n, E_n)
\]

\[
V_n = \{v_0, \ldots, v_n\}
\]

\[
E_n = \{e = (v_i, v_{i-1}) \mid 0 < i \leq n \text{ and } \text{weight}(e) \in \{0, 2^i\}\}
\]

\( G_n \) has \( n + 1 \) nodes and \( 2n \) edges. See Figure 13.
Proposition 3.11 Let \((G'_n, \rho_n)\) be an alignable path cover of \(G_n\) and \(\sigma\) an exact alignment of \(G'_n\). Let \(w_0 \in \rho^{-1}(v_0)\) then for all \(i, 0 \leq i \leq n\) and \(j, 0 \leq j < 2^i\), there exists a node \(w \in G'_n\) such that \(\rho(w) = v_i\) and \(\sigma(w) = \sigma(w_0) + j\).

Proof: By induction on \(i\). For \(i = 0\), we have \(j = 0\) and take \(w = w_0\). So the base case \(i = 0\) is true. Now assume that for \(i = k\), the conclusion of the proposition is true. Let \(j\) be in the range \(0 \text{ to } 2^{k+1} - 1\) and let \(j'\) be equal to \(j\) modulo \(2^k\). By the induction hypothesis, there exists a node \(w_k \in G'_n\) such that \(\rho(w_k) = v_k\) and \(\sigma(w_k) = \sigma(w_0) + j'\). Note that there are two edges from \(v_{k+1}\) to \(v_k\) with weights 0 and \(2^k\). Since \(G'_n\) is a path cover, there exist nodes \(w\) and \(w'\) in \(G'_n\) such that \(\rho(w) = \rho(w') = v_{k+1}\) and edges \(e = \langle w, w_k \rangle\) and \(e' = \langle w', w_k \rangle\) in \(G'_n\) such that \(weight(w) = 0\) and \(weight(w') = 2^k\). Since \(\sigma\) is an alignment for \(G'_n\), we have \(\sigma(w) = \sigma(w_k) = \sigma(w_0) + j'\) and \(\sigma(w') = \sigma(w_k) + 2^k = \sigma(w_0) + j' + 2^k\). Thus if \(0 \leq j < 2^k\) then \(j = j'\) and so \(w\) satisfies the conclusion and otherwise \(j = j' + 2^k\) and \(w'\) satisfies the conclusion. Thus for \(i = k + 1\) and \(0 \leq j < 2^i\), there exists a node \(w \in G'_n\) such that \(\rho(w) = v_{k+1}\) and \(\sigma(w) = \sigma(w_0) + j\). The proposition follows by induction.

End of Proof.

Corollary There exist DAG's with \(O(n)\) nodes and \(O(n)\) edges such that the minimal alignable path cover has \(O(2^n)\) nodes.

Proof: Immediate from previous proposition.

The exponential size of the minimal alignable path cover is very dependent on the exponential growth in the weights of the input graphs. In fact, we can restrict the size of the minimal alignable path cover for a weighted DAG \(G = (V, E)\) to be:

\[ ||V||^2 \cdot \max\{\text{weight}(e) \mid e \in E\}. \]
In the case above, the second term was $2^{|V|}$ and so the exponential growth was possible. This bound can be derived in a straightforward way from Algorithm 1. To show that this bound is reasonably tight, consider a modification of the graphs used above:

$$G_n = (V_n, E_n)$$

$$V_n = \{v_0, \ldots, v_n\}$$

$$E_n = \{e = (v_i, v_{i-1}) \mid 0 < i \leq n \text{ and } \text{weight}(e) \in \{0, 1\}\}$$

In these graphs, the maximum edge weight is 1 and the minimum alignable path cover has $i + 1$ copies of $v_i$ and so has $n(n + 1)/2$ nodes.

Even if we restrict our attention to DAG's with relatively small minimal alignable path covers, finding such a cover is NP-complete. To show this, I will construct a class of DAG's which have small minimal alignable path covers, pose a problem over this graph class which is equivalent to finding the minimal alignable path cover, give a linear time non-deterministic algorithm for solving that problem and finally give a reduction of the 3-CNF Satisfiability problem (3 SAT) to the given problem. The graph collection is defined by:

**Definition 3.15** Let $S$ be the collection of weighted DAG's $G$ such that every node is either a source or a sink for edges but not both.

With the help of the following lemmas, we show that a DAG $G = (V, E)$ in $S$ has an alignable path cover with not more than $|V| + |E|$ nodes. The next two lemmas show that a minimal alignable path cover has no spurious copies of nodes. We use these to show that graphs in $S$ have small minimal path covers and that the modified Algorithm 1 can generate a minimal alignable path cover if the correct initial alignment values are chosen for 'sink' nodes.
Figure 15. Example graph from class $S$

**Definition 3.16** Let $G$ be a directed graph, then define $\text{Sink}(G)$ to be the set of nodes in $G$ with no outgoing edges.

**Lemma 3.2** If $(G', \rho)$ is a minimal alignable path cover of $G$, then $\rho$ restricted to $\text{Sink}(G')$ is bijective (1-1 and onto).

*Proof:* Assume that $\rho$ is not bijective. Hence, either there are two distinct nodes, $w_1$ and $w_2$ in $\text{Sink}(G')$ such that $\rho(w_1) = \rho(w_2) = v$ and $v \in \text{Sink}(G)$ or there exists $w_1 \in \text{Sink}(G')$ such that $\rho(w_1) \notin \text{Sink}(G)$. In either case, we could delete the node $w_1$ and all of its incoming edges from $G'$ to get a new path cover $G''$. This new path cover is also alignable but has fewer nodes than a minimal alignable path cover for $G$, a contradiction. Therefore, if $(G', \rho)$ is a minimal alignable path cover than $\rho$ restricted to $\text{Sink}(G')$ is bijective.

*End of Proof.*

Based on the last lemma, we can refer to $\rho^{-1}(v)$ unambiguously when $v \in \text{Sink}(G)$ and $G'$ is minimal.

**Lemma 3.3** If $(G', \rho)$ is a minimal alignable path cover for $G$, and $\sigma$ is an exact alignment for $G'$, then for any nodes $w_1, w_2 \in V'$ if $\rho(w_1) = \rho(w_2)$ and $\sigma(w_1) = \sigma(w_2)$ then $w_1 = w_2$.

*Proof:* As above, if $\rho(w_1) = \rho(w_2)$ and $\sigma(w_1) = \sigma(w_2)$, we could make the outgoing edges of $w_2$ be outgoing edges from $w_1$, then delete $w_2$ and its incoming edges. The resulting graph is also an alignable path cover of $G$ but has fewer nodes than a $G'$, contradicting the assumption that $G'$ is a minimal alignable path cover. Therefore, $w_1 = w_2$.

*End of Proof.*
Proposition 3.12 If $G = (V, E) \in S$, then a minimal alignable path cover of $G$ has at most $\|V\| + \|E\|$ nodes.

Proof: Let $(G', \rho)$ be a minimal alignable path cover of $G$. By Lemma 3.2, $\|\rho^{-1}(V_0)\| = \|V_0\|$ and for graphs in $S$, if $v \notin V_0$ then the in-degree of $v$ is 0. Since each node $w \in \rho^{-1}(v)$ has the same in-degree as $v$ for any $v$, the sum of the in-degrees of the nodes in $G'$ is equal to the sum of the in-degrees of the nodes in $G$. Thus $G'$ has the same number of edges as $G$. If $v' \notin \rho^{-1}(V_0)$, then $\rho(v')$ has an outgoing edge by Lemma 3.3. Therefore, every node in $G'$ without outgoing edges is in $\rho^{-1}(V_0)$. As a result, the number of nodes in $G'$ is bounded above by $\|V_0\| + \|E\|$ which is bounded by $\|V\| + \|E\|$.

End of Proof.

Now we pose a problem on weighted DAG's which is closely related to finding minimal alignable path covers. We will then show that this problem is NP-complete.

Definition 3.17 A weighted DAG is $k$-alignable if it has an alignable path cover with at most $k$ nodes.

To show that this problem is solvable non-deterministically in polynomial time, we modify Algorithm 1. In the section of code labeled with '/* Build replication guide */', the initialization of $I$ to $\{0\}$ for nodes with no outgoing edges is arbitrary and we replace it with a singleton set containing a non-deterministically selected integer. These selections determine the rest of the path cover. Clearly the running time of this algorithm is linear in the size of the path cover constructed.

Next we must show that every minimal alignable path cover can be constructed by the modified Algorithm 1.

Proposition 3.13 Let $G$ be a DAG, $(G'', \rho, \sigma')$ be a minimal alignable path cover of $G$, and $\sigma'$ an alignment of $G''$. If the modified Algorithm 1 is applied to $G$, and if the initial value for a node $v' \in \text{Sink}(G')$ is taken to be $\sigma'(\rho^{-1}(v_0))$, then the output of the algorithm, $G'$, $\rho$, and $\sigma$ will equal $G''$, $\rho'$ and $\sigma'$ respectively.

Proof: To show equality, we will construct a map $f$ from $G'$ into $G''$ which is bijective and preserves $\rho$ and $\sigma$, i.e., for all $v, w \in G'$:

$$(v, w) \in G' \iff (f(v), f(w)) \in G''$$

$$\rho(v) = \rho'(f(v))$$

$$\sigma(v) = \sigma'(f(v)).$$
Define $V_k$ to be the set of nodes $v$ in $G$ such that the shortest path from $v$ to a node in $\text{Sink}(G)$ has $k$. Define $V'_k$ and $V''_k$ analogously for $G'$ and $G''$. Let $G'_k$ be the subgraph of $G'$ consisting of all of the nodes in $\bigcup_{i=0}^{k} V'_i$ and all of the edges between them. We will construct $f$ inductively by first constructing the restriction of $f$ to $G'_0$ then incrementally extending that restriction to $G'_k$.

Note that in Algorithm 1, a node $v'$ is created only if it is the only copy of a node $v$ in $\text{Sink}(G)$, in which case $\sigma(v') = \sigma'(\rho^{-1}(v))$, or there is already a node $y'$ in $G'$ and an edge $e$ in $G$ which is incoming to $\rho(y')$ from $v$. In this case, one copy, $v'$, of $v$ is created and $\rho(v')$ is set to $v$ and $\sigma(v') = \sigma(y') + \text{weight}(e)$. From this we can conclude that the conclusions of the two previous lemmas hold for $G'$ and $\rho$. Since $\rho'$ restricted to $V''_0$ is also bijective, define $f_0 = \rho'^{-1} \circ \rho$. Clearly $f_0$ is a bijective function from $V''_0$ to $V''_0$ which preserves $\rho$ and $\sigma$.

Assuming that $f_k$ is a bijective function from $G'_k$ into $G''_k$ which preserves the restrictions of $\rho$ and $\sigma$. $f_{k+1}$ will be an extension of $f_k$, so we only need to define it on the nodes in $V'_{k+1}$ and edges leaving those nodes. Let $v'$ be a node in $V'_{k+1}$. By definition, there must be a node $w'$ in $V'_k$ and an edge $e' = \langle v', w' \rangle$. Let $v = \rho(v')$, $w = \rho(w')$, $e = \rho(e')$, and $w'' = f_k(w')$. By the induction hypothesis, $w = \rho'(w'')$ and $\sigma'(w'') = \sigma(w')$. Since $\langle G'', \rho' \rangle$ is a path cover of $G$, there exists a node $v''$ such that $e'' = \langle v'', w'' \rangle \in G''$ and $\rho'(e'') = e$. Define $f_{k+1}(v') = v''$. Note that $\rho$ and $\sigma$ are preserved:

$$
\rho(v') = v = \rho'(v'')
$$

$$
\sigma(v') = \sigma(w') + \text{weight}(e')
= \sigma(f_k(w')) + \text{weight}(e)
= \sigma'(w'') + \text{weight}(e'')
= \sigma'(v'')
$$

To show that $f_{k+1}$ is well defined, let $w'_1$ and $w'_2$ be two nodes in $V'_k$ such that both $e'_1 = \langle v', w'_1 \rangle$ and $e'_2 = \langle v', w'_2 \rangle$ are in $G'$. Let $e'_1$ be the lift of $\rho(e')$ with sink at $f_k(e'_1)$ and $e'_2$ be the lift of $\rho(e')$ with sink at $f_k(e'_2)$. Then let $v''_1 = \text{source}(e'_1)$ and $v''_2 = \text{source}(e'_2)$ and:

$$
\sigma'(v''_1) = \sigma(w''_1) + \text{weight}(e''_1)
= \sigma(w'_1) + \text{weight}(e'_1)
= \sigma(v')
= \sigma(w'_2) + \text{weight}(e'_2)
= \sigma'(w''_2) + \text{weight}(e''_2)
= \sigma'(v''_2)
$$
and by Lemma 3.3, $v''_1 = v''_2$.

To show that $f_{k+1}$ extends to edges, let $e' = (v', w') \in G'$. Let $e'' = (v'', w'')$ be the lift of $\rho(e')$ ending at $w'' = f_{k+1}(w')$, then:

$$\rho'(v'') = \text{source}(\rho(e'))$$
$$= \rho(v')$$
$$= \rho'(f_{k+1}(v'))$$

and:

$$\sigma'(v'') = \sigma'(w'') + \text{weight}(e'')$$
$$= \sigma(w') + \text{weight}(e')$$
$$= \sigma(v')$$
$$= \sigma'(f_{k+1}(v'))$$

and hence $v'' = f_{k+1}(v')$ and so $f_{k+1}$ extends to edges.

To show that $f_{k+1}$ is 'onto', we perform the construction in reverse. To show that $f_{k+1}$ is 1-1, assume $w'_1$ and $w'_2$ are in $G'$ and $f_{k+1}(w'_1) = f_{k+1}(w'_2)$, then:

$$\rho(w'_1) = \rho'(f_{k+1}(w'_1))$$
$$= \rho'(f_{k+1}(w'_2))$$
$$= \rho(w'_2)$$

and:

$$\sigma(w'_1) = \sigma'(f_{k+1}(w'_1))$$
$$= \sigma'(f_{k+1}(w'_2))$$
$$= \sigma(w'_2)$$

and hence $w'_1 = w'_2$.

Let $n$ be the number of nodes in $G$, then $G'_n = G'$ and $G''_n = G''$. I conclude that if $f = f_n$ then $f$ is a bijective function from $G'$ to $G''$ which preserves the projection and alignment functions of $G'$ and $G''$. Therefore $G'$ is isomorphic with $G''$ and $\rho = \rho'$ and $\sigma = \sigma'$.

End of Proof.
Note that if $G = (V, E)$ is in $S$, then Algorithm 1 runs in time linear in $|V| + |E|$. If $G$ is $k$-alignable, then there is a minimal alignable path graph with less than or equal to $k$ nodes. The modified Algorithm 1 can find this graph non-deterministically in linear time. Thus we have proved the following proposition.

**Proposition 3.14** $k$-alignability of graphs in $S$ is solvable non-deterministically in polynomial time.

To show that $k$-alignability is NP-hard, we will give a reduction of 3 CNF Satisfiability to $k$-alignability. Let:

$$B = \prod_{i=1}^{n} F_i$$
$$F_i = l_{i_{1}} + l_{i_{2}} + l_{i_{3}}$$

where each $l_{i_{j}}$ is a literal over the set of variables $\{v_{1}, \ldots, v_{m}\}$. Assume further that no variable appears twice in the same factor. The choices in this problem are the assignment of the values TRUE and FALSE to the logical variables. We will encode this problem into a DAG, $G_B$, which is $k$-alignable for a particular $k$ if and only if the expression $B$ is satisfied by some assignment of logical values to logical variables.

First, there is one node for each variable. Each of these nodes is labeled with the variable associated with it. I will refer to these nodes as the "variable" nodes in $G_B$. These nodes will have no outgoing edges and will be subject to non-deterministic selection of alignment values by the modified Algorithm 1. Subject to shifting the entire alignment by a constant, we will encode a FALSE value for variable $v_i$ by assigning an alignment value of $3^i$ to the node labeled $v_i$. Likewise we will encode a TRUE value as $2 \cdot 3^i$. 

![Diagram of nodes and nodes related by edges representing the encoding process.](image-url)
Figure 17. Nodes $v_i, v_j$ and $v_k$ and part of a factor node set in $G_B$ for $F_i = v_i + \bar{v}_j + v_k$

To ensure that nodes are assigned values corresponding to TRUE and FALSE, we add another set of nodes, $w_{i,j}$, each of which has four outgoing edges. There is one of these nodes for each unordered pair $i, j$, so $w_{i,j} = w_{j,i}$. Two edges go to $v_i$ and are a weighted $3^i$ and $2 \cdot 3^i$. The other two edges go to $v_j$ and are weighted $3^j$ and $2 \cdot 3^j$. If both endpoints are assigned values whose difference corresponds to TRUE or FALSE, then $w_{i,j}$ can be split into three copies to break the alignment conflicts. If any other combination of values are selected, then $w_{i,j}$ has to be split into four copies.

To encode the fact that each factor $F_i$ in $B$ must be satisfied, we create seven nodes, each corresponding to a possible combination of true literals which makes the factor true. We will refer to these nodes as "literal" nodes in $G_B$. We will label each of these nodes with $f_{ik}$ where $k$ is in the range 1 to 7 and represented in binary. Thus the node $w_{110}$ corresponds to the condition that literals $l_{i_1}$ and $l_{i_2}$ are true and $l_{i_3}$ is false in factor $F_i$. From each of these nodes, there is an edge to each variable appearing in a literal in $F_i$. These edges are weighted so that if the logical variables are assigned values such that the value of the literals agrees with the binary label of the node, there are no alignment conflicts for this node. The following table shows how edges should be weighted for edges to variable $v_i$. Column headings indicate whether the literal in $F_i$ is the variable or the variable negated. Row headings indicate whether that literal should be true or false for this node as indicated by the binary label.

<table>
<thead>
<tr>
<th>$v_i$</th>
<th>$\bar{v}_i$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 \Rightarrow FALSE</td>
<td>$3^i$</td>
</tr>
<tr>
<td>1 \Rightarrow TRUE</td>
<td>$2 \cdot 3^i$</td>
</tr>
</tbody>
</table>
Clearly we can satisfy all of the alignment requirements of at most one of these seven nodes. Three of the remaining nodes will have two different alignment values and the other three will have three different alignment values. Thus if we satisfy one term, we must have at least 16 nodes to satisfy the alignment conflicts of these seven nodes.

From the construction, a minimal alignable path cover should have \( m \) nodes to cover the "variable" nodes, \( 3m(m - 1)/2 \) to cover the \( w_{i,j} \) nodes and \( 16n \) nodes to cover the "literal" nodes. Now we must show that the graph \( G_B \) has an alignable path cover with \( m + \frac{3}{2}m(m - 1) + 16n \) nodes if and only if the boolean expression \( B \) is satisfiable. One direction is fairly straightforward.

**Proposition 3.15** If \( B \) is satisfiable, then \( G_B \) is \( (m + \frac{3}{2}m(m - 1) + 16n) \)-alignable.

**Proof:** Let \( L: \{v_1, \ldots, v_m\} \mapsto \{\text{TRUE, FALSE}\} \) be a truth assignment which satisfies \( B \). We will now construct path cover \( (G', \rho) \) for \( G_B \) and an alignment \( \sigma \) for \( G' \). First, \( G' \) contains one copy of each of the nodes in \( G_B \) labeled with a variable, \( \rho^{-1}(v_i) = \{v_i'\} \). If \( L(v_i) \) is True, then \( \sigma(v_i') = 2 \cdot 3^i \), otherwise \( \sigma(v_i') = 3^i \).

\( G' \) contains three copies of each of the \( m(m - 1)/2 \) nodes \( w_{i,j} \). For a particular \( w_{i,j} \), one of the copies, \( w_{i,j}' \), is aligned by 0, \( \sigma(w_{i,j}') = 0 \), and has two outgoing edges. Each of the other two copies, \( w_{i,j}^1 \) and \( w_{i,j}^2 \), are aligned by \( \pm 3^j \) and \( \pm 3^j \) respectively and has one outgoing edge. The exact alignment value of these nodes depends on the truth value of \( v_i \) and \( v_j \). For example, if \( f(v_i) \) is True, then \( \sigma(v_i) = 2 \cdot 3^i \) and \( v_i \) has an incoming edge in \( G_B \) with weight \( 3^i \), thus \( \sigma(w_{i,j}^1) \) must be \( -3^i \) to satisfy the alignment requirement. The other cases are analogous.

The copies of the \( f_{i,k} \) nodes are more complicated. Since \( B \) is satisfied by \( L \), exactly one, \( f_{i,k}' \), of the \( f_{i,k} \) nodes corresponds to the combination of true and false literals determined by \( L \). Since the weights assigned to the edges from this node are each equal to the alignment value of the variable node they go to, each of these edges dictates that \( \sigma(f_{i,k}') = 0 \). Thus we need only one copy of this node to satisfy its alignment requirements. For three of the other \( f_{i,k} \) nodes, two of the edges will require an alignment of 0; the third will require a non-zero alignment. Thus we need two copies of these nodes, one aligned by 0 and the other aligned by \( \pm 3^j \) for some \( j \). The remaining three \( f_{i,k} \) nodes will have one alignment requirement of 0, since one of the literals will be "true", and two non-zero and non-equal alignment requirements generated by the two "false" literals. As a result,
these nodes will have to have three copies. The alignment $\sigma$ is constructed to satisfy these requirements.

This completes the construction of $G'$. Let $\rho$ be the projection function implicit in the construction. $(G', \rho)$ is a path cover of $G_B$ and $\sigma$ is an alignment for $G'$. There is one copy for each of the $m$ $v_i$ nodes in $G_B$, three copies of each of the $m(m-1)/2$ $w_{i,j}$ nodes and sixteen nodes for each set of seven $f_{i,k}$ nodes. Therefore $G'$ has $m + \frac{3}{2}m(m-1) + 16n$ nodes and so $G_B$ is $(m + \frac{3}{2}m(m-1) + 16n)$-alignable.

End of Proof.

To show the converse, that $B$ is satisfiable only if $G_B$ is $(m + \frac{3}{2}m(m-1) + 16n)$-alignable, we begin by showing that an alignable path cover of the subgraph consisting of a set of seven $f_{i,k}$ nodes and the three variable nodes involved is minimal only if it has nineteen nodes and one of the $f_{i,k}$ has a single inverse image. This will allow us to assert that the factor corresponding to the seven $f_{i,k}$ nodes is satisfied.

**Lemma 3.4** Let $G_f$ be a subgraph of $G_B$ consisting of the seven nodes $f_{i,001}$, $\ldots, f_{i,111}$ corresponding to factor $F_i$ in $B$ together with the three nodes corresponding to variables in $F_i$, and all of the edges between them. Let $(G', \rho)$ be a minimal alignable path cover of $G_f$, then $G'$ has nineteen nodes and for some node $f_{i,k}$, $||\rho^{-1}(f_{i,k})|| = 1$.

**Proof:** By the construction in the previous proposition, we know that $G_f$ has an alignable path cover with nineteen nodes, hence a minimal alignable path cover can have no more than that. Further, if there is some node $f_{i,l}$ such that $||\rho^{-1}(f_{i,l})|| = 1$, then by the argument in the previous proposition, $G'$ has exactly nineteen nodes. So assume that for every node $f_{i,k}$, $||\rho^{-1}(f_{i,l})|| > 1$. There must be at least five nodes with $||\rho^{-1}(f_{i,l})|| = 2$, otherwise $G'$ would have more than nineteen nodes and not be minimal.

Let $v_{i_1}, v_{i_2}$, and $v_{i_3}$ be the labels of the nodes corresponding to variables in $G_f$ and let $v'_1, v'_2$, and $v'_3$ be the labels of the lifts of these nodes to $G'$. Let $\sigma$ be an alignment of $G_f$ such that $\sigma(v'_a) \in \{3^a, 2 \cdot 3^a\}$ for $a = i_1, i_2, i_3$. By assumption, $||\rho^{-1}(f_{i,l})|| > 1$ for each $l = 001, \ldots, 111$ which is equivalent to saying that the assignment of values to the logical variables indicated by $\sigma$ does not satisfy any of the literal combinations. One of the literal combinations must be "false" with respect to every variable, so it has three distinct non-zero alignment requirements. Three of the literal combinations are "true" with respect to exactly one variable, therefore they have one zero and two distinct non-zero alignment
requirements. The remaining three literal combinations are “true” with respect to exactly
two variables and have two zero and one non-zero alignment requirement. As a result,
four of the literal combinations must be split into three copies and three must be split into
two copies. This indicates that \( G' \) has twenty-one nodes, contradicting the assumption
that \( G' \) was minimal. Thus \( \sigma(v'^a_i) \notin \{3^i, 2 \cdot 3^i\} \) for one of \( a = i_1, i_2, i_3 \).

By assumption, there are at least five nodes, \( f_{i,t} \), with \( \|\rho^{-1}(f_{i,t})\| = 2 \). Since there
is one edge from each \( f_{i,t} \) to each of the variable nodes, there are at least five nodes in
\( G' \) which have two outgoing edges to two different variable nodes. Let \( w'_1 \) and \( w'_2 \) be two
such nodes in \( G' \) such that, after possible relabeling, both of them have edges to \( v'_{i_1} \) and
\( v'_{i_2} \). Let \( \alpha_1, \alpha_2, \beta_1, \text{ and } \beta_2 \) be the weights of the edges as indicated in Figure 18.

Since \( G' \) is alignable, we know that \( \alpha_1 - \alpha_2 = \beta_1 - \beta_2 \). If \( \rho(v'_{i_1}) = v_i \) then \( \alpha_1 - \alpha_2 \in
\{0, \pm 3^j\} \) and likewise if \( \rho(v'_{i_2}) = v_j \) then \( \beta_1 - \beta_2 \in \{0, \pm 3^j\} \). By construction, \( i \neq j \) hence
\( \alpha_1 - \alpha_2 = \beta_1 - \beta_2 = 0 \) and so \( \alpha_1 = \alpha_2 \) and \( \beta_1 = \beta_2 \).

Let \( \rho(w'_1) = f_{i_1}a_2a_3 \) and \( \rho(w'_1) = f_{j_1}b_2b_3 \) where each \( a_k \) and \( b_k \) is a bit. Since the
weights on the edges, together with the “sign” of the variable in the factor \( F_i \) of \( B \), suffice
to determine the values of the bits in the labels of the literal nodes, we can conclude that
\( \alpha_1a_2 = b_1b_2 \). This implies that there can be at most two nodes with two outgoing edges
whose outgoing edges go to the same variables. In other words, if \( w'_3 \) is a node in \( G' \) with
two outgoing edge to \( v'_{i_1} \) and \( v'_{i_2} \), then \( w'_3 = w'_1 \) or \( w'_3 = w'_2 \).

Since there are at least five nodes with two outgoing edges, there is at least one node
with an edge to each pair of variables in in Figure 19. Let \( \sigma \) be an alignment of \( G' \) such that
\( \sigma(w'_{ij}) = 0 \), then \( \sigma(v'_i) \in \{3^i, 2 \cdot 3^i\} \) and \( \sigma(v'_j) \in \{3^j, 2 \cdot 3^j\} \). If \( \alpha_j = \beta_j \), then \( \sigma(w_{jk}) = 0 \)
and so $\sigma(v_k) \in \{3^k, 2 \cdot 3^k\}$. Thus, for each "variable" node $v_0$, $\sigma(v_0) \in \{3^a, 2 \cdot 3^a\}$ and, by the argument above, $G'$ has twenty-one nodes and hence is not minimal. Therefore $\alpha_i \neq \gamma_j$ and by a similar argument $\alpha_i \neq \beta_j$.

Since $\alpha_j, \beta_j \in \{3^j, 2 \cdot 3^j\}$, $\sigma(w_{jk}) = \pm 3^j$. Likewise, $\sigma(w_{ik}) = \pm 3^i$. Since $\sigma$ is an alignment, we have:

$$\beta_k - \gamma_k = \pm 3^i \pm 3^j$$

but the left hand side has to be 0 or $\pm 3^k$, a contradiction.

Therefore, if $G'$ has only nineteen nodes, than for at least one of the "literal" nodes $f_{i,k}$, we have $\|\rho^{-1}(f_{i,k})\| = 1$.

End of Proof.

**Lemma 3.5** If $(G', \rho)$ is an alignable path cover of $G_B$ with $m + \frac{3}{2}m(m-1) + 16n$ nodes, than there exists an alignment of $G'$ such that each "variable" node $v_i$, $\sigma(\rho^{-1}(v_i)) \in \{3^i, 2 \cdot 3^i\}$.

**Proof:** Let $\sigma$ be an alignment of $G'$. Let $v'_i = \rho^{-1}(v_i)$ for each "variable" node $v_i$. An alignable path cover of $G_B$ has $m + \frac{3}{2}m(m-1) + 16n$ only if $||\rho^{-1}(w_{ij})|| = 3$ for each $w_{i,j}$, hence there is a node $w'_{ij} \in \rho^{-1}(w_{ij})$ with outgoing edges to $v'_i$ and $v'_j$. Define $\sigma' = \sigma - \sigma(w'_{12})$ so that $\sigma'(w'_{12}) = 0$.

Thus, for each $i \neq 1, 2$, there is a subgraph of $G'$ as shown in Figure 19. Further, since $G'$ is alignable, we know that:

$$(\alpha_1 - \beta_1) + (\beta_i - \gamma_i) + (\gamma_2 - \alpha_2) = 0.$$
By construction we also know that:

\((\alpha_1 - \beta_1) \in \{-3,0,3\}\)
\((\beta_i - \gamma_i) \in \{-3^i,0,3^i\}\)
\((\gamma_2 - \alpha_2) \in \{-9,0,9\}\)

and the only solutions to these equations forces \(\alpha_1 = \beta_1\), \(\beta_i = \gamma_i\), and \(\gamma_2 = \alpha_2\). Thus \(\sigma'(w_{i1}) = 0\) for each \(i \neq 1\) and \(\sigma'(v_i') \in \{3^i,2 \cdot 3^i\}\) since \(G'\) is alignable and there is an edge from \(w_{i1}\) to \(v_i'\) with weight either \(3^i\) or \(2 \cdot 3^i\).

End of Proof.

**Proposition 3.16** If \(G_B\) is \((m + \frac{1}{3}m(m-1) + 16n)\)-alignable, then \(B\) is satisfiable.

**Proof:** By the above lemma, there is an alignment \(\sigma\) such that \(\sigma(\rho^{-1}(v_i)) \in \{3^i,2 \cdot 3^i\}\) for each variable node \(v_i\). Define a truth assignment \(f:\{v_1,\ldots,v_m\} \rightarrow \{\text{TRUE, FALSE}\}\) by the following rule:

\[
\begin{cases} 
\text{FALSE} & \text{if } \sigma(\rho^{-1}(v_i)) = 3^i \\
\text{TRUE} & \text{if } \sigma(\rho^{-1}(v_i)) = 2 \cdot 3^i 
\end{cases}
\]

Now we must show that this truth assignment satisfies each factor in \(B\). Let \(F_i\) be any factor in \(B\). By Lemma 5.4., there is one node, \(f_{i,k}\), for which \(\|\rho^{-1}(f_{i,k})\| = 1\). Let \(f_{i,k}' = \rho^{-1}(f_{i,k})\) then there is a subgraph of \(G'\) like the one shown in Figure 20. Since \(G'\) is alignable, we have:

\[
\sigma(v_i') - \alpha_i = \sigma(v_j') - \alpha_j
\]
but the left hand side is in the set \{-3^i, 0, 3^i\} and the right hand side is in the set \{-3^i, 0, 3^j\}. Thus \(\sigma(v'_i) = \alpha\) and so \(\sigma(f_{i,k}) = 0\).

By construction, the node \(f_{i,k}\) corresponds to a combination of values of literals which would satisfy \(F_i\). In particular, at least one of the literals in \(F_i\) is true. That literal is marked with a 1 in the label \(k\) of \(f_{i,k}\). Let \(v_j\) be the variable in that literal and \(\alpha\) the weight of the edge from node \(f_{i,k}\) to the node labeled \(v_j\). If \(v_j\) appears negated in \(F_i\), then \(\alpha = 3^j\) and \(\sigma(f_{i,k}) = 0\). As a result, \(\sigma(v_j) = 3^j\) and the truth assignment \(f\) maps \(v_j\) to FALSE and \(F_i\) is satisfied. On the other hand, if \(v_j\) appears unnegated in \(F_i\), then \(\alpha = 2 \cdot 3^j\) and so \(\sigma(v_j) = 2 \cdot 3^j\), therefore \(f\) maps \(v_j\) to TRUE and again \(F_i\) is satisfied.

I conclude that each factor \(F_i\) of \(B\) is satisfied by \(f\) and therefore \(B\) is satisfiable.

*End of Proof.*

**Theorem 3.5** 3-CNF Satisfiability is polynomially reducible to \(k\)-alignability. Therefore \(k\)-alignability for graphs in \(S\) is NP-complete.

*Proof:* The construction of the graph \(G_B\) from expression \(B\) given above takes time proportional to the size of \(G_B\) which has \(m + m(m - 1)/2 + 7n\) nodes and \(2m(m - 1) + 21n\) edges where \(m\) is the number of variables in \(B\) and \(n\) is the number of factors. Both \(n\) and \(m\) are bounded by \(|B|\), hence the reduction is polynomial. The two previous propositions show that the two problems are equivalent, and since the graphs constructed are in the class \(S\), Propositions 3.14, 3.15 and 3.16 show that \(k\)-alignability for graphs in \(S\) is NP-complete.

*End of Proof.*

3.5 Using Code Replication

Code replication provides an alternative to inserting a barrier synchronization point. The choice between barrier synchronization and code replication depends on the relative cost of a barrier synchronization point compared with the time to execute the code that would be added by replication. Given this piece of information however, the algorithm of Chapter 2 could be parameterized to decide which will be faster.

The choice of where to put barrier synchronization points is not trivial. Consider the loop:
DO I = 1,N
   A(I) = f()
   B(I) = A(I) + A(I-1) + g()
   C(I) = B(I) + B(I-1)
ENDDO

Assume that all three statements are individual strongly connected region of the dependence graph and that each is parallel. The terms f() and g() represent computations of arbitrary complexity. Note that there are alignment conflicts between both the first two statements and the last two statements. There are four alternatives for parallelizing this loop. The first puts all three statements in one parallel loop, replicating the first statement twice and replicating the second statement once. The second alternative makes three separate loops using two barrier synchronization points. The third alternative uses one barrier synchronization point, putting the first two statements together and replicating the first statement once. The last alternative puts the last two statements together and replicates the second statement once but not the first statement. The choice between these alternatives depends on the cost of a barrier synchronization point and the execution cost of f() and g().

Another aspect of choosing between barrier synchronization points and code replication is that other factors, such as serial regions, may force barrier synchronization points. Consider the loop:

DO I = 1,N
   S = S + A(I)
   B(I) = f()
   A(I) = B(I-1) + B(I)
ENDDO

The anti-dependence from the use of A to its definition forces a barrier synchronization point between the serial first statement and the parallel third statement. Since the barrier synchronization point is required anyway, there seems little reason to replicate the second statement to resolve the alignment conflict.

The difficulty of selecting from these alternatives and the underlying difficulty in deciding how to minimize replication suggest that heuristics be used. Since breaking

\footnote{This assumes that the overhead to start and stop a parallel loop is small compared to a replication of f(). While this assumption certainly will not hold in general, it may be realistic for machines tailored to fast parallel loop startup, such as the Ultracomputer and the Alliant FX/8.}
an alignment conflict replicates ancestor nodes, we want to build partitions beginning
with nodes at the "bottom" of the DAG. To do this we simply run the greedy algorithm
using a reverse topological order and convert check_sons into check_parents by looking at
immediate predecessors rather immediate ancestors. The set nopreds is renamed nosuccs
to indicate this change in direction of the algorithm. Beginning with this version of the
algorithm of Chapter 2 with loop alignment, we add replication as follows: Instead of
arbitrarily selecting a node from nosuccs — NotOk, the inner loop selects the "largest"
node (in terms of estimated execution time) from the set and attempts to add it to the
current partition. This choice gives a natural preference to replicating smaller nodes. The
method of combining connected components and using relative alignments developed to
incorporate alignment into the parallel code generator ensures that a node will only be
replicated if there is in fact an alignment conflict (as opposed to making blind choices for
sink nodes).

The set of nodes found to cause alignment conflicts is maintained, and after all nodes
are added to the current partition as can be without replication, we begin replicating nodes
in this set (and their ancestors) that are marked as priority nodes in the sense of section
2.6: nodes that can not be fused with an immediate ancestor\(^5\). As this proceeds, new
priority nodes may become ready for inclusion in the current partition and are replicated
next.

When all priority nodes and their priority ancestors have been replicated, or the total
cost exceeds the cost of a barrier synchronization point, we stop. In the first case, we
proceed to replicate the nodes, and in the latter we give up and start a new partition
(insert a barrier synchronization point). Replication is not attempted on non-priority
nodes\(^6\).

Of the alternatives listed for the example above, the one that cannot be selected by
this heuristic is the last one, where the second statement is replicated but the first is not.
This is a reflection of the short-sightedness of the definition of priority. A more complicated
definition would yield better average results. One alternative is to run the simple greedy

\(^5\) Note that if all remaining nodes could be added to the current partition via replication,
then all nodes are priority nodes

\(^6\) If parallel loop startup time is costly, then we can use it rather than barrier syn-
chronization cost to compare against the cost of replicating non-priority nodes. This
represents a secondary parameter to the algorithm
algorithm using forward topological sort rather than a reverse topological sort (as originally presented in Chapter 2). This will generate a partition that schedules loops as soon as possible. The relative position of a node in that partition would estimate the number of barrier synchronization points that must precede it. This pre-pass would also provide an upper bound on the number of partitions. The “reverse” greedy algorithm can then give a preference to nodes that would insure that the partition being built is smaller than this upper bound by giving a preference to nodes that the “forward” algorithm put farthest from the beginning of the partition.

3.6 Transformation Preventing Dependences

Two forms of dependences can prevent effective use of loop alignment and code replication. One of these, alignment preventing dependences, prevents alignment because the dependence becomes invalid through alignment. An example is:

```
DO 10 I = 1, N
    DO 10 J = 1, N
    A(I,J) = ...
    ... = ... A(I-1,J+1)...
10 CONTINUE
```

The true dependence generated by A has direction vector \((1, -1)\). After alignment of the outer loop, the dependence will have a direction vector of \((0, -1)\), which is invalid. This dependence is also interchange-preventing[Alle 83] since interchanging the I and J loops changes the distance vector to \((-1, 1)\) which is again invalid. Finally, this dependence prevents fusion after unrolling the outer loop since that would also result in a distance vector of \((0, -1)\) [CaKe 86].

The common theme to the three transformations mentioned above is that they affect the distance vector. Loop alignment and unroll-and-fuse [CaKe 86] directly alter the component of the distance vector corresponding to the loop being transformed. Loop interchange swaps components in the distance vector corresponding to the loops being interchanged. The set of dependences that inhibit unroll-and-fuse are the same as the alignment preventing dependences. As long as loop interchange is applied only to adjacent
procedure visit_node(v)
    visited(v) ← true
    σ(v) = 0
    for each edges e = (v, w) do
        if not visited(w)
            then visit_node(w)
        if d_k^e < 0
            then σ(v) = min{σ(v), σ(w) - d_k^e}

Figure 21. Algorithm to eliminate alignment preventing dependences at level k.

loops, the set of interchange preventing dependences is a proper subset of the set of alignment preventing dependences.

Loop alignment is a tool which can alter components of distance vectors. In particular, loop alignment can be used to change components of a distance vector other than the one corresponding to the level of the dependence. Figure 21 shows an algorithm that defines a function σ that can be used to eliminate alignment preventing consistent dependences. It is a simple depth-first search. The expression d_k^e represents the k^{th} component of the distance vector associated with edge e. After a node v has been visited, σ satisfies the following invariant: for all dependences e = (v, w), σ(v) ≤ σ(w) - d_k^e. Hence, if we use σ in for an alignment of the k^{th} loop, then, by Theorem 2.1, we have:

\[ d_k^{e} = d_k^e - σ(w) + σ(v) ≥ 0 \]

Thus loop alignment can be used to convert dependences that inhibit reordering transformations into dependences that do not inhibit those transformations. We can also extend this technique to remove consistent fusion preventing dependences. Consider the following two loops:
DO I = 1,N  
   A(I) = ...  
ENDDO  
DO I = 1,N  
   ... = ... A(I+1)...  
ENDDO  

These two references to A generate a loop independent true dependence between the statements that contain them. If these loops were fused, the references would generate a loop carried backward dependences with distance vector (1). Such a dependence is called fusion-preventing\(^7\).

The definition of a distance vector can be extended to include components corresponding the the loops that are not common to the references\(^8\). These extra components are computed as if the non-common loops were made common by loop fusion. The definition of valid dependence changes since the first non-zero term in the direction vector will be negative for a fusion-preventing dependences. Likewise, a dependence is loop carried only if a component of distance vector associated with a common loop is non-zero; otherwise it is loop independent. The fusion-level of a loop independent dependence is the index of the first non-zero component of the distance vector (if one exists) and \(+\infty\) otherwise. The fusion-threshold is the value of the component of the distance vector corresponding to the fusion-level.

The next loop nests illustrate that fusion-level corresponds to the outermost level at which the fusion-preventing dependence inhibits fusion of the loop nests.

DO 10 I = 1, N  
   DO 10 J = 1,M  
      A(I,J) = ...  
10 CONTINUE  
DO 20 I = 1, N  
   DO 20 J = 1,M  
      ... = ...A(I+1,J)...A(I,J+1)...  
20 CONTINUE

There are two loop independent dependences between the assignment statements: one has fusion-level 1 and fusion-threshold \(-1\), the other has fusion-level 2 and fusion-threshold

---

\(^7\) See Section 2.7  

\(^8\) Wolfe defines augmented direction vectors in a similar way when discussing loop fusion [Wolf 82]. Warren also gives a test for fusion-preventing dependences and an algorithm for loop fusion [Warr 84].
also -1. The first is fusion-preventing at the outer level. After loop interchange is applied to both loops, the second dependence would be fusion-preventing at the outer level.

Loop alignment can be used with these extended dependence vectors to remove fusion-preventing dependences in the same way as it was used to remove alignment preventing dependences. Consider the following loops:

\[
\begin{align*}
\text{DO } & \text{I} = 1,N \\
& \quad \text{A(I)} = \\
\text{ENDDO} \\
\text{DO } & \text{I} = 1,N \\
& \quad \ldots = \text{A(I+1)} \\\n\text{ENDDO}
\end{align*}
\]

If we align the first loop by 1, we can then safely fuse to get the loop:

\[
\begin{align*}
\text{DO } & \text{I} = 0,N \\
& \quad \text{IF (LGT.1) A(I+1)} = \\
& \quad \text{IF (LLE.N) } \ldots = \text{A(I+1)} \\\n\text{ENDDO}
\end{align*}
\]

3.7 Summary

Loop alignment is presented as a procedure for altering loop carried dependences so that they are carried at a deeper nesting level. This reduces the number of barrier synchronization points needed to synchronize the parallel program built by the parallel code generator of Chapter 2. The resulting algorithm maintains connected components of the dependence graph of the current partition as disjoint sets and each node has a relative alignment. This allows quick detection of alignment conflicts and computation of an exact alignment. When loop alignment cannot be applied due to alignment conflicts, code replication can be used to provide each iteration with a separate copy of the needed data. It is shown that sufficient code replication to generate a single parallel loop may result in exponential increase in the size of the loop, and finding a minimal amount of replication is shown to be NP-hard even when the growth is limited to twice the size of the loop. The trade-off between code replication and the use of barrier synchronization points is discussed and modifications to the parallel code generator are described based on the heuristic developed for loop interchange.

Finally, loop alignment is discussed as a general tool to adjust distance vectors associated with data dependences. It can be used to improve the effectiveness of other reordering transformations such as loop interchange, loop fusion and unroll-and-fuse.
Chapter 4:
Interprocedural Data Dependence Analysis

4.1 Introduction

A significant problem for the parallel code generator presented in Chapter 2 is lack of information about the side-effects of calls to external functions. When individual subroutines are submitted independent of any program context, very conservative estimates must be made about the effects of any external routines that subroutine invokes. In particular, it must be assumed that every actual parameter at a call site and global variable\(^1\) is both used (values read from) and modified by the external routine invoked at the call site. These assumptions not only prevent parallelization of the loops that contain the call site, but also generate recurrences that prevent distribution of the loops to isolate the call sites.

Section 1.3.3 discusses three techniques to improve the accuracy of dependence analysis in the presence of call to external routines. In the PFC transformation system developed at Rice University [AlKe 82], a more or less straightforward implementation of interprocedural summary information [AlCK 86] is in place. For each subroutine in the input program, two sets, USE and MOD, are computed. Theses contain the names of formal parameters and global variables that may be either used or modified respectively when the subroutine is invoked.

Our experience indicates that the granularity of interprocedural summary information (as described in [Bann 78], [Bart 77] and [Coop 83]) is too coarse to allow effective detection of parallelism in loops containing call sites. The problem is that the current analysis treats arrays as units. It is able to determine whether an array is modified somewhere, but not whether it is modified in only a single column or row. This limitation is disastrous for parallelization because the most effective way to parallelize a loop is through data decomposition, in which each parallel iteration works on a different subsection of a given array.

Some mechanism for determining the subsections actually affected by interprocedural side effects is needed. Triolet [TrIF 86] has proposed a procedure that would determine

\(^1\) In Fortran, this is every variable in a common block
the convex hull of the set of array locations affected as a side effect of a procedure call. Unfortunately, this method is too expensive to use in a compiler [TrIF 86]. Therefore, we seek to achieve a more limited goal: to recognize some important special cases of array side effects. For example, it would be extremely useful if we were simply able to recognize when the modification of an array by a procedure call is limited to a single column or row of the array.

While the current Fortran standard prohibits dynamic recursion, it is probable that the next standard will allow it. Most other high level languages support recursion. Therefore, it is important that an approach to summary information be able to handle recursive programs. The most successful algorithms for solving data flow problems have posed the summary problem as a least fixed point solution to a set of simultaneous equations over a bounded lattice [KaUl 77].

Fortunately, a generalization of the methods currently used to solve interprocedural data flow analysis problems can be used to develop more precise information about side effects. Although it may not be immediately clear from perusing the papers, the published algorithms for interprocedural analysis (e.g. Cooper and Kennedy [CoKe 84]), can be extended to work on lattices. Consider the example lattice of reference patterns to the array A show in Figure 22. Note that I, J, and K are arbitrary symbolic input parameters to the call. Bottom, \( \bot \), in a particular subscript position indicates that any value may be used in that position. The top row summarizes access to individual elements, the second row summarizes access to columns or rows of the array, and the bottom element indicates access to the entire array.

The interprocedural analysis algorithms will work correctly, although more slowly, on vectors of lattice elements (instead of bit vectors) if the lattice has the finite chain property – that is, there exist no descending chains of unbounded length. The (worst case) running times of the algorithms for such lattices is directly proportional to maximum lattice depth and time to execute a single meet operation.

Hence, it is easy to see that we can perform interprocedural analysis for sections of arrays. We will call any section that can be precisely described in the chosen lattice a regular section. The only remaining problem is how to choose a definition of regular section with the right blend of precision and efficiency. This choice is the subject of the next section.
4.2 Summarizing Array Accesses

The basic information used by the 'standard' data dependence tests consists of the expressions appearing in the subscripts of the arrays plus the upper bounds of the loops that contain these expressions. Normalization is done so that (as much as possible) the expressions that appear in the subscripts consist only of constants, loop control variables and loop invariant expressions and so that the loops iterate from 1 to some upper bound in increments of 1.

This same information can be used to describe the portions of an array that can be addressed by a particular subscripted reference to that array. In particular, for the reference:

\[ A(a_1 + b_1 \cdot i_1, a_2 + b_2 \cdot i_2) \]

where each \( a_j \) and \( b_j \) is loop invariant in the outermost loop and the variables \( i_j \) are the control variables of the loops that enclose the reference. When the loop nest is not triangular, i.e., all loop upper bounds are invariant in the outermost loop and all of the \( i_j \) are distinct, the region of \( A \) that is referenced by the above can be expressed in triplet notation as:

\[ A(a_1 + b_1 : a_1 + b_1 \cdot N_1 : b_1, a_2 + b_2 : a_2 + b_2 \cdot N_2 : b_2) \]
This reference abstracts away the original loops and could be used to determine if memory overlaps occur between different loop nests. This would be done by checking for intersection in the sets of indices for each subscript position in the pairs of references being examined. For example, the references:

\[ A(1:100:2,1:50) \]
\[ A(2:101:2,1:50) \]

can be seen to be disjoint since there is no common subscript value in the first subscript position. Thus, under fairly strong restrictions, this representation can be used to summarize the portion of an array accessed by a single reference to that array inside a loop nest.

In addition to comparing disjoint loop nests, the above representation can be used when testing for dependences between array references that are not in the same loop nest but have common loops around them by summarizing those loop levels not in common. For instance, the above array references could be generated by the loops:

\[
\begin{align*}
\text{DO } I & = 1,100 \\
\text{DO } J & = 1,50 \\
A(2*J-1,I) & = \\
\text{ENDDO} \\
\text{DO } J & = 1,50 \\
A(2*J,I) & = \\
\text{ENDDO} \\
\text{ENDDO}
\end{align*}
\]

While standard dependence analysis could handle this loop, detecting that no dependence exists between the two references to \( A \), it would not be able to if the inner loops were hidden in subroutines, as in:

\[
\begin{align*}
\text{DO } I & = 1,100 \\
\text{CALL ODD}(A,1,50,I,2) \\
\text{CALL EVEN}(A,2,50,I,2) \\
\text{ENDDO}
\end{align*}
\]

If interprocedural analysis can provide a summary of the usage of \( A \) like the one given above, then dependence analysis can determine that the \( I \) loop can be executed in parallel.

The above format for summary information is too restrictive. Many important access patterns can not be correctly expressed in triplet notation. Two examples are triangular access, and diagonal access, illustrated as:
DO I = 1,100
   DO J = 1,I-1
       A(I,J) = 0
   ENDDO
   A(I,I) = 1
ENDDO

The first reference to $A$ accesses the lower triangular portion of $A$ and the second accesses the main diagonal. Neither of these accesses can be expressed with simple triplet notation.

An alternative to triplet notation is leaving the subscripts of the arrays as symbolic expressions and including information about how the symbolic parameters vary. In particular, any variable that is inductive with respect to some loop enclosing the reference is replaced with an expression in terms of the loop control variable of that loop (induction variable substitution) and the upper bound of that loop is included in the summary information. This is sufficient to represent diagonal regions. The second reference in the above loop can be summarized as:

$$\langle A(i_1,i_1); 1 \leq i_1 \leq 100 \rangle$$

where the second component indicates that $i_1$ assumes all values from 1 to 100 in increments of 1 in any order. Since the reference is enclosed in one loop, there is one loop control variable appearing in the subscripts and one upper bound. When a reference is enclosed in $k$ loops, there will be $k$ induction variables $i_1, \ldots, i_k$, and $k$ upper bound expressions, $N_1, \ldots, N_k$, used in the constraints:

$$1 \leq i_1 \leq N_1, \ 1 \leq i_2 \leq N_2, \ 1 \leq i_k \leq N_k.$$  

In order to correctly express triangular regions, it is necessary to also allow upper bounds to include references to induction variables of outer loops. The first reference to $A$ above is summarized as:

$$\langle A(i_2,i_1); 1 \leq i_1 \leq i_2 - 1, 1 \leq i_2 \leq 100 \rangle.$$  

If a subscript varies in a way that is not affine with respect to the loop control variables (such as an indirect reference through an index array), dependence analysis will not, in general, be able to determine that dependences do not exist from that subscript. Therefore, we use special variables not occurring in user programs which are not constrained by loop bounds. For example, the reference to $A$ in the loop:

---

\[2\] We assume here that all loops are normalized to iterate from 1 to an upper bound in steps of 1.
DO I = 1,100
DO J = 1,50
A(IPIVOT(J),I) =
ENDDO
ENDDO

will be summarized as \(A(s,i_1); 1 \leq i_1 \leq 100\) where \(s\) is a never used in a program and indicates that we have no information about the values of \(IPIVOT\).

The general form for array access summary information is:

\[
\left< A(\theta_1(\vec{i}), \ldots, \theta_n(\vec{i})); 1 \leq i_1 \leq \phi(\vec{i}), \ldots, 1 \leq i_k \leq \phi_k(\vec{i}) \right>
\]

where each \(\theta_j\) is an affine function of the loop control variables \(\vec{i} = (i_1, \ldots, i_k)\). Similarly for the functions \(\phi_j\). By affine, I mean that for each \(j\), there exists loop invariant expressions \(a_0^j, \ldots, a_k^j\) such that:

\[
\theta_j(\vec{i}) = a_0 + \sum_{i=1}^{k} a_i^j \cdot i_j
\]

and likewise for each \(\phi_j\). If analysis determines that any subscript position or loop upper bound varies in a way that does not fit this form, that subscript or upper bound is assumed to vary in an unknown way and is represented by a non-program variable.

When a loop contains multiple references to the same array, these references may or may not access the same portions of the array. The previous example illustrates the latter case, since the main diagonal is disjoint from the lower triangular region. It is undesirable to keep independent information about each occurrence of a particular array variable because this will lead to large consumption of space for storage of this information and of time spent doing dependence testing for every occurrence. In the interprocedural case, the summary information may be different for every occurrence of a variable along every execution path in call graph! Thus the amount of summary information can grow linearly with the size of the program but exponentially with the depth of the (acyclic) call graph. It is therefore desirable to summarize a loop nest or subroutine body with only one reference to each variable used and one reference to each variable modified.

We need a technique to merge the summary information for two occurrences of a variable into a single item of summary information. Unfortunately, this will lead to loss of precision in the summary information. For instance, the references to \(A\) in:
DO I = 1,100
  DO J = I,100
    A(I,J) = 0
    A(J,J) = 0
  ENDDO
ENDDO

would be individually summarized as:

\[ \langle A(i_2, i_1); 1 \leq i_1 \leq i_2 - 1, 1 \leq i_2 \leq 100 \rangle \]
\[ \langle A(i_1, i_2); 1 \leq i_1 \leq i_2 - 1, 1 \leq i_2 \leq 100 \rangle \]

but to safely represent both of these regions using the above representation we must drop the information that the main diagonal is not modified and get the following summary information:

\[ \langle A(i_1, i_2); 1 \leq i_1 \leq 100, 1 \leq i_2 \leq 100 \rangle. \]

We could trade space for precision by not merging all of the occurrences for some of the variables.

Once we have a method for summarizing the memory accesses made directly by loops, we need to extend this to include the effects of call sites to external routines. This extension has two parts: how to propagate information across a single subroutine call and how to build summary information for each subroutine about what access may take place during the entire time the routine is active.

Propagating information from the context of the called routine into the context of the caller involves two steps: translation of the coefficients in the summary information from formal parameters to actual parameters, and mapping of subscripts of formal array parameters to subscripts of the actual arrays bound at the call site. For example, suppose we have the call site:

```plaintext
DO I = 1,N
  CALL SUB(A(1),I-1)
  CALL SUB(A(I),N-I)
ENDDO
```

and the formal parameters of the subroutine are declared:
SUBROUTINE SUB(V,N)
REAL V(N)

and we have determined that all of V is accessed, so the subroutine summary of V is:

\( \langle V(i); 1 \leq i \leq N \rangle \)

We can translate this into summaries for the two call sites as:

\( \langle A(i_1); 1 \leq i_1 \leq I - 1 \rangle \)

\( \langle A(i_1 + I - 1); 1 \leq i_1 \leq N - I \rangle \).

Standard data dependence tests can now determine that these subroutines calls can be executed concurrently.

The propagation of information across a call site is complicated in Fortran since the language allows an array formal parameter to be shaped differently from the actual parameter it is bound to, and the formal array parameter may overlay any contiguous portion of the actual parameter. The implication of this is that, even though an occurrence of the formal parameter may have subscripts that are affine in loop control variables, it may not be possible to translate those subscripts into subscripts for the actual parameter that are also affine. For example, in the following fragment, the formal parameter has fewer dimensions then the actual parameter but accesses every element of the actual parameter:

CALL SUB(A(1,1),N)

where the subroutine SUB is:

SUBROUTINE SUB(V,N)
REAL V(N)
DO I = 1,N
   V(I) =
ENDDO
END

The summary for the subroutine is the same as in the previous example but we can only translate through the call site into the summary (assuming column major storage):

\( \langle A(i_1,1); 1 \leq i_1 \leq N \rangle \)
if we know that $N$ is less than or equal to the bound of the first dimension of $A$. The reason for this is that the above summary information indicates that no elements in the second column of $A$ are accessed by the subroutine, but if $N$ is greater than the length of the columns of $A$, then some elements of the second column will be accessed. To handle this case, we need to be able to linearize references to the array $A$. So if $A$ is dimensioned $A(M,*)$ and we do not know if $N \leq M$, then we can still represent the summary information as:

$$\langle A(i_1); 1 \leq i_1 \leq N \rangle$$

and then take other summaries of $A$, such as:

$$\langle A(i_1,i_2); 1 \leq i_1 \leq L, 1 \leq i_2 \leq L \rangle$$

and linearize them to:

$$\langle A(i_1 + M \cdot i_2 - M); 1 \leq i_1 \leq L, 1 \leq i_2 \leq L \rangle$$

for dependence testing. We do not want to linearize unless we have to since we lose information when it is done.

Once information has been translated from the context of the called routine into the context of the calling routine, it can be treated like any other occurrence in the calling routine. In particular, it contributes to the summary information associated with the loop that contains it, and summary information of the calling routine. If the call graph of a program is acyclic, we can compute summary information in a single bottom-up pass over this tree. When we compute summary information for each statement, we will already have the summary information for any external routine that statement may invoke. Also, the summary information for the entire subroutine can be computed since the summary information for all the statements and loops is completely known.

In the case that the call graph is cyclic, we have a more general data flow problem. To use the extensive work already done in solving data flow problems, we need to construct data flow equations that describe the summary information associated with a routine in terms of the variables directly modified by that routine plus the effects of any routines called by it. These equations will be fairly complicated since they must represent translating formal parameter information to local information, encapsulating the effects of any
loops that surround a call site, and merging to reduce multiple occurrences of a variable to a single summary.

The following sections develop the ideas introduced in this section. Section 4.3 describes the most general form summary information kept for arrays. Section 4.4 outlines how two items of summary information are merged into one. Section 4.5 describes how summary information associated with subroutines is translated into local information at call sites. Section 4.6 describes a data flow framework for solving the summary problem for cyclic call graphs. Section 4.7 develops a lattice using a more restricted form of summary information. Section 4.8 discusses some difficulties in using the restricted lattice.

4.3 Regular Sections

This section presents the most general definition of regular section. It contains all information relevant to the standard dependence tests (separability, GCD, and Banerjee's Inequality).

4.3.1 Regular Sets and Regular Section Descriptors

A regular set is a set of integers that is the range of some affine function whose domain is a triangular iteration space. A triangular iteration space is a subset of $\mathbb{Z}^n$ ($n$-vectors of integers) defined by a set of affine inequalities and equal to:

$$I = \{(i_1, \ldots, i_n) \mid \forall j : 1 \leq j \leq n, \quad b^j + a_{j+1}^ji_{j+1} + \cdots + a_{n}^ii_n \geq i_j \geq 1\}.$$ 

where each $a_{k}^j$ and each $b^j$ is a rational value. Note that the range of each component is constrained only by the components to its right. Such a space is called triangular since the matrix of coefficients ($a_{i}^j$) is triangular: $a_{i}^j = 0$ if $i < j$.

A regular set can be operationally defined as the memory locations of an array $A$ that could be written by a loop nest of the form:
DO 10 \textit{I}_K = 1, N_K \\
DO 10 \textit{I}_{K-1} = 1, N_{K-1} \\
\ldots \\
DO 10 \textit{I}_1 = 1, N_1 \\
A(\theta_1, \ldots, \theta_n) = \\
10 \textbf{CONTINUE} \\

where each $\theta_j$ is an affine expression of the form:

$$\theta_j = a_{0j}^j + a_{1j}^j I_1 + \cdots + a_{kj}^j I_k$$

where each $a_{ij}^j$ is loop invariant in the outermost loop and each loop upper bound $N_j$ is affine in loops that contain it:

$$N_j = n_{1j}^j + n_{2j+1}^j I_{j+1} + \cdots + a_{kj}^j I_k.$$ 

and $A$ is an $n$-dimensional array with bounds $B_1, \ldots, B_n$.

The upper bound expressions define a triangular iteration space:

$$I = \left\{ \vec{i} = (i_1, \ldots, i_k) \mid \forall j : \ N_j[\vec{i}] \geq i_j \geq 1 \right\}$$

where $N_j[\vec{i}]$ is the value of $N_j$ after replacing each $I_j$ by $i_j$. The $\theta_j$ expressions define affine functions from the iteration space into memory by:

$$\Theta(\vec{i}) = \theta_1[\vec{i}] M_1 + \cdots + \theta_n[\vec{i}] M_n$$

where $\theta_j[\vec{i}]$ is the value of $\theta_j$ after replacing each $I_j$ with $i_j$ and the $M_j$ are the multipliers of $A$ defined by $M_1 = 1$ and $M_i = M_{i-1} B_{i-1}$ for $1 < i \leq n$. This definition is based on the assumption that $A$ will be stored into memory in column major order. Row major order will have different values for the multipliers.

A **regular section** of $A$ is a set of subscript vectors of $A$ that refer to a regular set of memory locations. The regular section of $A$ defined by $I$ and $\theta$ is elements of $A$ in the range of $\Theta$:

$$A^{-1}(\Theta(I)) = \{ \vec{b} \in B \mid \exists \vec{i} \in I : \Theta(\vec{i}) = b_1 M_1 + \cdots + b_n M_n \}$$

where $B = \Pi_k \{ 1..B_i \}$ is the set of valid subscripts of $A^3$.

\footnote{More general bounds, such as $A(-10,10)$, can also be used. Either subscripts can be normalized or the lower bounds can be maintained as part of the shaper information.}
Regular sections will be represented by terms called regular section descriptors (RSD) of the form:

\[ R = \langle A(B_1, \ldots, B_n); \theta_1, \ldots, \theta_m; N_1, \ldots, N_k \rangle \]

The first component, \( A(B_1, \ldots, B_n) \), contains the name of the variable and the bounds information associated with each subscript position. For variables that are part of a larger aggregates, such as a Fortran COMMON block or a Pascal record variable, the array name will be an ordered pair consisting of the name of the aggregate and the offset of variable from the beginning of that aggregate.\(^4\) The values \( B_i \) represent expressions that determine the bounds of the array. These expressions consist of integer constants, scalar parameters and scalar variables contained in COMMON blocks. The first component of a regular section of array \( A \) will be the same for all references to \( A \) in a single routine. If we assume that no variable name is used in more than one routine, the first component becomes program invariant.

Each \( \theta_j \) is an affine expression as described above; these expressions will be referred to as the subscript expressions of the RSD. The expressions that appear as coefficients consist of constants, formal parameters and global variables that do not vary in the routine containing the reference. This restriction allows RSD's from different loops to be compared freely. The variables \( I_1, \ldots, I_k \), representing the loop control variables of loops that enclose the reference are special names that do not appear in the input program. The shape of a regular section is the vector of array bounds \( (B_1, \ldots, B_{n-1}) \).

The expressions \( N_j \) will be called upper bound expressions: for each \( j \):

\[ 1 \leq I_j \leq N_j \]

The order of expressions is important; expressions are paired with variables in a left to right manner: variable \( I_j \) is associated with the \( j^{th} \) upper bound expression from the left. This order determines the iteration space \( I \) as defined above: \( N_j \) is 'outside' of \( N_k \) if \( j < k \).

\(^4\) This is sufficient for Fortran where all variables refer to contiguous storage. In a language like PL/I, where the elements of arrays can be non-contiguous, it is necessary to have more complicated data descriptors to retain the shape of the structures involved.
One implication is that the variable $I_j$ may only appear in upper bound expressions $N_k$ such that $k < j$.

The regular section of $A$ represented by a regular section descriptor $R$ will be denoted $A(R)$ and is defined as above in terms of $\Theta$ and $I$ where $\Theta$ is defined as:

$$\Theta(\vec{r}) = \theta_1[\vec{r}]M_{i_1} + \cdots + \theta_n[\vec{r}]M_{i_n}$$

For the purposes here, whenever information that appears explicitly in the above RSD can be inferred or is irrelevant, it will be omitted. In particular, shape information will not be shown. Thus the general term above will usually appear as:

$$\langle A; \theta_1, \ldots, \theta_m; N_1, \ldots, N_k \rangle.$$ 

Further, a variable and the upper bound associated with it will be jointly referred to as a loop. Thus the pair $I_1, N_1$ would be a loop. The level of a loop will refer to the subscript used on the control variable: the loop $I_j : N$ is the loop at level $j$. Also, any indexed set of expressions, such as $N_1, \ldots, N_m$ may be denoted $\vec{N}$, where it is understood that $N_i$ is the $i^{th}$ component of $\vec{N}$.

### 4.3.2 Construction of Regular Section Descriptors

To a large extent, RSD's can be determined from source programs by direct inspection: each reference to an array variable generates an RSD. Subscript expressions appear directly and upper bound expressions are taken from context. The shape information is available in the declaratory statements of the subroutine. One difficult point is determining which of the scalars appearing in expressions are routine invariant. This will, in general, require interprocedural summary information in addition to some global, intra-procedural information.

To get reasonable precision however, it is necessary to detect when scalars that vary inside the current routine vary directly with the loop control variable of a loop. The process of recognizing auxiliary induction variables and forward substituting them to points of use is called induction variable substitution. An algorithm is given in [AlKe 82] and a more precise version has been implemented based on definition-use chains in PFC. Further, the construction of return jump functions for use in interprocedural constant propagation
[CCKT 86] provides a method for finding auxiliary induction variables even when the modification of the scalar involved occurs inside a different subroutine.

Induction variables substitution will determine, for each loop and each scalar that is modified inside that loop, whether that scalar varies as an auxiliary induction variable and if so, a valid expression for that variable that consists of only the loop control variable and loop invariant expressions. By treating the entire subroutine as a loop with upper bound of 1, we can assume that all scalars are either variant or have inductive expressions in terms routine invariant expressions. Construction of RSD's at this point can be done by direct inspection of array references in the context of the loops that contain them.

4.3.3 Symbolic Terms

During construction of RSD's, variables that are formal parameters to the subroutine or have scope global to the current subroutine (e.g., in a COMMON block) may occur in either subscript expressions, upper bound expressions, or array bound expressions. Let $P$ be an infinite set of names that are used as scalar formal parameters and scalar global variables in programs. Elements of $P$ will be called program variables. Further, let $S$ be a set of names that are not used in programs; these names will be called non-program variables. Let $V = P \cup S$. A symbolic RSD is an RSD in which expressions contain elements of $V$ as well as integer constants. RSD's that are not symbolic are called constant RSD's.

We choose to interpret a symbolic RSD with symbolic terms as a function from $Z^V$ into the set of regular sections of the array that appears in the first component of the RSD. If $f: V \rightarrow Z$ is an element of $Z^V$, then the image of $f$ under a symbolic RSD $R$ is the constant RSD $R[f]$ obtained by replacing each occurrence of $v \in V$ that appears in $R$ with $f(v)$. 
4.3.4 Non-constant terms

During construction of RSD's, variables may appear in subscript and upper bound expressions that are neither constant over the entire subroutine nor auxiliary induction variables. In such a case, we must make the worst case assumption that they can assume any integer value. Such a variable will be said to be variant. An affine expression that contains a variant variable will also be assumed to achieve any integer value, and such an expression will also be called variant.

The interpretation of symbolic terms as function parameters provides the method used to represent variant expressions as symbolic terms. When an upper bound expression is variant, we must assume that any positive value can appear in the upper bound term. This fact can be encoded by selecting an element of $S$ that does not appear in the RSD and using it as the upper bound expression. The resulting symbolic RSD represents all possible values of the variable. Similarly, if a variant variable is used in a subscript position, no information is known about that subscript position and so a non-program variable is used in the corresponding subscript expression of the RSD being constructed.

4.3.5 Standardization

A problem with the above representation is that a given regular section will have many different regular section descriptors. This is one motivation for examining more restricted forms. Some transformations are listed below which preserve the regular section described by an RSD but alter it syntactically. The purpose of this section is to motivate the difficulty of working with this very general form summary information.

1. **Introduction of Scalar Loops.** A new loop can enclose an array reference, adding a new upper bound, but if the reference does not vary with respect to this loop, it does not affect the regular section described.

2. **Loop Reversal.** A loop can be reversed, running from $N$ to $1$ in steps of $-1$ without changing the access pattern. In fact, no reordering of the iterations will affect the access of data by the loop.

3. **Linearization.** An array reference could be linearized by making part of the address polynomial explicit. This does not occur in user programs but may be necessary to call sites where formal and actual parameters have different shapes.

4. **Loop Interchange.** The order of non-triangular loops can be arbitrarily rearranged. Even some triangular loops can be interchanged.

5. **Loop Sectioning.** A given loop can be split into two nested loops, the inner loop performing blocks of iterations of the original loop.

I will refer to a "standard form" without rigorously defining it. Intuitively, this standard form will involve as few loops and as little linearization as possible.
4.4 Merging Standard RSD's

This section describes a process whereby two RSD's are merged into a single RSD. The resulting RSD describes a regular section containing the union of the regular sections described by the two input RSD's. In general, this result will not equal the union of the input regular sections since the regular sections are not closed with respect to set union.

We begin by describing the merge process of the case where both RSD's have a single subscript expression. Let \( R_1 = \langle A; \theta_1; \bar{N} \rangle \) and \( R_2 = \langle A; \theta_2; \bar{M} \rangle \) where:

\[
\begin{align*}
\theta_1 &= a_0 + a_1i_1 + \ldots + a_ni_n \\
\theta_2 &= b_0 + b_1i_1 + \ldots + b_mi_m
\end{align*}
\]

The first step is to normalize the loops: if \( i_j \) is appears with a non-zero coefficient \( c \) in \( \theta_1 \) but does not in \( \theta_2 \), add \( ci_{m+1} - c \) and \( M_k = 1 \) to \( \theta_2 \), where \( k = ||\bar{M}|| + 1 \), and then interchange loops so that this new loop is at level \( j \). If \( a_j \) and \( b_j \) are the coefficients for the loop at level \( j \) with upper bound expression \( sN_j \) and \( M_j \) then let \( c_j \) be the greatest common denominator of \( a_j \) and \( b_j \). Replace \( a_ji_j \) and \( N_j \) in \( R_1 \) with \( c_ji_j + (a_j - c_j) \) and \( (a_j/c_j)N_j - (a_j/c_j - 1) \) respectively. If the greatest common divisor of two coefficients cannot be computed due to symbolic terms, it is taken to be 1.

The constant terms are handled somewhat differently. In \( R_1 \), \( a_0 \) is replaced with \( c_0 + c_ki_k \) where \( i_k \) is a new induction variable, \( c_0 = 2a_0 - b_0 \) and \( c_k = b_0 - a_0 \). The upper bound expression for the new level \( k \) loop is \( N_k = 1 \). In \( R_2 \), \( b_0 \) is also replaced with \( c_0 + c_ki_k \) but with \( N_k = 2 \). Note that \( a_0 \) and \( b_0 \) are the constant terms of the affine expression after coefficients have been normalized as in the previous paragraph.

The RSD \( R = \langle A; \theta; \bar{L} \rangle \) which is constructed by merging \( R_1 \) and \( R_2 \) is equal specified by:

\[
\theta = c_0 + c_1i_1 + \ldots + c_ki_k
\]

and for \( j < k \):

\[
L_j = \max\{N_j, M_j\}
\]

when the maximum can be expressed as an affine expression, otherwise an unused non-program variable is used for \( L_j \). It is assumed that any upper bound expression is greater than or equal to 1.

In the multi-subscript case, we first linearize the two RSD's so that each subscript position varies with the same set of loops and then apply the single subscript test above for each subscript independently.
4.5 Translation

In this section, techniques are described to include RSD summary information associated with one procedure into the RSD summary information associated with a routine that calls it. For example:

\[
\text{SUBROUTINE } A(V,N) \\
\text{REAL } V(N,N) \\
: \\
\text{DO } I = 1,N \\
\quad \text{CALL } B(V(I,I),N-I+1) \\
\text{ENDDO} \\
: \\
\text{SUBROUTINE } B(X,N) \\
\text{REAL } X(N)
\]

Assume that subroutine \( B \) modifies all of \( X \), \( R_B = (X(N); I_1; N) \). We want to build an RSD that is equivalent to \( R_B \) at the call site in \( A \), then include the effects of the loop enclosing the call site, and, finally, merge the new RSD into the existing summary information associated with subroutine \( A \).

The first step builds the RSD \( R_1 = (V; I_1 + I - 1; I; N - I) \), assuming column major storage. The DO loop binds the variable \( I \) to \( I_2 \) and so we get the RSD \( R_2 = (V; I_1 + I_2 - 1; I_2; N - I_2, N) \) which has standard form \( (V; N + 1 - I_1; I_2; N - I_2, N) \). This RSD can now be merged into the summary information for \( A \) using the techniques mentioned in the previous section.

The above process has two primary components. The first involves the translation of an RSD in the context of the called routine into an RSD at the call site in the context of the calling routine. The second component is the binding of symbolic variables to the affine expressions generated by loops around the call site.

Translation of upper bound expressions and array bound expressions is relatively straightforward. Global common block variables translate to themselves and formal parameters are replaced with their actual parameter bindings. The hard part is translating subscript expressions. The coefficients of the subscript expressions can translated the same way as upper bound expressions, but they must then be combined with the subscripts of the actual parameter.
All information that a subroutine has about the actual parameter must be explicitly passed as parameters. This invites abuse where the shapes of actual and formal parameters are wildly different. In cases where the shapes very are different, we have no choice but to linearize both the RSD of the called routine and the RSD associated with the actual parameter. When both the formal parameter and the actual parameter have only one subscript position, say $\theta_f$ and $\theta_a$, and the resulting subscript position is $\theta_f + \theta_a - 1$.

Frequently this linearization is unnecessary. In scientific code, two important cases arise. The first is that an entire array is passed as a parameter by an unsubscribed reference and the formal is a proper subsection of the actual parameter. This occurs when both formal and actual parameters have the same array shape (in the context of the calling routine), so that the name of the actual array can simply be substituted for the name of the formal array in the first component of the RSD. The second case occurs when the following definition is satisfied.

**Definition 4.1** A formal array parameter $f$ with bounds $F_1, \ldots, F_k$ is a proper section of an actual array parameter $a$ with bounds $A_1, \ldots, A_n$ and subscripts $\phi_1, \ldots, \phi_n$ if for all $j < k$:

\[
F_j = A_j \\
\phi_j = 1 \\
\phi_k + F_k \leq A_k
\]

The example above illustrates a proper section. The formal $V$ is contained in the first column of the actual $A$. If:

\[
R_f = \langle f(F_1, \ldots, F_k); \theta_1, \ldots, \theta_k; N_1, \ldots, N_m \rangle
\]

is the RSD associated with a formal parameter and:

\[
R_a = \langle a(A_1, \ldots, A_n); \phi_1, \ldots, \phi_n; \rangle
\]

is the RSD of the actual parameter at the call site, then when the formal is a proper section of the actual, we can translate the formal RSD into:

\[
\langle a(A_1, \ldots, A_n); \theta_1, \ldots, \theta_k + \phi_k, \phi_{k+1}, \ldots, \phi_n; N_1, \ldots, N_m \rangle
\]

and avoid the linearization step. The most significant bound of formal parameters is frequently abused and need not even be specified. Hence it is safer to examine the maximum
value obtained in the most significant subscript expression of the RSD itself than to simply
use the declared bounds information of the formal parameter to determine if the formal
parameter is used as a proper section.

In general, we can view a translation as defined by a 4-tuple:
\[
F^{A\rightarrow B}_c = \left\langle A; f_c; V \mapsto \Theta[V]; \vec{\theta}^c; \vec{N}_c \right\rangle
\]
where

- \( A \) is the name (and shape) of the actual array parameter; The \( A \mapsto B \) superscript
  indicates that formal parameter \( A \) is bound to actual parameter \( B \). At a given
  call site, a particular actual array name may be bound to more than one formal
  parameter name.
- \( f_c \) is a mapping from the set \( V \) of variables to \( \Theta[V] \), the collection symbolic affine
  expressions over the set of variables;
- \( \vec{\theta}^c \) is a vector of symbolic expressions corresponding to the actual subscripts
- \( \vec{N}_c \) are the upper bound expressions of the DO loops that enclose the call site.

The function \( f_c \) represents both the translation of formal parameters to actual parameter
and the auxiliary induction variables defined by loops around the call site. This 4-tuple
defines a function that maps RSD's in the context of the called routine into RSD's in the
context of the calling routine:
\[
F_c(\left\langle B; \vec{\theta}^b; \vec{N}_b \right\rangle) \mapsto \left\langle A; \vec{\theta}^b[f_c] + \vec{\theta}^c; \vec{N}_b[f_c], \vec{N}_c \right\rangle
\]
where \( \vec{\theta}^b[f_c] \) is the the vector of subscript expression obtained by replacing each variable,
\( v \), occurring in \( \vec{\theta}^b \) with \( f_c(v) \) for each \( i \); similarly for \( \vec{N}_b[f_c] \). Note that any loop control
variables that appear in \( \vec{\theta}^c \) are relabeled to be outer to loops in \( \vec{\theta}^b \). For brevity, information
about change of shape is omitted, hence the vector addition of \( \vec{\theta}^b \) and \( \vec{\theta}^c \) is not to be taken
literally. Not only may these two vectors be of different lengths, but the result may be
linearized to yet a third shape.

For the call site above, we have the following 4-tuple:
\[
F_c = \langle V; \{I \mapsto I_1, N \mapsto N - I_2 + 1\}; I_1 - 1, I_1; N \rangle
\]
that maps the RSD for \( X \):
\[
F_c((X; I_1; N)) \mapsto \langle V; I_1 + I_2 - 1, I_2; N - I_2 + 1, N \rangle
\]
that can be standardized into:
\[
\langle V; N + 1 - I_1, I_2; N + 1 - I_2, N \rangle
\]
Note that this is the same RSD as would have been obtained by direct construction after
in-line substitution and that \( I_1 \) in the function representation maps to \( I_2 \) in the resulting
RSD because the input RSD already has 1 loop.
4.6 A Data-flow Problem

For each code segment in a program, and each array parameter, initial RSD's describing what portions of the array are directly used or modified by the subroutine can be constructed from local information. The twin problems for summary of use and summary of modification are analogous, so we will discuss only summary of modifications here. For a given program array \( A \) and subroutine \( P \), let \( R^0_{P,A} \) represent the RSD constructed by direct inspection of the occurrences of \( A \) in \( P \). Our goal is to determine for \( A \) and \( P \) an RSD, \( R_{A,P} \), that describes all of \( A \) that could be modified by any invocation of \( P \), including the effects of any routines invoked by \( P \). Like more traditional versions of the summary problem [Coop 83], this problem is modeled as a flow-insensitive, data-flow problem over the program's call graph.

The set of regular sections of an array \( A \) form a lattice under the relation \( R_1 \subseteq R_2 \) if \( R_1 \supset R_2 \). The set of all elements of \( A \) is the bottom element (\( \bot \)) and the empty set is the top (\( \top \)) element for this lattice.

Let \( C = (V, E) \) be the call graph of a program. The nodes \( V \) represent procedures and the directed edges \( E \) represent call sites where the source procedure may invoke the sink procedure. We assume that each edge in the call graph is annotated with information describing the binding of formal parameters to actual parameters occurring at the associated call site. In particular, when formal parameter \( f \) is bound to actual parameter \( a \) at call site \( c \), the edge associated with \( c \) is annotated with translation functions \( F^{a\rightarrow b}_{c} \) of the form described in the previous section.

Suppose that we have an algorithm \( M \) that maps a pair of regular sections into the "smallest" regular section that contains their set-wise union. Smallest will be left undefined for now. If \( R_1 \) and \( R_2 \) are two regular sections associated with the same array, we let \( R_1 \land R_2 \) represent the result of \( M \) applied to these regular sections. If \( R^0_{A,P} \), and \( F^{a\rightarrow b}_{c} \) are interpreted as regular sections and functions on regular sections, then we can describe \( R_{A,P} \) as follows: let \( p = c_1 \cdots c_n \) be a path in the call graph with \( P \) the source of \( c_1 \) and \( Q \) is the sink of \( c_n \); let \( A_0, A_1, \ldots, A_n \) be a sequence of array parameters with \( A_0 = A \) such that at call site \( c_i \), \( A_{i-1} \) is bound to \( A_i \); and let \( \text{paths}(P) \) be the set of all such paths, then:

\[
R_{A,P} = R^0_{A,P} \land \bigwedge_{p = c_1 \cdots c_n \in \text{paths}(P)} \{ F^{A_0 \rightarrow A_1}_{c_1} \circ \cdots \circ F^{A_{n-1} \rightarrow A_n}_{c_n} (R^0_{A_n, Q_n}) \}
\]
This can be interpreted as specifying the regular section of $A$ associated with $P$ as the "smallest" regular section that includes both the modifications made directly by $A$ and the modifications made directly to $A$ by any procedure that could be invoked directly or indirectly from $P$.

This represents an ideal solution to the array summary problem. It also seems clear that it is not computable. Even if a reasonable, well-defined and computable definition of "smallest" could be found, the complexity of the lattice indicates that the resulting data-flow analysis framework [KaUl 77] would be monotone and not distributive and hence the "meet-over-all-paths" solution is not computable [KaUl 77].

Therefore, we simplify the summary problem in two ways. The first is that we solve the least fixed point of the following set of simultaneous equations:

$$R_{A,P} = R_{A,P}^0 \land \bigwedge_{c \in (P,Q)} \{F_{c}^{A \rightarrow B}(R_{B,Qc}^0)\}$$

where $c$ is a call site in $P$ that invokes procedure $Q$, binding parameter array $A$ to formal parameter $B$. This is a standard simplification technique for data-flow problems. The other simplification is to work with RSD’s (instead of regular sections) and use the merge operation outlined in previous sections as the meet function.

These simplifications alone are not sufficient to guarantee termination: the meet operation still does not distribute over the functions associated with call sites. Another problem is that the lattice of symbolic RSD’s contains infinite chains. Consider the recursive procedure:

```plaintext
SUBROUTINE TEMP(V,M,N)
REAL V(N)
V(1) = M
IF (M.GT.1) CALL TEMP(V(2),M-1,N-1)
RETURN
END
```

The initial RSD for $V$ is $R_{V,TEMP}^0 = (V;1;)$ and as we iterate through the above equations we see the sequence of RSD’s:

<table>
<thead>
<tr>
<th>Old $R_{V,TEMP}$</th>
<th>$f_{c}^{V \rightarrow V}(R_{V,TEMP})$</th>
<th>New $R_{V,TEMP}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$(V;1;)$</td>
<td>$(V;2;)$</td>
<td>$(V;I_1;2)$</td>
</tr>
<tr>
<td>$(V;I_1;2)$</td>
<td>$(V;I_1 + 1;2)$</td>
<td>$(V;I_1;3)$</td>
</tr>
<tr>
<td>$(V;I_1;3)$</td>
<td>$(V;I_1 + 1;3)$</td>
<td>$(V;I_1;4)$</td>
</tr>
<tr>
<td>$(V;I_1;4)$</td>
<td>$(V;I_1 + 1;4)$</td>
<td>$(V;I_1;5)$</td>
</tr>
</tbody>
</table>
This sequence alone suggests that at some point we should "jump" to \(\langle V; I_1; s \rangle\), where \(s\) is a non-program variable indicating an unknown upper bound. However, a more useful RSD would be \(\langle V; I_1; N \rangle\) which is also a fixed point for this example. It is not clear how either of these decisions can be made with in the data-flow framework.

4.7 Restrictions

Since the lattice described above contains infinite descending chains, to guarantee termination we need to modify the general problem to avoid the infinite chains. This section explores a less general form of summary information. By making some restrictions on the very general definition of RSD used previously, a much simpler lattice can be developed.

**Definition 4.2** A Restricted Regular Section Descriptor (RRSD) is a pair of the form \(\langle A; \bar{\theta} \rangle\) where \(A\) is the name of an array variable and \(\bar{\theta}\) is a vector of subscript positions such that each \(\theta_i\) is either a constant (routine invariant) expression or is of the form \(\pm I_j + \alpha\) where \(\alpha\) is routine invariant.

An RRSD is an RSD with simpler expressions, no shape information, and no upper bounds information. The regular section described by a constant RRSD \(R = \langle A, \bar{\theta} \rangle\), where each expression is an integer constant, is the set of all subscript vectors \(\bar{\theta}\) such that there exists a vector of integers \(\bar{t}\) such that:

\[
b_j = \theta_j[\bar{t}]
\]

where \(\theta_j[\bar{t}]\) is the value of \(\theta_j\) with each occurrence of \(I_k\) replaced with the constant \(i_k\). Note that \(i_k\) may be negative and that the array \(A\) is used as if any vector of integer values refers to a unique element of \(A\). This regular section will be denoted \(A^{-1}(R)\). Again, symbolic RRSD's are interpreted as functions from assignments of values to program variables to regular sections.

The access patterns that can be described with RRSD's include: single elements, complete rows and columns, and diagonals as well as their higher-dimensional analogues. Two access patterns that cannot be described precisely are triangular regions and discontiguous portions of rows:
DO 10 I = 1,N
   A(2*I) = ...
DO 10 J = 1,J
   B(I,J) = ...
10  CONTINUE

Since the primary application of summary information is for detection of parallel DO loops, this loss of precision does not seem very significant.

An equivalence relation between two RRSD’s is easy to define: since an induction variable \( I_j \) can assume any integer value, an expression \( I_j + \alpha \) can also assume any integer value, so replacing each occurrence of \( I_j \) by \( I_j + \alpha \) preserves regular sections. Likewise, if \( I_k \) does not appear in an RRSD \( R \), replacing each occurrence in \( R \) of \( I_j \) by \( I_k \) preserves regular sections. Two RRSD’s are equivalent if a sequence of these replacements can transform one into the other. An RRSD \( R = \langle A, \delta \rangle \) is in normal form if \( I_j \) appears in \( R \), then \( \delta_j = I_j \). It is clear that every RRSD is equivalent to exactly one normal form RRSD. Transforming an RRSD into the equivalent normal form is called normalizing and the result of normalizing \( R \) will be denoted \( \eta(R) \). Two RRSD’s that normalize to the same RRSD are said to be equivalent, and the set of RRSD’s that normalize to a particular normal form is called an equivalence class. Note that every RRSD in an equivalence class has the same number of distinct induction variables. This will be important in showing that the lattice of RRSD’s does not have infinite chains.

**Definition 4.3** The rank of an RRSD is the number of distinct induction variables that occur in the subscript expressions of that RRSD. The rank of an RRSD \( R \) will be denoted \( \rho(R) \).

Observe that if \( k \) is the number of dimensions in the array associated with an RRSD, that RRSD has \( k \)-subscript positions and each of these can reference at most one induction variable. Therefore we have argued the following lemma:

**Lemma 4.1** If \( k \) is the number of dimensions of an array \( A \), then \( 0 \leq \rho(R) \leq k \) for all restricted regular section descriptors of regular sections of \( A \).

To impose a lattice structure on these RRSD’s we need to develop a notion of “approximation” suitable for restricted regular sections descriptors. We use the following definition based on regular sections:
Definition 4.4. An RRSD \( R_1 \) approximates and RRSD \( R_2 \) if (under any assignment of values to program variables) the regular section described by \( R_1 \) contains the regular section described by \( R_2 \). This is denoted \( R_1 \subseteq R_2 \).

The next theorem ties together the syntactic concepts of rank and equivalence with the geometric concept of containment.

Theorem 4.1. If \( R_1 \) and \( R_2 \) are two RRSD's, and \( R_1 \subseteq R_2 \), then \( \rho(R_1) \geq \rho(R_2) \) and if \( \rho(R_1) = \rho(R_2) \), then \( R_1 \) and \( R_2 \) are equivalent.

Proof: By contradiction: assume \( \rho(R_1) < \rho(R_2) \), and let \( \eta(R_1) = \langle A, \vec{a} \rangle \) and \( \eta(R_2) = \langle A, \vec{b} \rangle \) be the normal forms of \( R_1 \) and \( R_2 \) respectively. Let \( I_j \) be an induction variable that appears in \( \eta(R_2) \) but not in \( \eta(R_1) \). If either \( R_1 \) or \( R_2 \) is symbolic, then assume every program variable is assigned a value of 0. There are two cases to consider:

1. \( \phi_j = I_j \) and \( \theta_j = \alpha \) where \( \alpha \) is a constant.

2. \( \phi_j = I_j \) and \( \theta_j = \pm I_k + \alpha \) where \( \alpha \) is a constant and \( k < j \).

If \( \theta_j = \alpha \), for some constant, then consider the array element obtained by setting each induction variable to 1 in \( \eta(R_2) \) except for \( I_j \) which is set to \( \alpha - 1 \). This contradicts the assumption that the regular section described by \( \eta(R_1) \) contains the regular section described by \( \eta(R_2) \) (under the current variable assignment) since the regular section described by \( \eta(R_1) \) contains only vectors \( \vec{b} \) with \( b_j = \alpha \).

If \( \theta_j = I_k + \alpha \), where \( k < j \), then again consider the array element obtained by setting each induction variable to 1 in \( \eta(R_2) \) except for \( I_j \) which is set to \( \alpha \). This contradicts the assumption that the regular section described by \( \eta(R_1) \) contains the regular section described by \( \eta(R_2) \) (under the current variable assignment) since the regular section described by \( \eta(R_1) \) contains only vectors \( \vec{b} \) with \( b_k - b_j = -\alpha \) and this vector has \( b_k - b_j = 1 - \alpha \). If \( \theta_j = \alpha - I_k \), this argument still holds, since the vectors of \( R_1 \) satisfy \( b_k + b_j = \alpha \) but at least one of \( R_2 \) does not.

If \( \rho(R_1) = \rho(R_2) \) and \( \eta(R_1) \neq \eta(R_2) \), two cases hold. Either there exists \( I_j \) that appears in \( \eta(R_2) \) and not in \( \eta(R_1) \), in which case we apply a construction just like the above to show that the regular section described by \( R_2 \) is not contained in the regular section described by \( R_1 \) contradicting the hypothesis. Otherwise, there exists a subscript position \( j \) such that \( \theta_j \neq \phi_j \). If these are both constants, then again the containment hypothesis is contradicted, and if \( \theta_j \) is a constant and \( \phi_j \) varies, say with \( I_k \), a value can be selected for \( I_k \) so that \( \phi_j[I] \) differs from the \( \theta_j[I] \) and so containment is violated.
If $\theta_j$ varies with an induction variable, then, since $\eta(R_1)$ and $\eta(R_2)$ have the same rank, if $\theta_j = I_j$, $I_j$ does not appear in $\eta(R_3)$ and there must exist some $I_k$ that appears in $\eta(R_2)$ that does not appear in $\eta(R_1)$. Thus we reduce to a previous case.

If $\phi_j = I_j$ and $\theta_j$ is either constant or varies with some other induction variable, say $I_k$, we can select a value for $I_j$ that is not equal to either $\theta_j$, $I_k - \theta_j$, or $I_k + \theta_j$ and so violate the containment hypothesis.

The last case is $\phi_j = \pm I_k - \alpha$ and $\theta_j = \pm I_k - \beta$ where $k < j$, and either $\alpha \neq \beta$ or the signs of the induction variables are different. In any case, all vectors in the regular section described by $R_1$ satisfies either:

$$b_k - b_j = \beta$$

or:

$$b_k + b_j = -\beta$$

but there exist vectors in the regular section described by $R_2$ that do not. Hence the containment criterion is violated.

*End of Proof.*

**Corollary** $R_1 \subseteq R_2$ and $R_2 \subseteq R_1$ if and only if $\eta(R_1) = \eta(R_2)$.

The next theorem provides a syntactic characterization of approximation which will provide the basis for an algorithm to merge two RRSD's into a single RRSD.

**Definition 4.5** Let $R = \langle A; \bar{\theta} \rangle$ be an RRSD, then a linear partition of $R$ is a collection of sets $V_0, V_1, \ldots, V_{\rho(R)}$ where $V_0$ is the set of subscript positions of $R$ that are constant (invariant expressions) and each $V_i$ corresponds to a set of subscripts positions that vary with the same induction variable.

Example: For RRSD $\langle A; 3, I_2, I_3, I_2 \rangle$, a linear partition consists of $V_0 = \{1\}$, $V_1 = \{2, 4\}$ and $V_2 = \{3\}$. When a set $V_i$ other than $V_0$ has more than one element, that set represents a collection of subscripts that vary together – a diagonal. An example is $V_1$ in the last example.
Definition 4.6 Let \( R_1 = \langle A; \theta \rangle \) and \( R_2 = \langle A; \phi \rangle \) be two RRSD's and let \( V^1_0, \ldots, V^1_{\rho(R_1)} \) and \( V^2_0, \ldots, V^2_{\rho(R_2)} \) be linear partitions of \( R_1 \) and \( R_2 \) respectively. Then \( R_1 \) textually approximates \( R_2 \) (denoted \( R_1 \preceq R_2 \)) if the following conditions are satisfied:

a. \( V^1_0 \subseteq V^2_0 \) and for all \( j \in V^1_0 \), \( \theta_j = \phi_j \)

b. For all \( i \) such that \( 1 \leq i \leq \rho(R_1) \), there exists \( k, 0 \leq k \leq \rho(R_2) \) such that \( V^1_i \subseteq V^2_k \)

c. For all \( i, j, k \in V^1_i \), either \( \theta_j - \theta_k = \phi_j - \phi_k \), where both sides are constants, or \( \theta_j + \theta_k = \phi_j + \phi_k \), where both sides are constants.

The first constraint indicates that the constant subscripts that appear in the “larger” RRSD \( (R_1) \) must also appear in the “smaller” RRSD \( (R_2) \). The second constraint indicates that a diagonal constraint in the larger is contained in a diagonal constraint of the smaller. The last condition indicates that a diagonal constraint in \( R_1 \) is the same diagonal as in \( R_2 \) and not a parallel one.

Theorem 4.2 For any two RRSD’s \( R_1 \) and \( R_2 \), \( R_1 \preceq R_2 \) if and only if \( R_1 \subseteq R_2 \).

Proof: Assume \( \bar{\theta}, \bar{\phi}, \) and \( V^j_i \) are as in the last definition. Assume that \( R_1 \subseteq R_2 \) but \( R_1 \not\subseteq R_2 \), then there exists \( i \) and subscript positions \( j, k \in V^1_i \) such that one of the following four cases applies:

1. \( j \in V^1_0 \) is constant and \( j \in V^2_0 \) and \( \theta_j \neq \theta_k \) (violates condition (a) of Definition 4.6).
2. \( j \in V^1_0 \) is constant and \( k \in V^2_l, l \neq 0 \) (violates condition (a)).
3. \( j \in V^1_i \) and \( k \in V^2_l \) and \( l_1 \neq l_2 \) (violates condition (b)).
4. \( \theta_j - \theta_k \neq \phi_j - \phi_k \) and \( \theta_j + \theta_k \neq \phi_j + \phi_k \) (violates condition (c)).

To show that each of these leads to a contradiction, we must select an assignment of values for program variables and construct an element \( \bar{b} \), in the regular section described by \( R_2 \) that is not in the regular section described by \( R_1 \) under this assignment. Unless otherwise specified, assume that every variable is assigned the value 0.

The first case is easy since every vector \( \bar{b} \) in the regular section described by \( R_2 \) has \( b_j = \alpha \) for some \( \alpha \) and no vector \( \bar{b} \) in the regular section described by \( R_1 \) has \( b_j = \alpha \), the regular section described by \( R_2 \) is not contained in the regular section described by \( R_1 \), contradicting the hypothesis (example: Figure 22(a)).

The second case corresponds to the case in which \( \phi_j = \alpha \) is a constant and \( \phi_k = \pm I_i + \beta \) for some \( I_i \). By construction, we know that \( \theta_j = \pm I_i + \gamma \) and \( \theta_k = \pm I_i + \delta \) for some \( i \).
Case 1: \[ R_1 = (A; 2) \quad \theta \quad \vec{b} = (3) \]
\[ R_2 = (A; 3) \quad \phi \]
(a)

Case 2: \[ R_1 = (A; I_1, I_1) \quad \theta \quad \vec{b} = (2, 1) \]
\[ R_2 = (A; I_2, I_2) \quad \phi \]
(b)

Case 3: \[ R_1 = (A; I_1, I_1) \quad \theta \quad \vec{b} = (0, 1) \]
\[ R_2 = (A; I_2, I_2) \quad \phi \]
(c)

Case 4: \[ R_1 = (A; I_1, I_1 + 1) \quad \theta \quad \vec{b} = (1, 3) \]
\[ R_2 = (A; I_1, I_1 + 2) \quad \phi \]
(d)

Figure 23. Examples for Theorem 4.2: \( \vec{b} \in A(R_2), \vec{b} \notin A(R_1) \)

Assume that in both cases the sign on the loop induction variable is positive. All other cases are analogous to this one. Select the vector \( \vec{i} \):

\[ i_k = \begin{cases} 0 & \text{if } k \neq l \\ \alpha - \beta + \delta - \gamma + 1 & \text{otherwise} \end{cases} \]

then let \( \vec{b} = \vec{\phi}[\vec{i}] \) so \( \vec{b} \in A^{-1}(R_2) \) and \( b_j = \alpha \) and \( b_k = \alpha + \delta - \gamma + 1 \), but all vectors \( \vec{b} \in A^{-1}(R_1) \) must satisfy \( b_k - b_j = \delta - \gamma \) and hence \( \vec{b} \) is not in \( A^{-1}(R_1) \) contradicting the assumption that \( R_1 \subset R_2 \) (example: Figure 22(b)).

The third case corresponds the case in which \( \phi_j = \pm I_l + \alpha \) and \( \phi_k = \pm I_k + \beta \) where \( l_1 \neq l_2 \). Let \( \theta_j \) and \( \theta_k \) be as above. Again, assume that the sign of each loop induction variable is positive, all other case are analogous. Using the vector \( \vec{i} \):

\[ i_k = \begin{cases} 0 & \text{if } k \neq l_2 \\ \alpha - \beta + \delta - \gamma & \text{otherwise} \end{cases} \]

let \( \vec{b} = \vec{\phi}[\vec{i}] \) so \( \vec{b} \in A^{-1}(R_2) \) and \( b_j = \alpha \) and \( b_k = \alpha + \delta - \gamma + 1 \), but again, all vectors \( \vec{b} \in A^{-1}(R_1) \) must satisfy \( b_k - b_j = \delta - \gamma \), hence \( \vec{b} \) is not in \( A^{-1}(R_1) \) contradicting the assumption that \( R_1 \subset R_2 \) (example: Figure 22(c)).

The final case corresponds to \( \phi_j \) and \( \phi_k \) varying with the same loop, but the difference \( \phi_j \pm \phi_k \) is not the same as \( \theta_j \pm \theta_k \). Intuitively, parallel but distinct diagonals are being described. Let \( \phi_j = \pm I_l + \alpha \) and \( \phi_k = \pm I_k + \beta \) and \( \theta_j \) and \( \theta_k \) be as before. Again, consider the case where all coefficients have positive signs, the other cases being analogous. Using \( \vec{i} \) from the second case, construct \( \vec{b} = \vec{\phi}[\vec{i}] \) so \( \vec{b} \in A^{-1}(R_2) \) and \( b_j = \alpha \) and \( b_k = \alpha + \delta - \gamma + 2 \). Again, all vectors \( \vec{b} \in A^{-1}(R_1) \) must satisfy \( b_k - b_j = \delta - \gamma \), hence \( \vec{b} \) is not in \( A^{-1}(R_1) \) contradicting the assumption that \( R_1 \subset R_2 \) (example: Figure 22(d)).
To show that the converse holds, assume that $R_1 \subseteq R_2$ holds. We want to show that $R_1 \subseteq R_2$. Let $\vec{b} \in A^{-1}(R_2)$ and let $\vec{i}$ be such that $\vec{b} = \vec{f}[\vec{i}]$. Each loop control variable $I_k$ that appears in $R_1$ corresponds to some $V_j^1$. Select a subscript position $j$ from that set. Setting $\theta_j[\vec{i}'] = \phi_j[\vec{i}]$ defines the value for $i_k'$. I claim now that $\vec{b} = \vec{f}[\vec{i}']$ which establishes that $R_1 \subseteq R_2$.

For each subscript position $j$, if $\theta_j$ is constant, then $j \in V_j^2 \subseteq V_j^1$ and $\theta_j = \phi_j$, therefore $\theta_j[\vec{i}'] = b_j$. Else, assume $j \in V_j^1$: then by construction there exists $k \in V_k^1$ such that $\theta_k[\vec{i}'] = \phi_k[\vec{i}] = b_k$. By assumption we have:

$$\theta_j - \theta_k = \phi_j - \phi_k$$

or:

$$\theta_j + \theta_k = \phi_j + \phi_k$$

Assume the former, the case for the latter is analogous. The expressions on either side of these equations are constants and do not vary with $\vec{i}$. Thus:

$$\theta_j[\vec{i}'] = \theta_k[\vec{i}'] + (\phi_j - \phi_k)$$

$$= \phi_k[\vec{i}] + (\phi_j - \phi_k)$$

$$= \phi_j[\vec{i}]$$

$$= b_j$$

_Ende of \text{\textit{End of}}_{\text{\textit{End of}}_{\text{\textit{End of}}}}$

This theorem provides a basis for an algorithm to combine two RRSD's into a single RRSD approximating them both. An important property of this algorithm is that it generates the "smallest" RRSD that approximates both inputs. Otherwise more information is lost during the in merge than is necessary to stay in the restricted regular section framework. The approach used in merge algorithm given below is to construct a partition of the result RRSD that satisfies the conclusion of Theorem 4.2.

**Algorithm 2.RRSD Merge**

**Input.** Two RRSD's, $R_1 = \langle A; \vec{g} \rangle$ and $R_2 = \langle A; \vec{g} \rangle$.

**Output.** RRSD $R_3$ that approximates both $R_1$ and $R_2$.

**Method:** Let $V_i^1$ and $V_i^2$ be as in the Definition 4.6. Let $W_0, \ldots, W_n$ be the coarsest partition such that for each $W_j$ there exist $V_j^1$ and $V_j^2$ such that $W_j \subseteq V_j^1$ and $W_j \subseteq V_j^2$. 


Further refine each \( W_j \) into the coarsest refinement \( W_{j,1}, \ldots, W_{j,t} \) such that for each pair \( i, k \in W_{j,m} \) either:

\[
\theta_j - \theta_k = \phi_j - \phi_k = \alpha
\]
or:

\[
\theta_j + \theta_k = \phi_j + \phi_k = \alpha
\]

for some invariant expression \( \alpha \). Refine \( W_0 \), the positions that are constant in both \( R_1 \) and \( R_2 \) into \( W_{0,0}, \ldots, W_{0,1}, \ldots, W_{0,t} \) where:

\[
\theta_j - \theta_k = \phi_j - \phi_k = \alpha
\]

for each pair \( j, k \in W_{0,m} \).

Construct \( R_3 = \langle A; \vec{\sigma} \rangle \) by:

\[
\sigma_i = \begin{cases} 
\theta_i & \text{if } i \in W_{0,0} \\
I_i & \text{if } i = \min W_{n,m} \text{ for some } m, n \\
I_j - \alpha & \text{if } i \in W_{m,n}, k = \min W_{n,m} \text{ and } \alpha = \theta_k - \theta_j \text{ for some } m, n \\
-I_j + \alpha & \text{if } i \in W_{m,n}, k = \min W_{n,m} \text{ and } \alpha = \theta_k + \theta_j \text{ for some } m, n 
\end{cases}
\]

End of Algorithm.

The output of Algorithm 2 on inputs \( R_1 \) and \( R_2 \) will be denoted \( \text{merge}(R_1, R_2) \).

**Theorem 4.3** For any RRSD's \( R_1, R_2 \), \( \text{merge}(R_1, R_2) = \text{merge}(R_2, R_1) \), \( \text{merge}(R_1, R_2) \subseteq R_1 \), and if \( R_3 \subseteq R_1 \) and \( R_3 \subseteq R_2 \), then \( R_3 \subseteq \text{merge}(R_1, R_2) \).

**Proof:** \( \text{merge}(R_1, R_2) = \text{merge}(R_2, R_1) \) follows by inspection of Algorithm 2 since it treats its parameters symmetrically. Likewise, \( \text{merge}(R_1, R_2) \subseteq R_1 \) follows from the construction of the sets \( W_{n,m} \) and Theorem 4.2.

To show the last claim, that \( R_3 \subseteq \text{merge}(R_1, R_2) \) I will use the syntactic notion of approximation and show that \( R_3 \preceq \text{merge}(R_1, R_2) \). Let \( W_{n,m} \) be the linear partition constructed for \( \text{merge}(R_1, R_2) \) by Algorithm 2. Assume \( R_1 = \langle A; \vec{\theta} \rangle \) and \( R_2 = \langle A; \vec{\phi} \rangle \) and let \( V^1_0, \ldots, V^1_{1(R_1)} \) and \( V^2_0, \ldots, V^2_{1(R_2)} \) be linear partitions for \( R_1 \) and \( R_2 \) respectively. Let \( R_3 = \langle A; \vec{\psi} \rangle \) and \( Z_0, \ldots, Z_{1(R_3)} \) be a linear partition of \( R_3 \). Finally, let \( \text{merge}(R_1, R_2) = \langle A; \vec{\sigma} \rangle \) If \( R_3 \npreceq \text{merge}(R_1, R_2) \), then one of the following conditions holds for some pair \( j, k \) in some \( Z_i \)

1. \( j \in Z_0 \) and \( j \in W_{0,0} \) and \( \psi_j \neq \sigma_j \).
2. \( j \in W_{0,0} \) is constant and \( k \in W_{n,m} \), \( n \neq 0 \) or \( m \neq 0 \).
3. \( j \in W_{n_1,m_1} \) and \( k \in W_{n_2,m_2} \) and \( n_1 \neq n_2 \) or \( m_1 \neq m_2 \).

4. \( \psi_j - \psi_k \neq \sigma_j - \sigma_k \) and \( \psi_j + \psi_k \neq \sigma_j + \sigma_k \)

For the first case, if \( \psi_j \) is a constant, then \( \theta_j \) and \( \phi_j \) must be the same constant, in which case the merge algorithm will put \( j \) in \( W_{0,0} \), hence \( \sigma_j = \theta_j = \psi_j \) (example: Figure 23(a)).

For the second case, \( \sigma_j \in W_{0,0} \) (a constant) indicates that both \( \theta_j \) and \( \phi_j \) are constants and \( \theta_j = \phi_j = \sigma_j \). Since \( Z_i \) is contained in \( V_i^1 \) for some \( i \) and we know that \( j \in Z_i \) is also in \( V_i^1 \), we conclude that \( Z_i \subset V_0^1 \). Analogously, \( Z_l \subset V_0^2 \). Therefore \( \theta_k \) and \( \phi_k \) must be constants. Further, they must be different constants since \( k \not\in W_{0,0} \). However:

\[
\psi_j - \psi_k = \theta_j - \theta_k
\]

since \( R_3 \leq R_1 \) and:

\[
\psi_j - \psi_k = \phi_j - \phi_k
\]

since \( R_3 \leq R_2 \) and therefore:

\[
\theta_k - \phi_k = \theta_j - \phi_j = 0
\]

contradicting the fact that \( \theta_k \) and \( \phi_k \) are different (example: Figure 23(b)).

The third case has the following facts: there exists \( V_i^1 \) and \( V_h^2 \) such that \( Z_l \subset V_i^1 \) and \( Z_l \subset V_h^2 \). Therefore there exists some \( W_n = V_i^1 \cup V_h^2 \) such that \( Z_l \subset W_n \). We know that:

\[
\psi_k - \psi_j = \theta_k - \theta_j = \phi_k - \phi_j
\]

and therefore there exists \( W_{n,m} \) such that \( \{j,k\} \subset W_{n,m} \), contradicting the assumption of case 3 (example: Figure 23(c)).

The fourth case follows directly out of the argument from the third case since:

\[
\sigma_k - \sigma_j = \theta_k - \theta_j = \psi_k - \psi_j
\]

contradicting the assumption of the fourth case (example: Figure 23(d)).

End of Proof.
Case 1: \( R_3 = \langle A; 2 \rangle \)
\[
\text{merge}(R_1, R_2) = \langle A; \sigma_1 \rangle \\
R_1 = \langle A; 2 \rangle \\
R_2 = \langle A; 2 \rangle \\
\psi \\
\sigma \\
\phi \\
\theta \\
\Rightarrow \sigma_1 = 2
\]

Case 2:
\[
R_3 = \langle A; 2, 3 \rangle \\
\text{merge}(R_1, R_2) = \langle A; 2, \sigma_2 \rangle \\
R_1 = \langle A; 2, 3 \rangle \\
R_2 = \langle A; 2, 3 \rangle \\
\psi \\
\sigma \\
\phi \\
\theta \\
\Rightarrow 2 \in Z_0 \\
\Rightarrow \phi_2 = \theta_2 \text{ a constant} \\
\Rightarrow 2 \in W_{0,0}
\]

Case 3:
\[
R_3 = \langle A; I_1, J_1 \rangle \\
\text{merge}(R_1, R_2) = \langle A; I_1, J_2 \rangle \\
R_1 = \langle A; I_2, J_2 \rangle \\
R_2 = \langle A; I_3, J_3 \rangle \\
\psi \\
\sigma \\
\phi \\
\theta \\
\Rightarrow Z_1 \subset V_1^2 
\Rightarrow Z_1 \subset V_1^2 \text{ and } Z_1 \subset V_1^2
\]

Case 4:
\[
R_3 = \langle A; I_1, J_1 + 1 \rangle \\
\text{merge}(R_1, R_2) = \langle A; I_1, J_1 + 2 \rangle \\
R_1 = \langle A; I_1, J_1 + 1 \rangle \\
R_2 = \langle A; I_1 - 1, J_1 \rangle \\
\psi \\
\sigma \\
\phi \\
\theta \\
\Rightarrow Z_1 \subset W_{1,1} \subset \{1, 2\} \\
\Rightarrow \sigma - \sigma_j = \theta_k - \theta_j = \psi_k - \psi_j
\]

Figure 24. Examples for Theorem 4.3: \( j = 1, k = 2 \)

**Theorem 4.4** Let \( R_3 = \text{merge}(R_1, R_2) \). If \( \eta(R_1) = \eta(R_2) \), then \( \eta(R_1) = \eta(R_3) \) and if \( \eta(R_1) \neq \eta(R_3) \), then \( \rho(R_3) > \rho(R_1) \).

**Proof:** The first follows immediately from Theorem 4.3: if \( \eta(R_1) = \eta(R_2) \) then \( R_1 \sqsupseteq R_1 \) and \( R_1 \sqsubseteq R_2 \), therefore \( R_1 \sqsubseteq R_3 \sqsubseteq R_1 \). Hence \( \eta(R_1) = \eta(R_3) \) by the corollary to Theorem 4.1. If \( \eta(R_1) \neq \eta(R_3) \) then, since \( R_3 \sqsubseteq R_1 \), \( \rho(R_3) > \rho(R_1) \) by Theorem 4.1.

**End of Proof.**

Enough machinery has been developed to define a semi-lattice based on restricted regular sections and their descriptors. Let \( L \) be the set of normal form RRSD's for some \( k \)-dimensional array variable \( A \). Define a partial ordering on \( L \) by \( \sqsubseteq \). Define the meet operator \( \wedge \) by \( R_1 \wedge R_2 = \text{merge}(R_1, R_2) \). Under these definitions, the following theorems hold\(^5\).

---

\(^5\) See [KaUl 77] for appropriate definitions.
Theorem 4.5 \( \langle L, \wedge \rangle \) is a bounded, semilattice.

Proof: By inspection, the merge algorithm is idempotent, reflexive and associative, therefore \( \langle L, \wedge \rangle \) is a semilattice. By Lemma 4.1 and Theorem 4.1, no chain has length greater than \( k + 1 \) and hence \( \langle L, \wedge \rangle \) is bounded.

End of Proof.

Theorem 4.6 For \( R_1 \) and \( R_2 \) in \( L \), the greatest lower bound of \( R_1 \) and \( R_2 \) is \( \text{merge}(R_1, R_2) = R_1 \wedge R_2 \).

Proof: Immediate from Theorem 4.3.

Translation functions associated with call sites will map an RRSD of the formal parameter, \( R_F \), into an RRSD of the actual parameter, \( R_A \), in the following way: each subscript position of \( R_A \) is assigned either a constant expression or a subscript position of \( R_F \). In the latter case, whatever expression appears in the subscript position of the \( R_F \) is copied over into the subscript position of the \( R_A \). For example, at the call site:

\[
\begin{align*}
\text{REAL } & \text{A(N,N,N),B(N,N),C(N,N)} \\
\text{CALL } & \text{SUB1(A(1,1),B(1,1),C(I,J),N)} \\
& \vdots \\
\text{SUBROUTINE } & \text{SUB1(X,Y,Z,N)} \\
\text{REAL } & \text{X(N,N),Y(N,N),Z}
\end{align*}
\]

For the first variable, an RRSD for \( X \) of \( \langle X; \alpha, \beta \rangle \) would translate to an RRSD of \( A \) equal to \( \langle A; \alpha, \beta, J \rangle \). An RRSD for \( Y \) of \( \langle Y; \alpha, \beta \rangle \) would translate to an RRSD of \( B \) equal to \( \langle B; \alpha, \beta, J \rangle \). An RRSD for \( Z \) of \( \langle Z; \rangle \) would translate to an RRSD of \( B \) equal to \( \langle C; I, J \rangle \).

4.8 Translation Functions for RRSD's

Languages such as Fortran and C allow programmers great freedom when passing arrays to subroutines. Only a pointer is passed and the programmer is free to base any shaped array on the location specified by the pointer. This generality of array parameter passing greatly complicates the translation of array summary information through a call site. While dropping information from the general RSD to get the restricted RSD allows a clean, bounded semilattice to be developed, the information that was dropped makes it difficult to translate information across call sites. Consider the following example:
REAL A(N,N)
::
CALL SUB1(A(I,I),N,N-I+1)
::
SUBROUTINE SUB1(V,INC,M)
REAL V(*)
::
J = 1
DO I = 1,M
  V(J) = ...
  J = J + INC
ENDDO

The RSD for V in SUB1 is \((V; INC \cdot I_1 + 1 - INC; M)\), which translates into the RSD at the call site for A of \((A; I, I_1 - 1 + I; N - I + 1)\). The fact that INC is bound to N, which is equal to the first bound of A, is needed to de-linearize the reference. This fact is omitted from the RRSD of V given by \((V; I_1)\).

A second example uses the same subroutine, but with the call site:

REAL A(N,N)
::
CALL SUB1(A(I,I),1,N-I+1)

The general RSD for V is the same, but this time a column of A is accessed instead of a row and the RSD for A would be \((A; I_1 + I - 1, I; N - I + 1)\). The fact that only \(N-I+1\) elements of V are accessed is needed to correctly de-linearize. Again, this information is omitted from the restricted RSD.

The first of these problems can be handled by allowing restricted RSD's to have coefficients associated with the loop induction variables. Minor modifications to the definition of normal form and textual approximation must be made. The merge operation adjusts coefficients so that containment is preserved. In general, the coefficient loop induction variable for a particular subscript position will either be the same as in the corresponding subscript position of the input RSD or it will be a divisor of that coefficient. This property preserves the finite chain property of the semi-lattice. It is not clear whether this generality is necessary, since only a few coefficients are interesting: those that correspond to rows, diagonals and similar sections of the actual parameters. Thus it seems practical
to perform some auxiliary analysis to determine how formal parameters relate to actual parameters.

The "coefficient" that is used as a stride in formal parameters and indicates how access patterns of the formal relate to access patterns of the actual can be derived from inspection of the direct accesses made to the formal parameter involved in the call site, and to the other formal parameters it is bound to indirectly: if \( A(\theta) \) is an actual parameter bound to \( F \), and we want to know if all accesses to \( A \) have coefficient \( c \), then all accesses to \( F \) must have coefficient of \( c \) and \( \theta \) must be one more than a multiple of \( c \).

The "coefficient" of the most significant dimension that appears in accesses to the formal determines how the formal relates to the actual: if it is 1, then the formal is a column; if it \( N \), the corresponding bound of the actual (as in the first example), then the access is a row; if it \( N + 1 \), then the access is a diagonal. These are cases where the formal has one fewer dimension than the actual; there are analogues for the cases when the formal has even fewer dimensions.

The other piece of information is harder; the presence of gradually increasing upper bounds leads to infinite chains in general RSD's. However, from the point of view of translating information from a subroutine to a caller, there are only a finite number of "interesting" values for upper bounds. These values correspond to a limit where accesses contained in one subsection (row, column, or diagonal) would "overflow" into the next. In the second example, we are only interested in knowing whether the upper bound of accesses to \( V \) is either in the range \([1..M]\) or in the range \([M + 1..\infty]\).

To determine how an RRSD describing access to a formal parameter translates into an RRSD for the actual parameter, we need to determine how the formal parameter's shape relates to that of the actual. The most common case is that they will be identical, but it will also be common for the formal parameter to have fewer dimensions than the actual. This is particularly true when the formal represents a subsection, like a row or column, of the actual. For simplicity, I will avoid the extreme cases where the shapes of the actual and the formal parameters are radically different. Worst case assumptions will be made for those situations. The cases I will consider more closely are those in which the formal parameter is a proper section\(^6\) of the actual, except that the bound of most

\(^6\) Definition 4.1
significant dimension of the formal need not be contained in the bound of the actual. The first example of this section illustrates this.

When the coefficient indicates that the formal may be contained in a column (or higher dimensional analogue) we will need an upper bound for how much of the formal is accessed (as in the second example). Since the formal shape declaration need not provide this information, our goal is to find an expression for each formal array parameter which bounds the largest value used as an index in the most significant dimension of that array. These expressions can be combinations of integers and formal parameters or a special symbol representing an unknown upper bound. These expressions must satisfy two constraints. The first is that they must bound the direct accesses made to the formal array parameter by the routine it is declared in. This can be checked by direct inspection of the general RSD information collected as summary. The second constraint is that, at every call site where actual parameter \( A(\theta) \) is bound to formal parameter \( F \), if \( N_F \) is the bound for \( F \) and \( M \) is that bound after substituting the parameters of the call in for the formal parameters in \( N_F \), then \( M + \theta \) must be less than or equal to \( N_A \), the bound for \( A \).\(^7\)

No technique is known to construct these bounds precisely from the information in the access summary information and call site information. The problem of infinite ascending chains prevents use of the obvious iterative approach. The approach suggested below uses information about what solution is “interesting”, and tries to construct a solution and prove it is a fixed point. It is an optimistic approach in the sense that Wegman-Zadeck constant propagation [WeZa 85] is optimistic, and is based on the observation that there are only a finite number of “interesting” solutions to the problem.

Initially, set the bound associated with each array to \( \top \), indicating that no bound is needed. Where a formal parameter has a different shape than the actual it is bound to, and we need the bound on the formal to establish a particular relation between accesses to the formal and access to the actual, and the formal’s bound is \( \top \), construct an expression in the context of the formal parameter that establishes the relationship suggested by the coefficients. In the first example, no information is needed: the coefficient indicates row access and no upper bound is need. In the second example, the coefficient indicates a

\(^7\) This assumes that both \( F \) and \( A \) are one dimensional, if they are higher dimensional, then \( \theta \) is the expression in the most significant subscript position of \( A \).
column access and we need to know if \( N - I + 1 \), which is \( M \) in the context of the formal parameter, is a valid bound. Thus \( M \) would be the initial bound for formal parameter \( V \) in the second example.

After this initial solution is constructed, we try to extend it to a fixed point by propagating bounds: where formal parameter \( F \) is passed as actual parameter and bound to another formal parameter \( G \), any bound on \( F \) will imply a new bound on \( G \). If \( G \) is not bound, then we make this new bound the bound for \( G \). If \( G \) is already bound, then we see if the old bound implies the new bound. If not, we set the bound on \( F \) to \( \bot \) to indicate an unknown upper bound. If the bound is set to bottom, and if some third formal \( H \) has been bound to \( F \), then the bound on \( H \) will also be \( \bot \). The bound \( \bot \) can never imply any other bound. Propagation must terminate since every formal parameter is assigned a bound at most twice.

Where we fail to find a bound, and information is used from a call site involving a shape change, it may be that some other, larger bound can be used, and a new solution generated and tested. For example, for a call site with actual parameter \( A(i, j, k) \), we may first try to show that the formal accesses only the column \( A(I_1, j, k) \). If that fails, we may try to show that the formal accesses only the block \( A(I_1, I_2, k) \). When second attempts are made, bounds set to \( \bot \) are reset to \( T \). Note that each new bound we try is based on a larger (in rank) regular section for the actual. Therefore, at most \( k \) different bounds will be attempted.

Some amount of auxiliary work could be done to improve the ability to test whether one expression is bounded by another. In particular, during initial examination of a procedure, we could construct an expression for each scalar argument in a parameter list at a call site, in terms of formal parameters and integers, that bounds it. This auxiliary information may be as simple as recognizing that induction variables are bounded by loop upper bounds (which are likely to be easy to bound in terms of formal parameters) or a full blown technique to derive assertions about variables could be done as in [Karr 76][CoHa 78]. This decision should wait until a (partial) implementation is available to determine the need for this auxiliary information.

Having determined how accesses to the formal parameter relates to accesses to the actual parameter for each call site that involves a shape change, we can solve the interprocedural summary problem using RRSD's.
4.9 Related Research

The section describes related techniques to more precisely handle loops that contain call sites. The first is an application of a classical optimization to this new situation; the second is an application of standard interprocedural data flow techniques; the third has the same goals but uses a very different representation; and the last is a method similar to general RSD’s.

4.9.1 In-line Expansion

In-line expansion (also called procedure integration[AlCo 72]) refers to the replacement of a subroutine or function call with the body of the called routine, after appropriate replacement of formal parameters with actual parameters. When this is done, all of the essentially local (single loop nest) analysis and transformations described above can be applied to the expanded routine body. The classical argument against expansion is the possibility of exponential growth in program size. While this technique solves the aliasing problem at the expanded call site, it does not completely solve the problem unless all calls are expanded. This is also true for interprocedural constant propagation. In-line expansion should be very effective for detecting parallel loops. It has the further advantage that parallel code in the subroutine can be separated (by loop distribution) from serial code in the subroutine, and so partial parallelization can be achieved.

4.9.2 Standard Interprocedural Analysis

An alternate approach is leaving the program structure intact and computing information which accurately summarizes: how a subroutine invocation may use and modify variables, what aliases may exist when a routine is invoked, and which parameters are constant for all invocations. Several techniques for computing this information can be found in the literature [Alle 74] [Bann 78] [Bart 77] [Myer 81] [Coop 83] [Torc 85] and some of these techniques have already been implemented [CCKT 86] in PFC [AIKe 82][AICK 86].

The principle flaw in these techniques is that the granularity of the information obtained is too large. It is not sufficient for precise dependence analysis to know simply that an array variable may be modified, it is also necessary to know how that array is modified.
4.9.3 Triolet’s Work

A technique has been proposed by Remi Triolet [IrTr 85, Trio 86] to overcome the problems of call sites. It extends standard interprocedural techniques by computing and propagating a collection of symbolic linear inequalities describing the sections of an array that are used or modified. Each collection is called a region. Regions can be merged by computing the convex hull of the linear assertions associated with the input regions. In addition, sets of linear inequalities are derived for each point in the program that are used to provide additional context. The data dependence analysis is extended to check for overlap between regions described by these sets of linear assertions by testing for consistency of the union of the two regions and the execution context.

Triolet has essentially the same problem with improper shape changes as discussed in section 4.8 and suggests a pessimistic approach [TrIF 86]. Triolet’s method applies only to acyclic graphs. Extending this approach to recursive programs will be difficult due to the same problems that troubled the general RSD approach. Triolet reports [Trio 86] that this approach is effective but warns [TrIF 86] that it is expensive.

4.9.4 Burke and Cytron

Burke and Cytron [BuCy 86a] report a comprehensive approach to dependence analysis that includes methods for both interprocedural summary information and alias detection. In an interprocedural context, all array references are linearized and viewed as one-dimensional accesses to memory. They extend standard alias detection algorithms by also computing minimum positive and maximum negative offsets between the first locations referenced by the two variables [BuCy 86b]. Their approach to the summary problem is essentially the same as the general RSD’s discussed in section 4.3 without the affine expression requirement. This approach is applied only to programs without dynamic recursion by using node splitting to break cycles. They avoid the problem of reshapes since all array references are fully linearized. To merge separate instances they suggest either a simple union where the individual access patterns are maintained or a more complicated union using an introduced iteration. Neither of these are particularly satisfying: the first approach could potentially consume a large amount of space and the second generates expressions that are non-linear in iteration variables, thus forcing the use of less precise dependence tests that may cause further loss of precision due to the linearization. Extending this approach to recursive problems will require the same information about upper bounds as are needed to determine whether a formal array parameter is a proper section of an actual array parameter.
4.10 Summary

The problem of extending interprocedural summary information to a lattice that provides information suitable for dependence analysis has been examined. An alternative form of summary information has been defined and transformations presented on that form which include merging multiple items of summary information together into a single item and translating summary information from the context of a called subroutine into the context of the caller. These two transformation provide the basis for an interprocedural data flow problem.

Even this restricted lattice is difficult to use when actual and formal array parameters have different shapes. An ad hoc, optimistic technique is suggested for determining how actual parameter access patterns relate to formal parameter access patterns.

The techniques of this chapter do not directly alter the algorithm of Chapter 2. Their effects are reflected in a more accurate, less conservative dependence graph.
Chapter 5:
Array Distribution

5.1 Introduction

In [Wolf 82], Wolfe describes a process called array contraction where the number of dimensions of an array variable can be reduced when no information flows from one section of that array to another. In particular, if an array is live only inside a serial loop, and each iteration of that loop accesses only one column of the array, then only one column is ever needed. He also suggests that for multi-processor machines with local memory, arrays can be distributed so that each element is allocated to a particular local memory.

Wolfe's goal was to reduce the space required for temporary arrays produced by unnecessary application of scalar expansion or simplistic code generation for vector oriented programming language. While he observes that array contraction can be applied to user variables, he admits that it is unlikely to have much effect.

In this chapter, I address the more ambitious problem of distributing user arrays to local memories for large sections of the program. The advantage of such a distribution is the reduction in global memory traffic caused by replacing global memory references with local memory references. This reduces the bandwidth needed for global memory and speeds up computation since local memory accesses will be faster (usually significantly faster) than global memory accesses. Also, the amount of global memory needed will be reduced if variables are distributed for their entire lifetimes. The distribution of shared arrays to local memories will affect which processors can execute particular iterations of parallel loops. While this may have negative effects on load balance, it can eliminate some synchronization that is otherwise necessary to avoid race conditions on shared variables.

The general approach is to assign a distribution function to each shared variable which maps elements of that variable to particular processors. This mapping will induce constraints on the scheduling of parallel loops that access these distributed variables. The goal will be to reduce global memory traffic as much as possible while sacrificing only as much parallelism as is necessary. After the basic problem is modeled, I will examine the effects of various reordering transformations on the solution to the array distribu-
Affine: \[ \delta_A(i) = p_j \ \forall i : a \cdot i + b \equiv_n j \]
Regular: \[ \delta_A(i) = p_j \ \forall i : a \cdot i + b \equiv_n j \]
where \( a \neq 0 \) and \( a \) and \( n \) are relatively prime.
Modulo: \[ \delta_A(i) = p_j \ \forall i : i + k \equiv_n j \]
Global: \[ \delta_A(i) = \top \ \forall i \]
Serial: \[ \delta_A(i) = p_j \ \forall i \]

Figure 25. Examples of Distribution Functions for array \( A \).

tion problem and how to integrate array distribution into the parallel code generator of Chapter 2.

5.2 Distribution Functions, Transfer Functions and Loop Schedules

Let \( P = \{p_0, \ldots, p_{n-1}\} \) be a set of processors. It is assumed here that each processor has an associated private memory, and so both the computational unit and the memory module will be referred to as a “processor”. The elements of \( P \) are thus also memory elements. Let \( A(1..U_1, \ldots, 1..U_k) \) be a shared array variable. The elements of \( A \) may themselves be arrays, in which case each element of \( A \) will be distributed as a unit. A distribution function for \( A \) is a mapping \( \delta_A \) from the set of elements of \( A \) into the set \( M = P \cup \{\top, \bot\} \) of memory elements. If an element is mapped to \( \bot \), that element’s location is undefined. If \( \delta_A(i_1, \ldots, i_k) = p_i \), element \( A(i_1, \ldots, i_k) \) is allocated to the local memory associated with processor \( p_i \). If \( \delta_A(i_1, \ldots, i_k) = \top \), element \( A(i_1, \ldots, i_k) \) is left in global shared memory. Some simple distribution functions for the case where \( A \) is one dimensional are shown in Figure 25. A distribution function that does not map any elements to global memory will be referred to simply as a distribution.

A distribution function is assumed to be constant over an entire live range of a variable. The output of the parallel code generator is a control flow graph, the nodes of which are collections of serial blocks and parallel DO’s. Each such collection will be called a block. A live range is any collection of these blocks. If no satisfactory distribution exists for an entire live range, the live range may be broken into two or more smaller live ranges. At the boundaries between live ranges, transfer functions are used to move elements of the shared variables between memory elements so that the different distribution functions are correctly implemented in the two live ranges. On global memory machines such as the RP³, transfer functions could be implemented via synchronized stores to and loads from
global memory. On message passing machines, transfer functions represent an exchange of messages. Transfer functions are completely defined by the distribution functions of the two live ranges and so will not be explicitly described.

The parallel DO construct described so far places no restrictions on which processor executes a particular iteration: scheduling of iterations on processors is done non-deterministically at execution time. This basic construct can be extended by associating with each loop a relation that specifies which processors may execute each iteration. If \( I_L \) is the set of iterations associated with a loop \( L \), then a schedule, \( S_L \), for \( L \) is a subset of \( P \times I_L \). Such a schedule specifies:

\[ \text{Processor } p_k \text{ can execute iteration } i_j \iff \langle p_k, i_j \rangle \in S_L \]

The basic self-scheduling parallel DO loop has the full set \( P \times I_L \) for a schedule, indicating that any processor may execute any iteration of the loop.

A schedule \( S_L \) is consistent if for all \( i \in I \) there exists \( p \in P \) such that \( \langle p, i \rangle \in S_L \). A consistent schedule is one in which every iteration can be executed by some processor. A schedule \( S_L \) is complete if for every iteration \( i \) there exists a unique processor \( p \) such that \( \langle p, i \rangle \in S_L \). If \( S_L \) is a complete schedule, then for each processor, \( p_i \), define \( \#p_i = \|\{ i \mid \langle p, i \rangle \in S_L \}\| \), the number of iterations processor \( p_i \) will execute. A complete schedule is \( n \)-balanced if \( |\#p_i - \#p_j| \leq n \) for all processors \( p_i \) and \( p_j \). A complete schedule is balanced if it is 1-balanced.

Since a program specifies which array locations are accessed by a particular iteration of a loop, distributing a shared array will induce a schedule on a loop. The schedule requires that only the processor which has a particular element of the shared variable in its local memory may execute any iteration that accesses that element. For example, if for the loop:

\[
\begin{align*}
\text{DO } & I = 1, N \\
& X(I) = X(I) + S \cdot Y(I) \\
\text{ENDDO}
\end{align*}
\]

we specify a distribution of \( X \) by the modulo distribution:

\[ \delta_X(i) = p_j \text{ where } i \equiv_n j \]
then processor $j$ can execute only those iterations that access the elements $X(j), X(j + n), X(j + 2n), \ldots$ and so we get the induced schedule:

$$S_{\delta_X} = \{(p_k, i) \mid \text{if iteration i accesses } X(l) \text{ and } \delta_X(l) = p_j \text{ then } p_j = p_k \}$$

$$= \{(p_j, i) \mid i \equiv_n j\}$$

The induced schedule allows an iteration that does not access a distributed variable to be executed by any processor. In this example, every iteration accesses the shared variable $X$, and so only a single processor is allowed to execute any particular iteration of $L$ under the schedule induced by $\delta_X$ (i.e., $S_{\delta_X}$ is complete). Any schedule that is a subset of $S_{\delta_X}$ is said to satisfy $\delta_X$.

Observe that if a distribution function does not evenly distribute the elements of a shared variable, then it may induce a schedule that reduces machine efficiency. For instance, a distribution that maps all odd numbered elements to one processor and all even numbered elements to another, may induce a schedule such that only those two processors will be able to work on a particular loop. The rest of the processors may be idle until that loop finishes.

It is also possible that a distribution function will induce a schedule that is not consistent: some iteration cannot be executed because no processor has access to all of the elements it needs. For instance in the loop:

```
DOALL I = 1,N
    A(I) = (B(I) + B(I+1))/2
ENDDO
```

any regular distribution\footnote{As defined in Figure 25.} of $B$ would induce an inconsistent schedule. In fact, no iteration of the loop could be executed. This is not to say that $B$ cannot be distributed, but the only choices are to leave $B$ in global memory or map every element to a single processor which effectively serializes the loop. Later, distribution functions will be allowed to be multi-valued to cleanly handle this problem.

5.3 Data Dependences and Array Distribution

Important first questions are when can a variable be distributed over a single parallel loop, and, the generalization of that question, when can a set of variables be simultaneously
distributed over some live range. We assume that a parallel loop is one without any carried true, output or anti-dependences.

**Definition 5.1** A variable $A$ is distributable over a parallel loop $L$ if there exists a distribution of $A$ that is satisfied by a balanced schedule for $L$.

This definition is somewhat restrictive since situations may arise in which selecting a distribution that is not 1-balanced is preferable to the alternatives, but it provides a reasonable basis for discussion. Note also that the existence of balanced distributions is affected by the number of processors: a balanced distribution may exist for a two processor system but not a 1024 processor system. We will assume that we are generating code for a machine with a sufficient number of processors so that certain non-scalable distributions are not considered balanced.

Data dependence analysis can be used to determine when a variable is distributable. In the previous example, while the variable $B$ does not generate any loop carried true, output, or anti-dependences, and hence the loop is a valid parallel DO loop, it does generate a loop carried input dependence. This input dependence indicates that one or more elements of $B$ are used on different iterations. If multiple iterations access the same element of $B$, then all of those iterations will have to be executed by the same processor.

**Proposition 5.1** If a shared variable $A(\ast)$ that is used in a parallel loop $L$, does not generate any input dependences carried by that loop, then there exists a distribution for that variable that is satisfied by a balanced schedule.

*Proof:* All access to a particular memory location of $A$ must be made by the same processor, otherwise a loop carried dependence would exist. Since the loop is parallel, no loop carried true, anti-, or output dependences exists and by hypothesis, no input dependences exist. Hence, every iteration must access a different set of the memory locations associated with $A$. If $I$ is the set of iterations of the loop\(^2\) and $n$ is the number of processors, define the schedule $S_L$ by

\[ (p_j, i) \in S_L \iff j \equiv_n i \]

---

\(^2\) Assume $L$ is normalized to run from 1 to some upper bound in steps of 1 so $I$ is $\{1..N\}$ for some $N$. 
This schedule defines a distribution:

\[ \delta_A(i) = j \text{ if } (p_j, l) \in S_L \text{ and iteration } l \text{ accesses } A(i) \]

Clearly \( S_L \) is balanced and satisfies \( \delta_A \), and so \( A \) is distributable over \( L \).

\textit{End of Proof.}

**Corollary** Any variable used in a parallel loop \( L \) that does not generate a loop carried input dependence is distributable over \( L \).

If the number of processors is sufficiently large, then the converse of this proposition will hold: if \( A \) generates a loop carried input dependence, then at least two iterations will be scheduled on the same processor. If the number of iterations is less than the number of processors, some processor will not have any iterations scheduled for it. Hence no distribution can be 1-balanced since some processors will be scheduled for at least two iterations and some processors will not be scheduled for any iterations.

The converse of this proposition is in general false when the number of processors is small. Consider the following loop.

\[
\begin{align*}
\text{DOALL I = } & 1 \text{,N} \\
& A(I) = (B(I-1) + B(I+1))/2 \\
\text{ENDDO}
\end{align*}
\]

This is similar to the previous example except that the threshold of the input dependences is now 2. We can divide \( B \) into even and odd indexed elements, and construct a distribution function that allows two processors to execute this loop. Note that if our machine has only two processors, this is a balanced distribution. This is an example of the non-scalable distributions mentioned earlier. We assume that the number of processors is effectively unbounded so that this special case does not satisfy the definition of distributable. We have argued:

**Theorem 5.1** If the number of processors is sufficiently large, a variable is distributable over a parallel loop if and only if it does not generate a dependence carried by that loop.

Note that in the proof of the last proposition, a schedule was selected and from that schedule a balanced distribution was induced. This is a general technique that will be used again.

If multiple variables are distributed, then each distribution will impose constraints on which processor a particular iteration can be executed on.
Lemma 5.1 If A and B are two shared variables and $\delta_A$ and $\delta_B$ are distributions over a parallel loop L for them respectively, then the schedule $S_{\delta_A,B} = S_{\delta_A} \cap S_{\delta_B}$ is the largest schedule that satisfies both $\delta_A$ and $\delta_B$.

Proof: Immediate from the definitions. Largest means that any schedule that satisfies both $\delta_A$ and $\delta_B$ will be contained in $S_{\delta_A,B}$.

Note that $S_{\delta_A,B}$ may not be consistent, but if we already know that L is parallel and each A and B is distributable, then at least some distributions of A and B will induce a balanced schedule.

Proposition 5.2 For any parallel loop, we can find distributions for all of the distributable shared variables so that are simultaneously satisfied by a balanced schedule.

Proof: Essentially the same as for Proposition 5.1

The implication of this proposition and the previous corollary is that for a given parallel loop, all variables not generating loop carried input dependences can be simultaneously distributed, and the schedule induced by these distributions is satisfied by a balanced schedule.

The next step is determining when a set of variables can be simultaneously distributed over a region that is larger than a single loop. The definition of distributable can be extended to serial regions by treating them as degenerate loops with only one iteration. Note that for a serial region, any consistent schedule is trivially balanced and a distribution induces a consistent schedule only if every memory location accessed by the serial region is contained in one particular local memory or in global memory.

Definition 5.2 A set of variables V is distributable over a set of blocks B if there exists distribution functions $\delta_v$ for each $v \in V$ and there exist balanced schedules $S_b$ for each $b \in B$ such that each $S_b$ satisfies every $\delta_v$.

Note that distribution functions are constant over the set of blocks but schedules are defined independently for each block in the set. Some examples are shown in Figure 26. For the code fragment shown in Figure 26(a), both A and B can be distributed via the distributions

$$\delta_A(i) = p_j \text{ where } i + 1 \equiv_n j$$

$$\delta_B(i) = p_j \text{ where } i \equiv_n j$$
DOALL 10 I = 1,N
A(I) =
ENDDO
BARRIER
DOALL 20 I = 1,N
B(I) = A(I+1)
ENDDO
BARRIER
DOALL 20 I = 1,N
C(I) = B(I)
ENDDO

DOALL 10 I = 1,N
A(I) =
ENDDO
BARRIER
DOALL 20 I = 1,N
B(I) = A(I+1)
ENDDO
BARRIER
DOALL 30 I = 1,N
C(I) = B(I) + A(I)
ENDDO

(a) (b)

Figure 26. Distribution Examples

These distributions clearly induce balanced schedules on the first and third loops. Also note that independently, each of the above distributions induces a balanced schedule on the middle loop, and that they induce the same balanced schedule. Thus the set of variables \{A, B\} is distributable over the blocks shown in Figure 26(a).

However, this set of variables is not distributable over the code fragment shown in Figure 26(b). Observe that any consistent schedule for the middle loop must have \(B(I)\) and \(A(I + 1)\) in the same local memory, but the third loop implies that \(B(I)\) and \(A(I)\) must be in the same local memory. Hence \(A(I)\) and \(A(I + 1)\) must be in the same local memory for all \(I\) in the range of 1 to \(N\). An analogous argument indicates that \(B(I)\) must be in the same local memory as \(B(I - 1)\). Clearly no balanced schedule exists if both \(A\) and \(B\) are distributed.

In the above examples, both \(A\) and \(B\) are distributable over each parallel loop, and in fact each is independently distributable over all three loops in both examples. The distribution functions given for the loops in Figure 26(a) are strongly related to the idea of loop alignment: the basic schedule maps iteration \(i\) to processor \(p_i\), this schedule is shifted for the second loop so that the same processor executes iteration \(i - 1\) and can reuse the value of \(A(I)\) it just computed. This scheme breaks down for the fragment shown in Figure 26(b) when we reach the last loop and find that \(A(I)\) and \(B(I)\) are computed by different processors but are needed on the same processor.
The code in Figure 26(b) illustrates the effects of an alignment conflict on array distribution. The next theorem establishes the relationship between alignment conflicts and distributability. Alignment conflicts do not precisely correspond to non-distributivity because "simultaneous" solutions may not exist, i.e., there may not be some particular set of memory locations that cannot be distributed. One easy example involves loop reversal:

```
DOALL I = 1, N
    A(I) =
ENDDO
BARRIER
DOALL I = 1, N
    ... = A(N+1-I)
ENDDO
```

Despite the presence of the alignment conflict (generated by an inconsistent dependence), any balanced schedule for the first loop can be used to construct a balanced schedule for the second, therefore $A$ is distributable over both loops.

The question of distributability for finite processors, even for the case in which all array references are very simple, is at least NP-hard\(^3\). This can be demonstrated by a straight-forward reduction from the colorability problem. An instance of $k$-colorability consists of an undirected graph $G = (V, E)$ and a positive integer $k$. The problem has a solution if there is a $k$-coloring, a function $c$ from $V$ into $\{1..k\}$, such that for each edge $e = \langle v_i, v_j \rangle$, $v_i$ and $v_j$ are mapped to different elements. The coloring problem can be mapped into a distribution problem: let $T$ be a array of $|V|$ elements and for each edge $e = \langle v_i, v_j \rangle$ construct a loop of the form

```
DO I = i, j, j-i
    T(I) =
ENDDO
```

Note that each such loop has exactly two iterations, one accessing $T(i)$ and the other $T(j)$.

It can now be shown that $T$ is distributable over all of these loops for a $k$ processor machine if and only if $G$ is $k$-colorable. If $T$ is distributable, then there exists a distribution $\delta_T$ that is satisfied by a balanced schedule for every loop. A balanced schedule for each loop must have each iteration executed by a different processor. Construct a coloring $c$ by

\(^3\) In some sense, it is undecidable since detection of parallelism is undecidable[Bern 66].
$c(v_i) = \delta_T(i)$. For any edge $e = (v_i, v_j)$, there exists a loop that has a balanced schedule, hence $c(v_i) = \delta_T(i) \neq \delta_T(j) = c(v_j)$ and therefore $c$ is a valid $k$-coloring of $G$.

Conversely, if $c$ is a valid $k$-coloring of $G$, then define a distribution $\delta_T(i) = c(v_i)$. For any loop in the collection, there corresponds an edge $e = (v_i, v_j)$ such that $c(v_i) \neq c(v_j)$, hence the two elements of $T$ accessed by the loop are mapped to different processors and therefore the loop has a balanced schedule that satisfies $T$. This result depends on a fixed, finite number of processors being specified as part of the problem. If an unlimited number of processors are available, then this result no longer holds.

In scientific programs, it is reasonable to take alignment conflicts as equivalent to non-distributivity. Handling situations such as the loop reversal example above can be considered as optimizations to improve array distribution in special cases. Further, under this assumption very simple distributions generate balanced schedules, as is shown in the next theorem.

**Theorem 5.2** Let $V$ and $B$ be as Definition 5.2. If there are no alignment conflicts in the dependence graph restricted to $V$ and $B$, then $V$ is distributable over $B$.

**Proof:** Consider the graph formed by treating each loop in $B$ as a node and each inter-loop dependence as an edge. This graph has no alignment conflicts and by Proposition 3.3.4, it has an exact alignment $\sigma$. For each loop in $l \in B$, define the schedule

$$S_l = \{(i, p_j) \mid i + \sigma(l) \equiv_n j\}$$

where $n$ is the number of processors available. From these schedules we can define a distribution for each variable $v \in V$ by

$$\delta_v(i) = \begin{cases} 
 j & \text{if there exists a loop } l \text{ and an iteration } k \text{ of } l \text{ that accesses } v(i) \text{ and } (k, p_j) \in S_l \\
\bot & \text{otherwise}
\end{cases}$$

If $\delta_v$ is not well defined for some $i$, then there exist loops $l_1$ and $l_2$ and iterations $k_1$ and $k_2$ such that iteration $k_1$ of $l_1$ and $k_2$ of $l_2$ both access $v(i)$ and $(k_1, p_{j_1}) \in S_{l_1}$ and $(k_2, p_{j_2}) \in S_{l_2}$ where $j_1 \neq j_2$. There must be a data dependence between the two loops: let $d$ be that data dependence. Since $\sigma$ is an exact alignment, we know $\sigma(l_1) - \sigma(l_2) = weight(d) = k_2 - k_1$ or $\sigma(l_2) - \sigma(l_1) = weight(d) = k_1 - k_2$ depending on the direction of $d$. In either case, we have $\sigma(l_1) + k_1 = \sigma(l_2) + k_2$, hence $\sigma(l_1) + k_1 \equiv_n \sigma(l_2) + k_2$ but
\( \sigma(l_1) + k_1 \equiv_n j_1 \) and \( \sigma(l_2) + k_2 \equiv_n j_2 \) contradicting the assumption that \( j_1 \neq j_2 \). Therefore each \( \delta_v \) is well defined.

Each \( \delta_v \) is clearly satisfied by the balanced schedule \( S_l \) for every \( l \in B \) and therefore \( V \) is distributable over \( B \).

**End of Proof.**

The last result of this section extends the previous theorem to apply to an entire subroutine.

**Definition 5.3** A cover is a collection of live ranges such that for every variable, every loop is contained in exactly one live range of that variable. A cover is distributable if there exists distribution functions \( \delta_v \) for each live range \( r \) such that every for loop \( l \) in the subroutine, there exists a balanced schedule for \( l \) that satisfies every \( \delta_v \) for every live range \( r \) that contains \( l \).

A dependence graph restricted to a cover consists of all dependences such that both endpoints are in the same live range of the variable that generates the dependence.

**Theorem 5.3** Let \( C \) be a cover for a subroutine. If the dependence graph restricted to \( C \) has no alignment conflicts, then \( C \) is distributable.

**Proof:** Let \( L \) be the loops in the subroutine. As in the proof of Theorem 5.2, let \( \sigma \) be an exact alignment for the graph formed by collapsing the nodes in the dependence graph corresponding to statements in a single block to a single node. Also define schedules \( S_l \) and distributions \( \delta_v \) as in the proof of Theorem 5.2.

Again, if \( \delta_v(i) \) is not well defined for some variable \( v \) and element \( i \) of \( v \), then there exist two blocks, \( l_1 \) and \( l_2 \), and iterations \( k_1 \) and \( k_2 \) such that \( l_1 \) access \( v(i) \) on iteration \( k_1 \) and \( l_2 \) access \( v(i) \) on iteration \( k_2 \) and \( k_1 + \sigma(l_1) \neq_n k_2 + \sigma(l_2) \).

Since \( l_1 \) and \( l_2 \) are in the same live region, all edges between them are in the graph for which \( \sigma \) is an exact alignment. Therefore, there exists a data dependence from \( l_1 \) to \( l_2 \) with weight \( k_2 - k_1 \) or \( k_1 - k_2 \). Without loss of generality, assume this dependence is directed from \( l_1 \) to \( l_2 \), the other case is analogous. The weight on the dependence edge is \( k_2 - k_1 \), and, since \( \sigma \) is exact:

\[
\sigma(l_1) - \sigma(l_2) = k_2 - k_1
\]

which contradicts the selection of \( l_1 \) and \( l_2 \).

**End of Proof.**
5.4 An Algorithm

This section incorporates the test developed in the previous section into the algorithm of Chapter 2. Recall that that algorithm is applied to a collection of statements; the basic steps are: separate statements in the strongly connected region of the dependence graph; recurse on regions carrying recurrences; then recombine regions to minimize barrier synchronization points. Now we must also decide how to distribute arrays and, in particular, when an array can be distributed over a strongly connected region and when two strongly connected regions should be in different live regions. The algorithm will initially be presented without reference to the algorithm in Chapter 2, then the implications of array distribution on parallel code generation will be discussed.

An outline of the distribution algorithm is given in Figure 27. Note that the particular order in which strongly connected regions are visited is determined by the greedy algorithm of Chapter 2, (using reverse topological ordering as discussed in section 3.5). The rest of this section develops the test for distributivity over a strongly connected region, and discusses whether a given strongly connected region can be fused with a live region and maintain distributivity. The test for distributivity is based on dependences that have both endpoints in the same live region of the variable that generates the dependence. Hence, we are concerned with a graph \( G \) that consists of statements in \( S \) and dependences with both endpoints in the same live region. As live regions are fused, new edges will be added to this graph; the distribution algorithm merges live regions together when doing so does not create an alignment conflict in \( G \).

Initially, every node (strongly connected region) is in a separate live region for every variable. Hence \( G = (S, \emptyset) \) and trivially has no alignment conflicts. As live regions are combined, edges are added to \( G \). Each edge generates an alignment requirement between its endpoints. The (undirected) connected components of \( G \) represent sets of statements that have fixed relative alignment. When an edge is added to \( G \), if its endpoints were in different connected components, that edge cannot create an alignment conflict: it simply generates new relative alignment requirements between the nodes in one connected component and the other. On the other hand, if both endpoints are already in one connected component, adding the edge will create an alignment conflict if and only if the weight of the edge differs from the relative alignment of the two endpoints.
procedure distribute($S, k$
break $S$ into strongly connected regions
foreach strongly connected region $r$ that carries a recurrence do
distribute($r, k + 1$)
determine which variables are distributable over each strongly connected region
foreach strongly connected region $r$ in reverse topological order do
foreach distributable variable $v$ accessed by $r$ in order of decreasing cost do
fuse $r$ with current live range of $v$ if no alignment conflicts arises, otherwise
start a new live region for $v$

Figure 27. Sketch of Distribution Algorithm

The test for alignment conflicts as edges are added is an application for the fast
disjoint set union-find algorithm, as described in section 3.2.4. This algorithm maintains
disjoint sets as trees. Each element contains a pointer to its father in the tree. Set union
is performed by making the root node of one set a child of the root node of the other.
Assume that every edge is labeled so that if the root of a set were to be given an alignment
value of 0, an exact alignment for the set would be defined by assigning to each node the
sum of the labels along the path from that node to the root.

We maintain disjoint sets of statements in $S$ that correspond to connected components
of $G$. As sets are unioned together, the weight of the edge that caused them to be merged
is used to label the new edge introduced to the forest of trees representing the disjoint
sets. It is easy to see that the relationship between exact alignments and tree edge labels
is thus preserved. Note that if path compression is used during the find operations, the
labels on the edges will need to be adjusted.

The basic step in fusing two live regions would look like

distributed
\begin{align*}
\text{foreach edge } e & = (v, w) \text{ between the two regions do} \\
(A, k) & = \text{find}(v) \\
(B, l) & = \text{find}(w) \\
\text{if } A \neq B \text{ then union}(A, B, \text{weight}(e) - k + l) \\
\text{else if } \text{weight}(e) & \neq k - l \\
\text{then found alignment conflict, abort fusion}
\end{align*}

where find($v$) returns the set containing node $v$ as well as the relative alignment of that
node from the root of the set, and union($A, B, x$) merges sets $A$ and $B$ with relative offset
Figure 28. Effects of merge for edge $e = (v, w)$: $z = \text{weight}(e) - k + l$

$x$. The diagram in Figure 28 illustrates the result of the union operation used in the above loop.

A major step in the algorithm is determining when a variable is distributable over a particular strongly connected region. If it is not, that strongly connected region will be a live region for that variable and no other live regions will be fused with it. If the strongly connected region does not carry a recurrence at level $k$, a variable is distributable if it does not generate an input dependence carried at level $k$, and is non-distributable otherwise.

If the strongly connected $r$ region does carry a recurrence, a variable $v$ is distributable over that region if and only if after the recursive call $\text{distribute}(r, k + 1)$, adding the level $k$ dependences generated by $v$ to graph $G$ does not create an alignment conflict.

It may be that every variable is individually distributable, but they are not all simultaneously distributable. Hence, variables should be checked for distributivity in order of decreasing cost, where cost is taken to be proportional to the amount of storage accessed by the strongly connected region via this variable. This will give a preference to distributing large arrays over smaller arrays. Likewise, when attempting to fuse live regions, variables again should be processed in the order of decreasing amounts of storage accessed.

The test for whether a particular strongly connected region can be added to the
current live range of a variable entails finding the relative alignment values of all of the edges between the strongly connected region to a statement in the live region carried by the variable. Once these are found, they can be compared with the weights on the edge to determine if an alignment conflict would be created. For example, in the loops

\[
\begin{align*}
\text{DO } & I = 1,N \\
& B(I) = A(I) \ldots \\
\text{ENDDO} \\
\text{DO } & I = 1,N \\
& \ldots = A(I) + B(I-1) \\
\text{ENDDO}
\end{align*}
\]

Initially, each loop corresponds to a live region for each variable. The first node processed corresponds to the last loop\(^4\), and no alignment conflicts are found. When the first loop is processed, assume \(B\) is processed before \(A\). The edge carried by \(B\) will generate an alignment constraint of 1 between the two loops. When the edge generated by \(A\) is examined, a new alignment requirement of 0 is found between the statements, contradicting the relative alignment of 1 already specified. This alignment conflict will force the two uses of \(A\) to be in different live regions.

When all nodes have been processed, the roots of the sets can be assigned an alignment value of 0, and the absolute alignment value of each node then becomes equal to its relative alignment value with respect to the root of the set that contains it.

5.5 Impact on Parallel Code Generator

The parallel code generator developed in Chapter 2 began with a notion of when explicit synchronization was needed between two statements. This notion was based on the possibility that two or more processors may access a particular location simultaneously. When all data is global, every processor can access every word of data, but when an array is distributed, only a single processor can access the data allocated to a particular local memory. The implication is that data dependences carried by distributed arrays need not be explicitly synchronized.

For example, consider the following two loops:

\(^4\) Recall the greedy algorithm as modified in section 3.5 visits node in reverse topological order.
DO I = 1,N
   A(I) =
ENDDO
DO I = 1,N
   ... = A(I-1)
ENDDO

If $A$ is distributed across both loops, there is no need to insert a barrier synchronization between these loops: all writes to locations in $A$ will occur before any reads since the same processor does both reads and writes to any particular location. Hence, when we integrate the distribution machinery of the previous section into the algorithm of Chapter 2, the routine check.sons($v$) should be modified to examine only those dependences outgoing from $v$ that are either carried by non-distributed variables or that go to nodes in different live regions.

Code replication, as presented in Chapter 3, provides a tool for breaking alignment conflicts. An analogue of code replication, data replication, can also be used to improve distributivity of arrays. Note that code replication involves two parts: replicating the statements which generate the values, and renaming variables to separate the various instances. Data replication involves only the renaming of variables. Instead of replicating the computation, the value is simply copied over from the original variable in to the new variable. For example, in the loops

DO I = 1,N
   B(I) =
ENDDO
DO I = 1,N
   ... = B(I) + B(I-1)
ENDDO

The use of $B$ in the second loop is not distributable due to the loop carried input dependence. Further, there is an alignment conflict generated by all of the accesses to $B$. An application of data replication would create a new variable $B1$ and use it to hold a copy of $B$, as in:
DO I = 1,N
    B(I) =
    B1(I) = B(I)
ENDDO
DO I = 1,N
    ... = B(I) + B1(I-1)
ENDDO

Note that a barrier synchronization point will be needed between these loops and only one of \( B \) and \( B1 \) can be distributed, but at least half of the loads from global memory of \( B \) are avoided. This transformation does not really imply addition storage for \( B1 \): the global store for \( B \) is not used while \( B1 \) is live and so can be used to hold the values of \( B1 \). This can be viewed as a multi-valued distribution of \( B \) for the second loop.

Another variant on data/code replication has all processors execute a statement, simultaneously updating local copies of a variable. An example is:

\[
T = 1/A(K,K)
\]
\[
DOALL I = K+1,N
    A(I,K) = A(I,K)/T
ENDDO
\]

If \( T \) is a local variable and every processor executes the statement assigning into \( T \), then no synchronization is needed between that statement and the following loop. This will be profitable (for this example) if the cost of barrier synchronization point is more than the cost of a global memory fetch and a divide operation. Again, this can be viewed as a multi-valued distribution of \( T \).

5.6 Summary

This chapter has examined the problem of distributing arrays to local memories while maintaining balanced assignment of loop iterations to processors. A dependence-based test using the concept of alignment conflict developed in Chapter 3 is presented to determine when effective distribution is possible. This test is used to augment the algorithm of Chapter 2 with a mechanism for performing array distribution.
Chapter 6:
Conclusions

The goal of this dissertation was to provide a complete framework for automatically translating sequential FORTRAN programs into parallel programs. The algorithm presented in Chapter 2 is suitable for parallelizing complete subroutines: explicit extensions of the algorithm to use loop interchange, loop alignment and code replication have been given and loop fusion is implicit. Techniques were developed in Chapter 4 to improve the effectiveness of the algorithm in the presence of call sites. With these techniques, program modularity should not inhibit parallelization. The other significant new hardware common to multiprocessor computers is local private memory associated with each processor. Chapter 5 develops techniques to use that memory by distributing arrays across the processors.

Much of the work discussed in this dissertation has been implemented in the Parallel Fortran Converter (PFC) developed at Rice University with the support of IBM. This final chapter discusses the status of this implementation, future topics of research in this area, and ramifications of these techniques on programming and program development.

6.1 The Implementation

PFC is an aggressive, restructuring, vectorizing, source-to-source translator. All of the underlying mechanics and data structures of that system are needed by the parallelizing system, PFCPLUS. In particular, the frontend (parsing, interprocedural analysis, normalization, global data flow, induction variable substitution, scalar optimizations) is used without modification. The phase that builds the data dependence graph is also used without modification except for additional information computed to support loop interchange. Code to compute strongly connected regions also needed only minor modification.

The current system implements the algorithm of Chapter 2 including loop interchange. Due to restrictions reasonable for a vectorizer, the algorithm is currently only applied to outermost DO loops rather than entire subroutines. This deficiency will soon be eliminated. It also partially implements loop alignment and code replication from Chapter 3: these transformations are performed only on the innermost loop due to insufficient information. New code added to implement this system was less than 3500 lines (PL/I with preprocessor support for debugging and linked list utilities).

Initial experience is encouraging, but the current system is still too limited, so no data will be reported here.
6.2 Future Work

There are many topics that need further examination. Some are particular to the algorithms presented here and some are more general to the area of automatic parallelization.

A model of synchronization was selected that employs only barrier synchronization. This model was selected because of its simplicity and the ease with which it can be discussed and reasoned about. Alternative models exist; an important one involves explicit processor-processor synchronization to preserve a specific dependence. This model is used to realize the DOACROSS on the Alliant FX/8 and could be implemented via the "asynchronous variables" on the Denelcor HEP. If the cost of synchronization for specific dependences can be quantified, this approach can be incorporated into the algorithm of Chapter 2 as an alternative to barrier synchronization and code replication.

A global "planning" phase should be incorporated into the parallelization process. This planning phase would be used to locate regions in the program where extra effort should be applied during optimization and parallelization. This extra effort might use in-line expansion more aggressively or use more complicated but more precise algorithms to solve some of the NP-complete optimizations problems already identified as useful. The extra effort might also employ exhaustive pattern matching for special case recurrences and loop constructs that could be replaced with faster algorithms.

The support code for vectorization is almost identical as for parallelization, and many machines will have multiple processors each with vector capabilities. For either short vectors or few processors, there are interesting trade-offs to be made between running a particular loop as a DOALL, serial vector code, or some form of concurrent vector code. Factors that affect this choice are: costs of synchronizing processors together, vector start up times, vector register lengths, and the expected lengths of vectors. The complexity of control flow is also a factor, since branching in a loop only affects load balance for the DOALL but may generate complicated masks for vector operations.

The difficulties of determining whether two loops are fusible after arbitrary loop interchange is a problem for the algorithm of Chapter 2. Its solution is to only allow fusion of loops with the same original level in the outermost position (i.e., both have the original outer loop or both have the original inner loop in the outer most position). This is probably reasonable for loops that were created via loop distribution, but may not be
effective for loops from independent loop nests. While dependence testing could be done for each proposed loop interchange, this would probably be prohibitively expensive.

Chapter 3 examines other applications of loop alignment. More work can be done to complete the ideas discussed there and look for other applications. Loop alignment viewed as a general tool to change the distance vector of a dependence could also find applications in vectorizations, vector register assignment, and sectioning.

Loop alignment is not effective for inconsistent dependences since not all memory uses will line up. What techniques can remove these dependences? Can code replication be used, or perhaps a non-linear form of loop alignment?

Where the techniques of Chapter 4 fail to indicate a loop containing a call site can be parallelized, in-line expansion might still be attempted in the hope that loop distribution can be done to isolate the serial code. What tests could be developed, based on summary information, that could predict when in-line expansion and loop distribution will yield one or more parallel regions?

There is a strong relationship between the “general RSD” data flow problem and the interprocedural maximum value problem: find for each formal parameter a (perhaps symbolic) value that, for invocations of the subroutine, is strictly greater than the value of the formal parameter. In both cases, the semi-lattices have unbounded chains and functions associated with call sites can generate infinite ascending chains.

The approach to array distribution developed in Chapter 5 is unsatisfying for a number of reasons. No complexity results are known for the final problem of minimizing data transfer costs based on the alignment conflict test. While it seems very likely the problem is NP-complete, no proof is known. The algorithm presented in Chapter 5 was developed to mate well with the algorithm of Chapter 2, and it is not clear how effective other heuristic approaches would compare. One possibility is a prioritized algorithm similar to the priority coloring algorithms for register allocations [ChHe 84]. The problem of load balancing is not addressed by the algorithm of Chapter 5. Finally, there is no empirical data to determine if the alignment conflict test will be effective for real programs.

The interaction between array distribution and parallel code generation needs further exploration. The decision to distribute arrays will imply that certain dependences, even though they are loop carried, do not need synchronization. Where there are choices to be made (such as where to break a live region into smaller live regions), how does this choice affect the quality of the code produced by the parallel code generator?

Alignment conflicts that arose during array distribution simply forced a break between live regions. As alternatives there are analogues to code replication for data and the possibility of "multi" value distribution functions. It is unknown whether these are needed and how they would impact the distribution algorithm of Chapter 5.
6.3 Ramifications

All aspects of programming for multiprocessors is harder than for scalar processors. Algorithm development, program design and program development (especially debugging) are complicated by the need for synchronization and the problem of non-determinism. In addition, the aggressive nature of program compilation implicit throughout this dissertation further complicates these tasks.

The compiler should be viewed as a tool in the program development process. The PTOOL project at Rice University [ABKP 86] is a step in the direction of using the power of the compiler to analyze programs during program development. Currently, information is only fed back from PFC to the programmer, giving the programmer access to the results of PFC's dependence analysis. Information can flow in the other direction as well: the programmer can provide information that PFC is unable to determine for itself. This could include information about the values of loop bounds, whether certain arrays used for indirect indexing are permutations, and information about unavailable entry points.

Interprocedural flow of information is contrary to the independent compilation spirit of FORTRAN. Modifications in one subroutine, even when they do not change the subroutine linkage, may force recompilation of other routines that have used interprocedural information based on the previous version. Intelligent recompilation strategies have been addressed already [CoKT 86] but will need to be reconsidered as the nature of interprocedural information changes. The R system programming environment [HoKe 84] was designed as vehicle to support experiments in interprocedural analysis and its impact on program development.

The VM/EPEX [SDNP 85] language does not currently allow parallel constructs to be nested, hence two versions of certain subroutines may be required: one parallelized and one in original serial form. Even for languages which allow arbitrary nesting of parallel constructs, it may be desirable to have both parallel and serial versions of subroutines, since the parallel versions will suffer some performance degradation compared to the serial version when executed by single processor. This is a special case of a more general linkage tailoring and subroutine cloning problem [Coop 83].

Debugging of parallel programs will be more difficult than that of their serial counterparts due to the presence of non-deterministic scheduling of concurrent operations. If serial programs are parallelized using correct automatic techniques, as described here,
this non-determinism is guaranteed not to affect the results. However, if the user provides
additional information during the compilation process, or modifies the parallel program
produced by a source-to-source translator, this assurance is lost. In the former case, the
compiler could generate defensive code to verify at run time the assertions the programmer
has made, but this may be impossible or impractical.

The effectiveness of interprocedural information and therefore amount of program
restructuring done to enhance parallelism will be severely hampered when library routines
are available only in object form. If libraries were available in the intermediate form of thecompiler, they could be treated like user subroutines and optimized within the context of
the complete the program.

Programmers currently need only be concerned with source programs, object mod-
ules, object libraries and final executables. Compilation dependences are explicit in the
form of “include” files. Source level interactive debugging of unoptimized code has be
handled by embedding symbol table information and simple maps of statement numbers
to object code locations in the object modules and executables. The amount of inform-
ation retained by aggressive compilation include interprocedural summary information
and compilation dependences derived from using this information. Any extra information
supplied by the user that is not explicit in the source programs must be stored for future
compiles (or requested of the user again). Complicated source to object maps resulting
from restructuring will also need to be maintained to allow source-level interactive debug-
ing. The increase in information retained about the compilation process implies the need
for software to maintain it in a consistent and usable form.
References


[Bane76] U. Banerjee, ‘Data dependence in ordinary programs’, November 1976, Dept. of Computer Science, University of Illinois at Urbana-Champaign, Report 76-837


[BuCy86b] M. Burke and R. Cytron, 'Interprocedural Dependence Analysis and Parallelization (Extended Version)', 7 November 1986, IBM T.J. Watson Research Center, RC 11794 (#55140)

[CaKe86] D. Callahan and K. Kennedy, 'Applications of data dependence analysis to pipelined Processors', October 1986, Dept. Of Computer Science, Rice University, Supercomputer Software Newsletter #10


[FeOt83] J. Ferrante and K. J. Ottenstein, A Program Form Based on Data Dependency in Predicate Regions, Conference Record of the Tenth ACM Symposium on the Principles of Programming Languages, January 1983


[HoKe84] R. Hood and K. Kennedy, 'A Programming Environment for Fortran', June 1984, Dept. Of Computer Science, Rice University, Rice COMP TR84-1


[Trio86] Remi Triolet, 'Interprocedural Analysis Based Program Restructuring With Parafrase', 1986, University of Illinois, Center for Supercomputing Research and Development


