INFORMATION TO USERS

The most advanced technology has been used to photograph and reproduce this manuscript from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

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

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.
Interactive parallelization of numerical scientific programs

Balasundaram, Vasanth, Ph.D.

Rice University, 1989
RICE UNIVERSITY

Interactive Parallelization
of Numerical Scientific Programs

by

Vasanth Baiasundaram

A THESIS SUBMITTED
IN PARTIAL FULFILLMENT OF THE
REQUIREMENTS FOR THE DEGREE

Doctor of Philosophy

APPROVED, THESIS COMMITTEE:

Ken Kennedy, Chairman
Noah Harding Professor of Computer Science

Lori Pollock,
Assistant Professor of Computer Science

Robert Bixby,
Professor of Mathematical Sciences

Houston, Texas
April, 1989
Interactive Parallelization
of Numerical Scientific Programs

Vasanth Balasundaram

Abstract
Parallel computers can provide impressive speedups, but unfortunately such speedups are difficult to realize in actual practice unless the program is written in a way that effectively exploits the parallel hardware. This may require a considerable intellectual effort on the part of the programmer, especially when the programs are large and very complex. Programs of this sort are typically encountered in scientific applications, where the compute-intensive nature of the application makes high execution speeds very critical. This thesis develops a framework that can be used as a basis for the design of an interactive parallel programming tool, that can aid in the development of parallel programs.

A general theory of data dependence testing is presented that allows detection of different granularities of parallelism to be treated in a uniform manner. The parallelism detection process can either be guided by the programmer or performed automatically. Some new parallelism enhancement techniques such as task pipelining are also presented. To deal with programmer specified parallelism, a static analysis technique is developed for identifying potential race conditions in a parallel program. This technique is also used to identify some forms of deadlock.

A parallel programming tool designed using these techniques will help considerably reduce the time and effort that is usually devoted to the development of correct and efficient parallel programs.
Acknowledgments

I would like to thank several people that have helped make this dissertation possible. My thesis advisor Dr. Kennedy for his guidance and support. Lori Pollock and Kathryn McKinley for proof reading the dissertation and making useful suggestions. Andy Boyd, Ulrich Kremer, Jaspal Subhlok and Joe Warren for all the interesting discussions that led to many of the ideas in this thesis. My closest friends, Ulrich Kremer, Suparna Rajaram, Kanneganti Ramaraao, Julie Richards, Sanjay Saigal, Kumar Srikantan, Kavitha Srinivas and Jaspal Subhlok for making life at Rice so enjoyable. The department basketball group for all those fun evenings at the gym. The faculty and students in the Computer Science department. My parents and brother Arvind for being the most wonderful family anybody could hope to have. Finally, I would like to dedicate this work to the memory of my grandfather, who has been the source of immense inspiration in my life.
Contents

Abstract ii
Acknowledgments iii
List of Illustrations viii

1 Introduction

1.1 The nature of numerical scientific programs 2
1.1.1 PIC particle in cell simulation 3
1.1.2 SIMPLE hydrodynamics simulation 3
1.1.3 Seismic interpolation 3
1.2 Parallelism model 4
1.3 Specification of parallelism 4
1.4 Program partitioning for parallelism 6
1.5 Parallel programming tools 9
1.6 Overview of the thesis 10

2 A General Theory of Dependence Testing 12

2.1 An extended definition of data dependence 12
2.2 Summarizing data access in a region 13
2.2.1 Simple sections 14
2.2.2 Intersection and union of simple sections 19
2.3 Formal definition of the Data Access Descriptor 24
2.3.1 Traversal order 25
2.3.2 Reference template 27
2.3.3 Data Access Descriptor for a scalar variable .......... 30
2.4 Determining the Data Access Descriptor of an array accessed within
a loop ............................................. 30
  2.4.1 Dealing with auxiliary induction variables .......... 33
  2.4.2 Dealing with symbolic variables ................... 34
2.5 Translation of Data Access Descriptors ................... 37
  2.5.1 Translation to a surrounding loop ................. 38
  2.5.2 Translation from a subroutine to a call site ....... 38
2.6 Dealing with non-unit step sizes .......................... 46
  2.6.1 Intersection and union of Data Access Descriptors .... 48
2.7 The Data Window revisited .............................. 49
  2.7.1 Intersection and union of Data Windows .......... 49
2.8 Dependence testing using Data Windows .................. 53
  2.8.1 Programmer supplied assertions ................... 54
  2.8.2 Implementing dependence testing ................... 55
2.9 Inadequacies of the Data Access Descriptor representation ... 56
2.10 Dependence as a measure of inter-process communication .... 59
2.11 Comparison with related work ........................ 59
2.12 Summary .......................................... 61

3 Parallelism Detection and Enhancement ......................... 62
  3.1 Task graphs ...................................... 62
  3.2 Structured and non-structured task graphs ............... 64
    3.2.1 FJ-graphs .................................. 64
    3.2.2 Chain production .................................. 68
    3.2.3 FJ-reducible graphs ............................ 68
  3.3 Significance of structured and non-structured task graphs .... 70
  3.4 A hierarchical parallelism detection procedure .......... 71
3.4.1 Detection of loop parallelism ........................................... 77
3.4.2 Finding nested parallelism ............................................. 77
3.4.3 Dealing with programmer specified parallelism ....................... 79
3.4.4 Time complexity ...................................................... 80
3.4.5 Accuracy of dependence testing ........................................ 81
3.5 Task Pipelining ............................................................ 83
  3.5.1 Pipelining two loops .................................................. 87
  3.5.2 Pipelining with a PARALLEL SECTIONS ......................... 90
3.6 Automatic pipelining ....................................................... 92
3.7 Programmer guided pipelining ............................................. 92
3.8 Related work .............................................................. 93
3.9 Summary ................................................................. 94

4 Analyzing synchronization .................................................. 95
  4.1 Simplifying assumptions .................................................. 96
  4.2 Representing synchronization and control flow ......................... 98
  4.3 The co-graph ................................................................ 98
    4.3.1 Determinacy of parallel programs .................................. 100
    4.3.2 Constructing the co-graph ........................................... 100
    4.3.3 Detecting race conditions using the co-graph ..................... 101
  4.4 The co-graph as a graph of concurrent states of a program .......... 103
  4.5 Multiple posts and conditional branches ................................ 106
    4.5.1 Deadlock and infinite wait ......................................... 107
  4.6 Loops and event arrays .................................................. 108
    4.6.1 Deadlock revisited .................................................. 113
  4.7 Putting it all together ................................................... 115
    4.7.1 Safe distances for the whole program ............................ 116
    4.7.2 Safe distances and the co-graph .................................. 117
5 A Parallel Programming Tool 121
5.1 Internal representation of the program 121
5.2 Parallelism detection 124
  5.2.1 Programmer guided parallelism detection 125
5.3 Parallelism enhancement 127
5.4 Identification of race conditions 128
5.5 Debugging support 128
5.6 Information about program behavior 129
5.7 A parallel programming tool 129
5.8 Summary 131

6 Conclusion 132
6.1 Justification for programmer guidance 132
6.2 Justification of parallelization strategy 133
6.3 Programmer-tool interaction 134
6.4 The Task Dependence Graph as an abstract representation of
  parallelism in the program 135
6.5 Expected impact of this work 136

Bibliography 138
### Illustrations

<table>
<thead>
<tr>
<th>Section</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Data dependence between two regions.</td>
<td>13</td>
</tr>
<tr>
<td>2.2</td>
<td>Example of a convex and a non-convex polytope.</td>
<td>15</td>
</tr>
<tr>
<td>2.3</td>
<td>Some simple sections in 2 and 3 dimensions.</td>
<td>15</td>
</tr>
<tr>
<td>2.4</td>
<td>Simple boundaries in 2 dimensions.</td>
<td>17</td>
</tr>
<tr>
<td>2.5</td>
<td>A simple section in 2 dimensions and its redundant boundaries.</td>
<td>18</td>
</tr>
<tr>
<td>2.6</td>
<td>Algorithms to compute the intersection and union of simple sections.</td>
<td>20</td>
</tr>
<tr>
<td>2.7</td>
<td>Intersection and union of simple sections.</td>
<td>21</td>
</tr>
<tr>
<td>2.8</td>
<td>Computation of redundant boundaries by algorithm INTERSECT.</td>
<td>25</td>
</tr>
<tr>
<td>2.9</td>
<td>Traversal order $\tau = x_2 \succ {x_1, x_3}$.</td>
<td>28</td>
</tr>
<tr>
<td>2.10</td>
<td>Algorithm to compute the simple section of an array accessed within a loop.</td>
<td>32</td>
</tr>
<tr>
<td>2.11</td>
<td>Some array references within loops.</td>
<td>37</td>
</tr>
<tr>
<td>2.12</td>
<td>Algorithm to translate a Data Access Descriptor in terms of an enclosing loop.</td>
<td>39</td>
</tr>
<tr>
<td>2.13</td>
<td>Algorithm to translate a Data Access Descriptor from a subroutine to its call site.</td>
<td>40</td>
</tr>
<tr>
<td>2.14</td>
<td>LU Decomposition program.</td>
<td>43</td>
</tr>
<tr>
<td>2.15</td>
<td>Data Access Descriptors for the stores to A within LUDCOMP.</td>
<td>46</td>
</tr>
<tr>
<td>2.16</td>
<td>Procedure to test for data dependence (Data Access Descriptor intersection).</td>
<td>50</td>
</tr>
<tr>
<td>2.17</td>
<td>Procedure to compute the union of two Data Access Descriptors.</td>
<td>51</td>
</tr>
<tr>
<td>2.18</td>
<td>Some non-simple sections in 2 and 3 dimensions.</td>
<td>58</td>
</tr>
</tbody>
</table>
2.19 (a) A non-simple section, and (b) its simple section approximation.  
3.1 A task graph and its chain expression.  
3.2 Some structured and non-structured graphs and their chain expressions.  
3.3 Graph grammars for (a) FJ-graphs and (b) FJ-reducible graphs.  
3.4 Some examples of FJ-graphs: F denotes a fork node and J a join node.  
3.5 FJ-reducible graphs and their equivalent FJ-graphs.  
3.6 Classes of structured task graphs.  
3.7 Algorithm to uncover parallelism in the program.  
3.8 Procedure to recursively search for parallelism at deeper levels.  
3.9 Illustration of procedure PAR_SEARCH.  
3.10 (a) The input task graph \( G \) and (b) the output of algorithm FIND_PARALLELISM.  
3.11 Parse of \( G \) using the structured graph grammar.  
3.12 Algorithm to pipeline two tasks.  
3.13 Procedure to insert synchronization for pipelining.  
3.14 Illustration of task pipelining.  
3.15 Parallelism detection and enhancement using the Task Dependence Graph.  
4.1 (a) A parallel program segment, (b) its Program Execution Graph and (c) its co-graph.  
4.2 Three graph representations of a parallel program.  
4.3 Algorithm to construct the co-graph.  
4.4 Effect of multiple posts and conditional branches.  
4.5 Infinite wait and circular wait (or deadlock). \( P \) and \( W \) denote post and \( \text{wait} \) respectively; \( c \) denotes a boolean expression.  
4.6 Program Execution Graph with loops.
4.7 Deadlock due to incorrect use of event arrays. .................. 114

5.1 A program and its abstract syntax tree representation. ........... 123
5.2 A parallel programming tool. ........................................ 130
Chapter 1

Introduction

Over the last decade, parallel computing has emerged as a relatively low-cost alternative to expensive and exotic hardware for realizing very high computation speeds. Several commercially available parallel computers claim peak computation speeds in the range of Gigaflops (i.e., several billion floating point operations per second). These are very impressive figures, but unfortunately such speeds are not easy to achieve in practice. To fully exploit the power of a parallel computer, a program should be able to partition the total computation across all processors, driving each one at its maximum speed. Determining the "best" partition involves making a compromise between maximizing the parallelism and minimizing run-time overhead arising from scheduling and synchronization of the parallel processes. Also, the partitioned program must consistently produce correct results.

Most people are familiar with conventional uniprocessor computers, and writing sequential programs for them is relatively easy. Unfortunately, the speed provided by such computers is insufficient for complex compute-intensive applications. A parallel computer solves the problem of speed, but writing parallel programs that achieve the desired speed demands considerable intellectual effort from the programmer. It is paradoxical that the time gained with fast parallel hardware is lost to some extent in trying to write the parallel program. This thesis presents a framework for the design of a parallel programming tool that can aid in the program development process and lessen the time and effort devoted to writing efficient parallel programs.
1.1 The nature of numerical scientific programs

Throughout this thesis, we will focus on parallel programs for scientific applications. Examples of scientific applications include weather prediction, seismic exploration, satellite image processing, industrial process control and simulations of physical phenomena. A characteristic feature of these problems is their highly compute-intensive nature, involving large amounts of data. Very high computation speeds are required to process huge amounts of data in a reasonable time. For example, a weather prediction program that predicts the next day's weather is not very useful if the program itself needs 24 hours to run! Aggressive parallelization can provide the kind of speed required for such applications.

Programs for scientific applications typically use numerical techniques to model the mechanics of the entities whose behaviour is being studied. Spatial structures are usually modeled by 1, 2 or 3 dimensional grids which are represented in the program by 1, 2 or 3 dimensional arrays respectively. The elements of the array correspond to grid points in the model. The numerical algorithms that are used to perform computations on the array elements (or grid points) generally access the array in very regular ways [Ric83, Ric81, Hel78, DS87]. For 2 dimensional arrays, the following access patterns are encountered very frequently: entire array, successive columns, rows or diagonals, upper and lower triangular sections and bands. The higher dimensional analogs of these access patterns are observed frequently in arrays of larger dimensions. Much of the potential parallelism in scientific programs exploits the fact that different portions of the program access different sections of an array and can therefore be executed simultaneously on parallel processors [DS86]. The programs tend to be several thousands of lines long, and their structure is fairly complex, consisting of several nested procedures and loops. The remainder of this section describes some examples of the kind of numerical scientific programs targeted in this thesis.
1.1.1 PIC particle in cell simulation

The PIC algorithm models the collisionless, electrostatic interaction between two superimposed plasma beams with a relative drift velocity [LFHM85, HLM84]. The field and charge distributions are discretized over a 2 dimensional grid. During each time step of the simulation, Poisson's equations for electric potential, with periodic boundary conditions, are used to determine the electric field. The particles are then accelerated under the influence of this field. All the particles are kept in a central particle bank modeled by a 2 dimensional array. Each column of the array contains data parameters for one of the particles. The parallelism in the program is inherent in the computational independence of each particle being accelerated by the electric field. Each process picks a particle from the particle bank and accelerates it for one time step. Their charges are summed and the results placed in a global grid within a critical section of the code, and Poisson's equation is solved over the grid.

1.1.2 SIMPLE hydrodynamics simulation

The SIMPLE code models the hydrodynamic and thermal behavior of fluids, using a 2 dimensional Lagrangian finite difference formulation of the equations of hydrodynamics with an explicit time scheme. Some of the computationally different loops in SIMPLE can be executed concurrently. For example, boundary calculations performed during the hydrodynamics segment of each time step can be processed in parallel with the interior mesh computations, with appropriate synchronization specified at the boundaries.

1.1.3 Seismic interpolation

Seismic interpolation attempts to obtain a picture of the subsurface from field data gathered using reflection seismology [Ben86]. Forward modeling is used to predict the type of field data that would be obtained from a given structure, and this is used to obtain a more accurate interpretation for a given volume of earth. A 3 dimensional
array is used to approximate the geological structure, and wave equations are solved over the grid points (array elements) to obtain values of the variables. Computations over independent plane sections of the grid can be done in parallel. Within each plane, there are additional opportunities for parallelism between accesses to independent rows.

1.2 Parallelism model

The techniques developed in this thesis assume that the target parallel computer is of the Multiple-Instruction-Multiple-Data stream (MIMD) category. The parallel computer is assumed to have a highly interleaved global shared memory. The shared memory is the only means of communication between the processors. The processors may have local memories (chaches), and are connected to the global shared memory via a high speed interconnection network.

The parallelism specified by the programmer and the parallelism uncovered by the proposed analysis techniques has the fork-join property. Parallel tasks are created simultaneously at a fork node, resulting in multiple threads of execution, one for each task. Each fork node has a corresponding join node, where all tasks created at the fork terminate together to form a single thread of execution. The tasks within a fork-join region can be synchronized to enforce a partial order between statements in different execution threads.

1.3 Specification of parallelism

Most numerical scientific programs are written in some flavor of Fortran, and almost all commercial parallel computers support this language. They also provide several extensions to the language for explicitly specifying parallelism. The techniques presented in this thesis are not restricted to programs written in Fortran. They applicable to most block-structured languages. However, the techniques cannot handle recursive
procedures, pointer variables and dynamically allocated structures; extension of the
techniques for these cases is a topic of future research.

This techniques developed in this thesis support programmer specified parallelism
in the program. Unfortunately, there is currently no accepted standard for parallel
language extensions. Thus, the parallel constructs provided in the PCF Fortran lan-
guage were chosen as a representative for the kinds of parallelism that can be specified
in a program. PCF Fortran is a language standard that extends the Fortran77 stan-
dard with parallel constructs. It has been proposed by the Parallel Computing Forum,
an organization which consists of several leading researchers and representatives from
industry [For88]. The relevant parallel constructs are described below.

- PARALLEL SECTIONS:

This construct describes parallelism between sections of code. A section can be
any legal Fortran77 or PCF Fortran program segment. Its syntax is:

\[
\begin{align*}
\text{PARALLEL SECTIONS} \\
\text{[ LOCAL } \text{name list} \text{]} \\
\text{SECTION} \\
\text{ statement } [\ldots] \\
\text{[ SECTION} \\
\text{ statement } [\ldots] \text{] } [\ldots] \\
\text{END PARALLEL SECTIONS}
\end{align*}
\]

Variables declared LOCAL are local in scope to each of the parallel sections.
Serial DO loop indices are automatically declared LOCAL to the first parallel
construct that encloses them. Modification of variables that are not local to the
parallel sections must be appropriately protected.

- PARALLEL DO:

This construct describes parallelism between iterations of a loop. Its syntax is:

\[
\begin{align*}
\text{PARALLEL DO} \\
\text{[ LOCAL } \text{name list} \text{]} \\
\text{ statement } [\ldots] \\
\text{END PARALLEL DO}
\end{align*}
\]
The loop induction variable of a PARALLEL DO loop is automatically declared LOCAL. Serial DO loop indices are automatically declared LOCAL to the first parallel construct that encloses them. Modification of variables that are not local to the iterations of the parallel DO must be appropriately protected.

- Critical sections:
  A critical section provide a mechanism for exclusive access to a section of the code. Its syntax is:
  
  \[
  \text{LOCK ( } \text{sync} \text{ )} \\
  \text{statement [ . . . ]} \\
  \text{UNLOCK ( } \text{sync} \text{ )}
  \]
  
  where \text{sync} is the name of a variable. If \text{sync} is an array variable, then the LOCK and UNLOCK must be performed on a specified element of the array.

- Synchronization:
  The POST, WAIT and CLEAR statements provide a mechanism for synchronizing processes. Their syntax is:
  
  \[
  \text{POST EVENT ( } \text{sync} \text{ )} \\
  \text{WAIT EVENT ( } \text{sync} \text{ )} \\
  \text{CLEAR EVENT ( } \text{sync} \text{ )}
  \]
  
  where \text{sync} is the name of a variable. If \text{sync} is an array variable, then the POST, WAIT and CLEAR is performed on a specified element of the array. The POST statement changes the state of \text{sync} to \text{posted}, and is a non-blocking operation. The WAIT statement causes the process executing it to block until \text{sync} is \text{posted} by another process. The CLEAR statement changes the state of \text{sync} to \text{cleared}, so that the names event variable can be reused.

1.4 Program partitioning for parallelism

A parallel programming language gives the programmer the option of dealing with the problem of program partitioning at the higher abstract level of the programming
language. The programmer can use an algorithm that lends itself to a natural decomposition into independent parts, and explicitly specify the decomposition using the language extensions. The price for the flexibility provided by these language extensions is the burden of ensuring the correctness of the program. In particular, synchronization between the parallel processes must be explicitly specified and accesses to shared variables must be appropriately guarded. This is a task that can be complicated even for fairly simple problems. In more complex problems like those encountered in numerical scientific codes, it can be much more arduous. Researchers at several national laboratories and universities have been spending large amounts of time partitioning supercodes by hand, reordering or rewriting sections of the code in clever ways so that they can be decomposed into separate units that can be run in parallel. They are extremely cautious to avoid introducing any synchronization errors that could produce race conditions during parallel execution.

An alternative approach to the problem of program partitioning for parallelism is to continue using conventional “sequential” languages and let a “parallelizing” compiler automatically perform the partitioning. This relieves the programmer from having to worry about possible non-deterministic behaviour of the partitioned program. This is now the compiler’s responsibility. Compilers like Parafrase (developed at the University of Illinois at Urbana-Champaign) and PFC (developed at Rice University) use a technique known as data dependence analysis to detect and expose parallelism in loops [AK82]. The motivation for performing data dependence analysis is that any execution order of the statements in a program will give correct results so long as it satisfies all data dependences between the statements. The data dependences impose a partial order on the statements of a program, which can be represented by a data dependence graph. The nodes of the data dependence graph are statements, and the edges represent the dependences between the statements.

Most of the computation in scientific numerical programs tend to be concentrated in loops. Thus the loop is a good candidate for the kind of medium to coarse grain
parallelism that would provide good speedups on many of the currently available parallel computers. The absence of dependences between iterations of a loop (called loop-carried dependences) indicates that the results of executing the loop are independent of the order of scheduling of its iterations on the parallel processors. Such a loop is said to be safely parallel, because the results of its execution are deterministic regardless of the order in which its iterations are executed. If loop-carried dependences exist, the compiler usually attempts to restructure the code within the loop in an attempt to eliminate the dependences. When some of these dependences cannot be removed, the compiler may insert synchronization to satisfy the execution order dictated by these dependences. Since the compiler ensures that all data dependences are preserved, the partitioned program is guaranteed to preserve the semantics and results of sequential execution. The programmer can use conventional sequential debugging techniques to debug the sequential program and leave the task of parallelizing to the compiler.

One drawback of this approach is that the compiler may not be able to detect all regions of the program that can be parallelized. This is primarily because the compiler uses data flow information that is statically collected, and sometimes has to make conservative decisions based on this information. More importantly, the compiler only looks for parallelism in the physical program that implements an algorithm, while the programmer usually has a much better idea of the inherent parallelism in the algorithm. In particular, the inherent parallelism in the algorithm may not be restricted to an iterative computation, but the compiler only analyzes iterative portions (which are typically coded as DO loops) for parallelism. To capture all of the parallelism that the algorithm provides, it is necessary to investigate parallelism detection techniques at a more global level than the DO loop.
1.5 Parallel programming tools

A more promising approach to program partitioning for parallelism would be to support both language extensions (i.e., parallel language constructs) and the parallelizing compiler in the same environment. The programmer would be allowed to express parallelism in his program explicitly, and tools would be provided to aid him in the program development process. These tools could use the data dependence information collected by the compiler to report regions of the program that have not been safely parallelized, and perhaps identify locations where synchronization must be inserted [ABKP86, Tri87, PGW87, SAB88, GGGJ88]. In addition, they could be used to find regions that can be run in parallel, which the programmer may have overlooked. This scheme gives the programmer the power to tailor the parallelism in the program to suit a particular target computer, with assistance from the compiler in detecting potential anomalies. The PTOOL project at Rice University is a step in this direction [ABKP86].

PTOOL allows the programmer to focus on a selected region of the program and browse through the list of data dependences that inhibit parallelization of the region. PTOOL also lists the variables that must be declared in shared global storage to avoid non-deterministic results on parallel execution of the selected region. While such a tool can be very effective in locating parallelism hazards, it does not aid in removing parallelism inhibiting dependences. The ParaScope Editor (PED) which is one of the utilities in the ParaScope parallel programming environment, currently under development at Rice University, is aimed at providing such a facility [CCH+88, BKK+89]. PED has all the capabilities of PTOOL, with the additional feature that the programmer can request the tool to suggest or perform the transformations necessary to eliminate the parallelism inhibiting dependences in a selected region. When such a transformation is not possible PED reports that the selected region must be run sequentially. ParaScope is implemented as one of the utilities in the IRN programming
environment, a software development environment designed to support the development of large programs [AK84].

While dependence analysis tools like PTOOL have shown much promise, there are several inadequacies that need to be addressed [Hen87]:

1. *Understanding of programmer specified parallelism and synchronization.* The tool should check for potential anomalies in programmer specified parallel regions as well as detect any opportunities for parallelism that the programmer may have overlooked. Often, the programmer inserts synchronization in order to satisfy some data dependence that he has recognized. The tool should ignore such (satisfied) dependences, since they do not further inhibit parallelism.

2. *Information about higher levels of parallelism.* Several parallel computers allow the programmer to exploit coarser grain parallelism than the DO loop. Both PTOOL and PED are currently helpful only in detecting parallelism in a DO loop. It is unclear how these techniques can be extended to report about other levels of parallelism.

3. *Ability to interact at a "more conceptual" level.* Interactive dependence analysis tools are useful only if the programmer understands the concept of dependence and its relationship to parallelism. Dependences represent fairly low level information, and the programmer can be easily overwhelmed by the enormity of parallelism inhibiting dependences. It would be useful if the interaction with the programmer could be conducted at a higher conceptual level that is more intuitive than data dependences between statements.

1.6 **Overview of the thesis**

This thesis will focus on the problems involved in partitioning large-scale scientific applications for execution on shared memory MIMD type parallel computers. The overall emphasis will be developing a framework that can serve as a basis for the
design of powerful parallel programming tools. The proposed framework allows parallelism of different granularities to be treated in a uniform manner. Interaction with the programmer can be conducted at a conceptual level, or optionally low-level information can be presented on request.

Chapter 2 proposes a framework that allows dependence testing in a parallel programming tool to be treated in a uniform manner. Chapter 3 presents an intermediate representation that is more powerful than the data dependence graph and can be used to represent partitioned programs. This internal representation is used for parallelism detection and enhancement. The algorithms for parallelism detection use the dependence testing methods developed in Chapter 2. Chapter 4 presents a technique for analyzing synchronization in programmer specified parallel regions. This allows a parallel programming tool to support programs with programmer specified parallelism. Chapter 5 suggests a method of integrating the techniques discussed in the previous chapters into a parallel programming tool that can aid the programmer in the development of parallel programs. Finally, Chapter 6 discusses the impact that these ideas might have on the design of future parallel programming tools, and identifies some problems that require further research.
Chapter 2

A General Theory of Dependence Testing

This chapter presents a general theory of dependence testing, that allows us to treat data dependences between entire regions of the program in the same manner as data dependences between statements. A data dependence at the region level will be a summary of all statement level data dependences between the regions. In this sense, a region level data dependence will contain “higher bandwidth” information than a statement level data dependence. The general theory presented here is fundamental to the concepts discussed in the rest of this thesis. In subsequent chapters, unless otherwise specified, we will use the term dependence to mean the general definition given here.

2.1 An extended definition of data dependence

Traditionally, a data dependence is said to exist between a pair of statements if both access one or more common memory locations. The following definition generalizes the notion of data dependence to groups of statements (or regions) of a program:

Definition 2.1 (data dependence): Let $R_1$ and $R_2$ be two regions in a program. Then, there exists a data dependence between $R_1$ and $R_2$ if there is some set of memory locations that are accessed by both regions. The dependence between the two regions is quantified by the set of memory locations that are accessed by both.

The set of memory locations accessed in region $R_1$ is denoted by $\Delta(R_1)$, and the set accessed in region $R_2$ by $\Delta(R_2)$. A dependence between the two regions can be represented by $\Delta(R_1 R_2) = \Delta(R_1) \cap \Delta(R_2)$, consisting of all locations referenced by both regions (see Figure 2.1). $\Delta(R)$ is called the Data Window of the region $R$. The
Data Window of a region describes the portion of the program's data structures that may be accessed during an execution of the statements in that region. The Data Window consists of a set of Data Access Descriptors, one for each variable referenced in the region. The Data Access Descriptor of a variable $A$ in a given region $R$, denoted by $\delta_R(A)$, contains a summary of the accesses to the variable $A$ within the region $R$. $\Delta(R_1 R_2)$ is the Data Overlap between $R_1$ and $R_2$, and represents the portion of the program's data structures that are accessed by both the regions $R_1$ and $R_2$.

2.2 Summarizing data access in a region

A program variable that is referenced in a region can either be a scalar or an array. One way to describe the variables accessed within a region is to simply keep a list of all the variable names referenced in that region. While such a list is sufficient in the case of scalar variables, it does not provide enough information about array variables because references to different portions of the same array will not be differentiated. Two regions may access different portions of the same array, but we would incorrectly
conclude that there is a dependence between the regions. Much parallelism, especially
in numerical scientific applications, originates from partitioning arrays so that diffe-
rent processes can operate on separate portions of the same array simultaneously. It
is therefore crucial that we keep track of the section of each array that is accessed
within a region. This is the purpose of a simple section.

2.2.1 Simple sections

A simple section is designed to provide a compact representation of commonly en-
countered array access shapes. It also allows certain frequent operations involving
simple sections to be fast and easy to implement. To achieve these properties, it is
necessary to impose some restrictions on the shapes that can be described by a simple
section. The most important restriction is that a simple section can only be bounded
by simple boundaries.

Definition 2.2 (simple boundary): Given an n dimensional space
with coordinate axes \( x_1, x_2, \ldots, x_n \), a simple boundary is a hyperplane of
the form \( x_i = c \) or \( x_i \pm x_j = c \), where \( x_i, x_j \) are any two different coordinate
axes, and \( c \) is an integer constant.

Thus, a simple boundary can either be parallel to any one coordinate axis, or at a
45° angle to a pair of coordinate axes. The above definition can be extended to allow
a symbolic expression rather than the integer constant. The only kind of section
describable by the Data Access Descriptor is the simple section.

Definition 2.3 (simple section): Any convex polytope with simple
boundaries is a simple section.

A polytope is convex if for any pair of points in the polytope, the line segment
between them lies entirely within the polytope (see Figure 2.2). Many commonly
occurring access patterns, such as access of the entire array, access of a triangular
section and access of a single diagonal can be described precisely by a simple section.
Figure 2.3 shows some simple sections in 2 and 3 dimensions. For sections that do
not have simple boundaries, we can either assume that the entire array is accessed, or determine the simple section that contains it. The properties of simple sections are developed in the following observations and lemmas.

**Observation 2.1** Any simple section in 2 dimensions can be described by at most 8 simple boundaries.

A simple section in 2 dimensions with the maximum possible number of boundaries can be constructed in the following manner. It has one pair of simple boundaries of the form \( x = c \), one pair of the form \( y = c \), one pair of the form \( x + y = c \) and one pair of the form \( x - y = c \). These four pairs of lines bound a figure in 2 dimensional space.
We have exhausted all possible simple boundaries for this figure\(^1\). If it had a ninth boundary, it would be a non-simple boundary, and the figure would not be a simple section. Figure 2.4 illustrates a simple section in 2 dimensions that has 8 boundaries. The following lemma generalizes the above observation for arbitrary dimensions:

**Lemma 2.1** Any simple section in \(n\) dimensions can be described by at most \(2n^2\) simple boundaries.

**Proof:** Consider an \(n\) dimensional space with \(n\) coordinate axes \(x_1, x_2, \ldots, x_n\). Let us construct a simple section in this space which has the maximum possible number of simple boundaries. For each coordinate axis \(x_i\), there is a pair of bounding hyperplanes of the form \(x_i = c\). By definition, each of these are simple boundaries. This accounts for \(2n\) simple boundaries. For every pair of coordinate axes \(x_i\) and \(x_j\) there is a pair of “diagonal” bounding hyperplanes of the form \(x_i + x_j = c\) and another pair of “diagonal” bounding hyperplanes of the form \(x_i - x_j = c\). These boundaries are also simple by definition. Since there are \(\binom{n}{2}\) possible ways of choosing a pair of coordinate axes from \(n\) axes, the diagonal hyperplanes account for \(2\binom{n}{2} = n(n - 1)\) more pairs of simple boundaries, or \(2n(n - 1)\) simple boundaries. We have exhausted all possible ways of forming simple boundaries. The simple section described by these bounding hyperplanes has \(2(n + 2\binom{n}{2}) = 2(n + n(n - 1)) = 2n^2\) boundaries. \(\square\)

A simple section in \(n\) dimensions can have fewer than \(2n^2\) boundaries. In this case, the remaining boundaries that do not contribute to the shape of the section are called redundant boundaries. Figure 2.5 shows a simple section in 2 dimensions with its redundant boundaries shown as dotted lines. Although redundant boundaries do not contribute to the shape of a section, they are very useful when computing the union of simple sections. Thus, a simple section in \(n\) dimensions is described by specifying all its \(2n^2\) simple boundaries (i.e., both non-redundant and redundant boundaries). All

\(^1\)The line \(y - x = c\) is equivalent to a line of the form \(x - y = c\); the minus is absorbed in the constant term.
the boundaries are required to be tight boundaries (i.e., each boundary containing at least one point of the simple section). Since redundant boundaries do not contribute to the shape of the section, a tight redundant boundary must touch the section at a vertex.

A convenient way to represent a simple section $S$ is to specify its simple boundary pairs:

**Definition 2.4 (simple boundary pair):** A simple boundary pair is an inequality of the form $\alpha \leq \psi(x_1, \ldots, x_n) \leq \beta$, where $\psi(x_1, \ldots, x_n)$ is a function of the coordinate axes $x_1, \ldots, x_n$, such that $\psi(x_1, \ldots, x_n) = \alpha$ and $\psi(x_1, \ldots, x_n) = \beta$ are both simple boundaries of the section $S$. $\psi(x_1, \ldots, x_n) = \alpha$ is called a lower boundary and $\psi(x_1, \ldots, x_n) = \beta$ is called an upper boundary.

For example, the 2 dimensional simple section in Figure 2.5 can be represented by the simple boundary pairs:

\[
\begin{align*}
2 & \leq x \leq 7 \\
1 & \leq y \leq 6 \\
3 & \leq x + y \leq 10 \\
-4 & \leq x - y \leq 4
\end{align*}
\]
Thus, any simple section in $n$ dimensions can be represented by $n^2$ simple boundary pairs. If the maximum number of dimensions is fixed, any simple section can be represented in a constant of storage by listing the integer constants of its simple boundary pairs. A $k$ dimensional simple section (described in a space with coordinate axes $x_1, \ldots, x_k$), is represented by its simple boundary pairs in the following order according to their $\psi(x_1, \ldots, x_k)$ expressions:

\[
x_1, x_2, \ldots, x_k,
\]
\[
x_1 + x_2, x_1 + x_3, \ldots, x_1 + x_k,
\]
\[
x_2 + x_3, x_2 + x_4, \ldots, x_2 + x_k,
\]
\[
\vdots
\]
\[
x_{k-1} + x_k,
\]
\[
x_1 - x_2, x_1 - x_3, \ldots, x_1 - x_k,
\]
\[
x_2 - x_3, x_2 - x_4, \ldots, x_2 - x_k,
\]
\[
\vdots
\]
\[
x_{k-1} - x_k.
\]
Thus, we only require enough storage for $2(\text{MAXDIMS})^2$ signed integer constants to describe any simple section of dimension $k \leq \text{MAXDIMS}$. For example, the simple boundary pairs of a 3 dimensional simple section are specified in the following order:

$$
\begin{align*}
&c_1 \leq x \leq c_2, \quad c_3 \leq y \leq c_4, \quad c_5 \leq z \leq c_6, \\
&c_7 \leq x + y \leq c_8, \quad c_9 \leq x + z \leq c_{10}, \\
&c_{11} \leq y + z \leq c_{12}, \\
&c_{13} \leq x - y \leq c_{14}, \quad c_{15} \leq x - z \leq c_{16} \\
&c_{17} \leq y - z \leq c_{18}
\end{align*}
$$

Any 3 dimensional simple section can be uniquely described by listing its 18 constant terms $c_1, c_2, \ldots, c_{18}$ in the above order. In real programs, arrays of large dimensions are rare, so that it is meaningful to fix the maximum number of dimensions to be some small number (such as 3 or 5).

2.2.2 Intersection and union of simple sections

Union and intersection operations are frequently done on simple sections. Algorithms UNION and INTERSECT compute the union and intersection of simple sections respectively (Figure 2.6).

**Observation 2.2** Simple sections are closed under intersection, but are not closed under union.

By definition, every simple section is a convex polytope. Convex polytopes are closed under intersection; therefore the intersection is convex. Each boundary of the intersection is also a boundary of one of the two intersecting sections. Since the two sections are simple sections, these boundaries must also be simple boundaries. Thus, the intersection has simple boundaries and is convex. It follows by definition that the intersection is itself a simple section. This justifies the first part of the observation. Figure 2.7 shows two simple sections labeled $S_1$ and $S_2$. The union of these two sections is not simple because the convexity condition is violated. This justifies the latter part of the observation.

Algorithm INTERSECT always chooses the “more interior” of the two compared boundary pairs and adds it to the intersection $S$ (line 5). The check on line 3 is done
Algorithm INTERSECT

Input: two simple sections $S_1$ and $S_2$, both of dimension $n$, represented by their simple boundary pairs.
Output: a simple section $S = S_1 \cap S_2$ if the intersection is non-empty, and $\phi$ otherwise.
begin
1 for each simple boundary pair $B_i = \alpha_i \leq \psi(x_1, \ldots, x_n) \leq \beta_i \in S_1$ do
2 get its corresponding boundary pair $B_i' = \alpha_i' \leq \psi(x_1, \ldots, x_n) \leq \beta_i' \in S_2$;
3 if $\max(\alpha_i, \alpha_i') > \min(\beta_i, \beta_i')$
4 then return $\phi$
5 else add the boundary pair $B = \max(\alpha_i, \alpha_i') \leq \psi(x_1, \ldots, x_n) \leq \min(\beta_i, \beta_i')$ to $S$;
endfor
6 return $S$;
end

Algorithm UNION

Input: two simple sections $S_1$ and $S_2$, both of dimension $n$, represented by their simple boundary pairs.
Output: $S = simp(S_1 \cup S_2)$, the simple section containing the union $S_1 \cup S_2$.
begin
1 for each simple boundary pair $B_i = \alpha_i \leq \psi(x_1, \ldots, x_n) \leq \beta_i \in S_1$ do
2 pick its corresponding boundary pair $B_i' = \alpha_i' \leq \psi(x_1, \ldots, x_n) \leq \beta_i' \in S_2$;
3 add the boundary pair $B = \min(\alpha_i, \alpha_i') \leq \psi(x_1, \ldots, x_n) \leq \max(\beta_i, \beta_i')$ to $S$;
endfor
4 return $S$;
end

**Figure 2.6** Algorithms to compute the intersection and union of simple sections.

to ensure that the boundary pair to be added to $S$ is consistent. An inconsistent boundary is encountered only when the two sections $S_1$ and $S_2$ do not intersect; the algorithm returns $\phi$ in this case. $\phi$. Note that even when $S_1$ and $S_2$ have only one point in common, the algorithm will treat the intersection $S$ as being non-empty.

Since it is not possible to compute the exact union and still stay within the simple section framework, algorithm UNION computes an approximation to the exact union, $S = simp(S_1 \cup S_2)$, the simple section containing $S_1 \cup S_2$. The algorithm always
chooses the "more exterior" of the two compared boundary pairs and adds it to $S$ (line 3). The following lemma proves an interesting property about simp($S_1 \cup S_2$):

**Lemma 2.2.** Algorithm UNION computes $S = \text{simp}(S_1 \cup S_2)$, the smallest simple section containing the union $S_1 \cup S_2$.

**Proof:** The boundaries of $S$ must be simple boundaries because algorithm UNION compares corresponding simple boundary pairs in $S_1$ and $S_2$ to produce a simple boundary pair for $S$. The algorithm always chooses the "more exterior" boundary in line 3 to add to $S$. Since only corresponding simple boundaries are compared, both compared boundaries are parallel to one another. Choosing the "more exterior" of the two parallel boundaries will ensure that all points that satisfy the interior boundary also satisfy the exterior boundary. Clearly, the section $S$ described by these boundaries contains both $S_1$ and $S_2$. To show that $S$ is convex, consider any two points $P$ and $Q$ in $S$. Let their coordinates be of the form $(x_1, x_2 \ldots, x_n)$. Arbitrarily choose a coordinate position, say the $k$th coordinate. Let $x_k = p$ for point $P$ and
$x_k = q$ for point $Q$. Since $S$ is bounded by simple boundaries, both $p$ and $q$ must satisfy inequalities of the form $\alpha_1 \leq x_k \leq \beta_1$ and $\alpha_2 \leq x_k + x_i \leq \beta_2$ ($1 \leq i \leq n$, $i \neq k$).

Consider any point on the line segment $\overline{PQ}$. Its $k$th coordinate can be expressed in the form $pr + (1 - r)q$, for some constant $r$. Substituting the bounds for $p$ and $q$ from the first inequality, we get:

$$\alpha_1 r + (1 - r)\alpha_1 \leq pr + (1 - r)q \leq \beta_1 r + (1 - r)\beta_1$$

$$\Rightarrow \quad \alpha_1 \leq pr + (1 - r)q \leq \beta_1$$

Thus, the $k$th coordinate of any point on the line segment $\overline{PQ}$ also satisfies the first inequality. Similarly, by substituting the bounds for $p$ and $q$ from the second inequality, it can be shown that the $k$th coordinate of any point on the line segment $\overline{PQ}$ also satisfies this inequality. Since the coordinate position $k$ was chosen arbitrarily, we can conclude that every point on the line segment $\overline{PQ}$ satisfies all the inequalities satisfied by the points $P$ and $Q$. We have just demonstrated that for any two points in $S$, the line segment between them is contained entirely within $S$. $S$ is therefore convex by definition.

Since $S$ is convex and has simple boundaries, it is a simple section. $S$ also completely contains the union $S_1 \cup S_2$. We must now show that it is the smallest such simple section. Recall that every simple section is represented by all possible simple boundaries (both redundant and non-redundant) that define it, so we cannot introduce any new simple boundaries. Moreover, all the boundaries are tight boundaries (i.e., each boundary contains at least one point of the simple section). The only way to construct a smaller section is to move one or more boundaries of $S$ towards the interior of $S$. That is, we must pick some simple boundary $b$: $\psi(x_1, x_2, \ldots, x_n) \leq \beta$ and move it “inward” by decreasing $\beta$. However, since $b$ is also a tight simple boundary of one of the two sections $S_1$ and $S_2$, there must be a point belonging to one of them that is on the boundary $b$. This point will be excluded from the section if $b$ were moved inward. Thus, the resulting simple section would not contain the entire
union $S_1 \cup S_2$. Arguing in this manner, we can show that it is impossible to move any boundary of $S$ inward without excluding some point in the union $S_1 \cup S_2$. It follows that $S = \text{simp}(S_1 \cup S_2)$ is the smallest simple section containing the union. \qed

\textbf{Lemma 2.3} The union computed by algorithm UNION is associative, i.e., if $S_1$, $S_2$, and $S_3$ are all simple sections of the same dimension, then:

$\text{simp}(S_1 \cup \text{simp}(S_2 \cup S_3)) = \text{simp}(\text{simp}(S_1 \cup S_2) \cup S_3)$.

\textbf{Proof:} Let $X = \text{simp}(S_1 \cup \text{simp}(S_2 \cup S_3))$ and $Y = \text{simp}(\text{simp}(S_1 \cup S_2) \cup S_3)$. We need to show that $X = Y$. By construction, all simple boundaries of $\text{simp}(S_2 \cup S_3)$ are tight boundaries (i.e., each boundary contains at least one point of $S_2$ or $S_3$). Moreover, the simple section $\text{simp}(S_2 \cup S_3)$ is comprised of some boundaries of $S_2$ and some boundaries of $S_3$. No new boundaries are introduced. Similarly, the simple section $X = \text{simp}(S_1 \cup \text{simp}(S_2 \cup S_3))$ will be comprised of some boundaries of $S_1$ and some boundaries of $\text{simp}(S_2 \cup S_3)$. Clearly, each boundary of $X$ must contain at least one point in $S_1$, $S_2$ or $S_3$. Using a similar argument, we can show that every boundary of $Y$ must also contain at least one point in $S_1$, $S_2$ or $S_3$. It is also easy to show that $X$ and $Y$ must contain all of $S_1$, $S_2$ and $S_3$.

Suppose $X \neq Y$. We should be able to find some boundary that belongs to $X$ but not to $Y$. Let us denote this boundary by $b_X$. Let us compare the equation of the boundary $b_X$ with its corresponding counterpart $b_Y \in Y$. Since $b_X$ is not the same as $b_Y$, it follows that one of them must be a tighter boundary than the other. Without loss of generality, let $b_Y$ be the tighter boundary. Since $b_X$ is a tight boundary of $X$ that contains at least one point in $S_1$, $S_2$ or $S_3$, there must be some point belonging to one of these sections that is excluded from $Y$ because it does not satisfy the boundary $b_Y$. This is a contradiction since every point in $S_1$, $S_2$ and $S_3$ belongs to the simple section $Y$. Thus $b_X$ cannot be different from $b_Y$. $X$ and $Y$ must therefore have the same boundaries and thus represent the same simple section (i.e., $X = Y$). \qed
This important result states that the final shape of a simple section computed by unioning together several different simple sections is independent of the order in which the unions are performed.

**Observation 2.3** Algorithms INTERSECT and UNION each take time proportional to square of the number of dimensions of the section. When the maximum number of dimensions is fixed, they take constant time.

Both algorithm INTERSECT and algorithm UNION examine all corresponding boundary pairs of the sections $S_1$ and $S_2$ only once. When the maximum number of dimensions is fixed, at most $2(MAXDIMS)^2$ boundaries are compared, which takes constant time. There is one subtle aspect to the INTERSECT algorithm that may not be immediately evident. It involves the handling of redundant boundaries. Figure 2.8 shows two regions with some of their redundant boundaries shown as dotted lines. Algorithm INTERSECT computes the boundary labeled $b$ as one of the redundant boundaries of $S$, which is not a tight boundary. $b$ can be refined to the redundant boundary $b'$ which is a tight boundary for the simple section $S$. The particular redundant boundary chosen does not affect the shape or properties of the section $S$. However, if we later union $S$ with another simple section, the shape of the resulting simple section computed by algorithm UNION may depend on the position of the redundant boundaries in $S$. Since algorithm UNION computes an approximation to the exact union of two sections, the redundant boundaries may affect the accuracy of the approximation. In particular, it cannot be guaranteed that algorithm UNION still computes the *smallest* simple section containing the exact union. It is fairly trivial to extend algorithm INTERSECT so that it refines redundant boundaries when they are not tight and still maintains its constant time complexity. For the purposes of this thesis, we assume that all boundaries are tight boundaries.

### 2.3 Formal definition of the Data Access Descriptor

The Data Access Descriptor is defined as follows:
Definition 2.5 (Data Access Descriptor): Given an $n$ dimensional array $A$ that is accessed within a region $R$ of a program, the Data Access Descriptor for $A$ in $R$, denoted by $\delta_R(A)$ is a 3 tuple $(\theta | S | \tau)$, where $\theta$ is the reference template, $S$ is the simple section approximating the portion of $A$ that is accessed within $R$, and $\tau$ is the traversal order of $S$.

The terms reference template, and traversal order will be defined shortly.

$\delta_R(A)$ actually consists of two components, called the write component and read component, denoted by $\delta^w_R(A)$ and $\delta^r_R(A)$ respectively. $\delta^w_R(A)$ is a Data Access Descriptor $(\theta^w|S^w|\tau^w)$ that describes the simple section of $A$ that is written or stored into within region $R$, and $\delta^r_R(A)$ is a Data Access Descriptor $(\theta^r|S^r|\tau^r)$ that describes the simple section of $A$ that is read within region $R$.

2.3.1 Traversal order

The traversal order field of the Data Access Descriptor also contains information about how the section described by the simple section field is traversed. To gather this information from an array reference, we must first determine the dominant induction variable of each subscript position of the reference.

Definition 2.6 (dominant induction variable): Let $A(x_1, x_2, \ldots, x_n)$ be an array reference contained within $m$ loops with induction variables
Consider the loop at nesting depth \( k \) with induction variable \( I_k \) (\( 1 \leq k \leq m \)). The dominant induction variable of the \( i \)th subscript position of the array reference (with subscript expression \( x_i \)) with respect to the \( I_k \) loop is determined as follows:

- If \( x_i \) does not contain any \( I_s \), \( k \leq s \leq m \) (i.e., an induction variable of a loop at depth \( k \) or greater), \( x_i \) has no dominant induction variable.

- If \( x_i \) consists of only one \( I_s \), \( k \leq s \leq m \) (i.e., any one induction variable of a loop at depth \( k \) or greater), \( I_s \) is its dominant induction variable.

- If \( x_i \) contains more than one induction variable belonging to a loop at nesting depth \( k \) or greater, its dominant induction variable is the induction variable of the loop with the greatest nesting depth.

We assume that for a given array reference and a loop \( I_k \) that contains the reference, the dominant induction variables of the subscript positions are available as a vector \( \text{IDOM}_{I_k} \). If the \( i \)th subscript position has no dominant induction variable, \( \text{IDOM}_{I_k}(i) = "-" \), and if its dominant induction variable is \( I_s \), \( s \geq k \), then \( \text{IDOM}_{I_k}(i) = I_s \). The length of the vector \( \text{IDOM}_{I_k} \) is equal to the dimension of the array (i.e., there is one component in the vector for each subscript position in the array reference). \( \text{IDOM}_{I_k} \), for a given loop \( I_k \) containing the array reference, can be computed in a straightforward manner using the method outlined in the definition above.

The traversal order of a section is determined from the dominant induction variables of the array reference.

**Definition 2.7 (traversal order):** Let \( A(x_1, x_2, \ldots, x_n) \) be an array reference contained within \( m \) loops with induction variables \( I_1, I_2, \ldots, I_m \). Consider the loop at nesting depth \( k \) with induction variable \( I_k \) (\( 1 \leq k \leq m \)). Let \( D_1, D_2, \ldots, D_p \) be the list of distinct dominant induction variables of the subscript expressions \( x_1, x_2, \ldots, x_n \) with respect to the \( I_k \) loop, in decreasing order of loop nesting depth. Construct the sets \( V_1, V_2, \ldots, V_p \), where: \( V_i = \{ x_j, 1 \leq j \leq n \mid \text{IDOM}_{I_k}(j) = D_i \} \). The traversal order of the given array reference with respect to the \( I_k \) loop is given by \( r_{I_k} = V_1 \succ V_2 \succ \cdots \succ V_p \).
Two traversal orders $\tau$ and $\tau'$ are said to match if the orders described by both are consistent with one another. This is denoted by $\tau \equiv \tau'$.

Example 2.1:

The computation of traversal order is illustrated for the following reference:

\[
\begin{align*}
&\text{DO } I = 1, 100 \\
&\text{DO } J = 1, 100 \\
&\text{DO } K = 1, 100 \\
&\quad A(I+J, K+2, J) = \cdots \\
&\text{ENDDO}
\end{align*}
\]

If we embed the array $A$ in a 3 dimensional space with coordinates $x_1, x_2, x_3$, then any reference to $A$ can be represented as $A(x_1, x_2, x_3)$. The dominant induction variables for $x_1$, $x_2$ and $x_3$ with respect to the outermost loop are $J$, $K$ and $J$ respectively (i.e., $\text{IDOM}_1 = \{J, K, J\}$). The list of distinct dominant induction variables in decreasing order of nesting depth is: $K$, $J$. We must construct two sets $V_1$ and $V_2$, that consist of the subscript positions of the array reference that have $K$ and $J$ as dominant induction variables respectively. In this case, only the second subscript position has $K$ as a dominant induction variable. Thus $V_1 = \{x_2\}$. The first and third subscript positions of the reference have $J$ as their dominant induction variables. Thus $V_2 = \{x_1, x_3\}$. The traversal order for this array reference with respect to the $I$ loop is thus $x_2 \succ \{x_1, x_3\}$.

Figure 2.9 shows the traversal order represented by $x_2 \succ \{x_1, x_3\}$. This traversal order indicates that access along dimension $x_2$ occurs faster than access along dimensions $x_1$ and $x_3$. $<$

2.3.2 Reference template

The reference template field of the Data Access Descriptor is defined in the following manner.

Definition 2.8 (reference template): Let $A(x_1, x_2, \ldots, x_n)$ be an array reference contained within $m$ loops with induction variables $I_1, I_2, \ldots, I_m$. The reference template for this reference with respect to the $I_k$ loop is a vector with $n$ components. The $i$th component of the vector is determined as follows:
Figure 2.9 Traversal order $\tau = x_2 \succ \{x_1, x_3\}$.

- If $x_i$ has a dominant induction variable with respect to the $I_k$ loop, the $i$th component of the reference template is the variable \( \llbracket x_i \rrbracket \). It indicates that the $i$th subscript is variant with respect to the $I_k$ loop, and IDOM$_{I_k}(i) \neq $ "\.

- If $x_i$ has no dominant induction variable with respect to the $I_k$ loop, the $i$th component of the reference template is the subscript expression $x_i$. In this case, the $i$th subscript is invariant with respect to the $I_k$ loop, and IDOM$_{I_k}(i) =$ "\.

The reference template is used to translate the Data Access Descriptor for a variable from the context of one region to the context of an enclosing region. This is necessary when, for example, the data access information within a subroutine is required in terms of the variables at its call site. Since the data access information for the subroutine is in terms of the formal parameters to the subroutine, it must be translated to be in terms of the actual parameters passed at the call site. Translation of Data Access Descriptors is the subject of a later section.

The simple section of an array reference within a given loop will be described only by those \( \llbracket x_i \rrbracket \) that correspond to subscripts that are variant with respect to that loop. The range of values of the variant subscripts define the shape of the section accessed. The variables \( \llbracket x_j \rrbracket \) that correspond to invariant subscripts do not appear in the simple boundary pairs that describe the simple section. They still contribute to
the shape of the simple section since they represent simple boundary pairs of the form \( \gamma \leq x_j \leq \gamma \), where \( \gamma \) is the expression in the \( j \)th subscript position that is invariant with respect to the given loop.

**Dealing with complex subscript expressions**

Sometimes, a subscript expression may be a non-linear expression in the loop induction variables, or a subscripted variable (such as an array element). Such subscripts are called complex subscripts. When an invariant subscript position has a complex expression, its corresponding entry in the reference template vector will be denoted as "\( \exists \)". This indicates that the subscript position is a symbolic constant whose value is unknown. When a variant subscript position has a complex expression, it may not be possible to accurately determine the range of values that it can have. When constructing the simple section, it is necessary to be conservative, and assume that such a subscript position can have any value within its declared range. This is denoted by a "\( \forall \)" symbol in its corresponding entry in the reference template.

**Example 2.2:**

Consider the following loop nest:

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

The IDOM vector (i.e., the dominant induction variables) and reference template \( \theta \) for the above reference with respect to the \( K \), \( J \) and \( I \) loops are respectively:

- K loop: \( \text{IDOM}_K = \{-, -, K, -, -\} \) and \( \theta_K = \{I, I+J, x_3, 50, \exists\} \).

- J loop: \( \text{IDOM}_J = \{-, J, K, -, J\} \) and \( \theta_J = \{I, x_2, x_3, 50, \forall\} \).

- I loop: \( \text{IDOM}_I = \{I, J, K, -, J\} \) and \( \theta_I = \{x_1, x_2, x_3, 50, \forall\} \).

<
2.3.3 Data Access Descriptor for a scalar variable

Scalar variables will be treated differently from arrays by defining the Data Access Descriptor for a scalar reference to be simply the name of the scalar variable. All references to the same scalar variable will thus have the same Data Access Descriptor.

2.4 Determining the Data Access Descriptor of an array accessed within a loop

Consider a reference to an n dimensional array A enclosed within m loops:

\[
\begin{align*}
\text{DO } I_1 &= L_1, U_1 \\
\text{DO } I_2 &= L_2, U_2 \\
\vdots \\
\text{DO } I_m &= L_m, U_m \\
A(x_1, x_2, \ldots, x_n) &= \cdots
\end{align*}
\]

where the \( I_k \), \( 1 \leq k \leq m \) are loop induction variables, and \( L_k, U_k \) are the lower and upper bounds of \( I_k \) respectively. \( L_1 \) and \( U_1 \) are integer constants, and \( L_i \) and \( U_i \), \( 2 \leq i \leq m \) are of the form \( p + qI_s \), \( 1 \leq s \leq i - 1 \) where \( p, q \) are signed integer constants. Each \( x_i \), \( 1 \leq i \leq n \), is a subscript expression of the form \( c_0 + \sum_{j=1}^{m} c_j I_j \), where the \( c_j \), \( 1 \leq j \leq m \), are signed integer constants.

The simple section that approximates the accessed portion of the array in the region enclosed by the \( I_1 \) loop is determined as follows. By lemma 1, a simple section in \( n \) dimensions can be described by \( n^2 \) simple boundary pairs of the form \( \alpha \leq \psi(x_1, \ldots, x_n) \leq \beta \). \( n \) of these are of the form \( \alpha \leq x_i \leq \beta \), \( 1 \leq i \leq n \), \( \left( \begin{array}{c} n \\ 1 \end{array} \right) \) are of the form \( \alpha \leq x_i + x_j \leq \beta \), \( 1 \leq i \leq n \), \( i \neq j \), and the remaining \( \left( \begin{array}{c} n \\ 2 \end{array} \right) \) are of the form \( \alpha \leq x_i - x_j \leq \beta \), \( 1 \leq i \leq n \), \( i \neq j \). Since each \( \psi \) is either of the form \( x_i \) or of the form \( x_i \pm x_j \), and each \( x_i \) is a linear function in the loop induction variables, \( \psi \) can also be expressed as a linear function in the loop induction variables \( I_1, I_2, \ldots, I_m \):

\[ \psi = a_0 + \sum_{k=1}^{n} a_k I_k. \]

Given an array reference, we can compute the simple boundary pairs of the array section referenced within the region enclosed by the \( I_1 \) loop by determining the lower
bound \( lb(\psi) \) and the upper bound \( ub(\psi) \) for each of the \( n^2 \) possible \( \psi \). These bounds are given by:

\[
\begin{align*}
\frac{a_0 + \sum_{k=1}^{m} (a^+_k L_k - a^-_k U_k)}{lb(\psi)} & \leq \frac{a_0 + \sum_{k=1}^{m} a_k I_k}{\psi} \leq \frac{a_0 + \sum_{k=1}^{m} (a^+_k U_k - a^-_k L_k)}{ub(\psi)}
\end{align*}
\]

(2.1)

where \( a^+ \) and \( a^- \) are the positive and negative parts of a number, and are defined as follows:

\[
a^+ = \max(a, 0) \quad \text{and} \quad a^- = \max(-a, 0)
\]

The inequality given in (2.1) is a property of linear functions, and is adapted from the following result due to Banerjee [BanSS]:

**Theorem 2.1** If the rectangle \( R \subset \mathbb{R}^n \), defined by \( R = \{(z_1, z_2, \ldots, z_n) : p_1 \leq z_1 \leq q_1, p_2 \leq z_2 \leq q_2, \ldots, p_n \leq z_n \leq q_n\} \) is nonempty, then the minimum and maximum values of the function \( f(z) = \sum a_k z_k \) in \( R \) are \( \sum (a^+_k p_k - a^-_k q_k) \) and \( \sum (a^+_k q_k + a^-_k p_k) \) respectively.

In our case, \( R \) is the iteration space \( R = \{(I_1, I_2, \ldots, I_m) : L_1 \leq I_1 \leq U_1, L_2 \leq I_2 \leq U_2, \ldots, L_m \leq I_m \leq U_m\} \) defined by the loop induction variables \( I_1, I_2, \ldots, I_m \). The array reference maps a portion of the iteration space into memory. The simple section is an approximate description of this portion.

The bounds for a given \( \psi \) cannot be evaluated in a single step using equation (2.1) because the \( L_k, U_k \), \( k \geq 2 \) can be of the form \( p + q I_s \), \( 1 \leq s \leq k - 1 \). \( lb(\psi) \) and \( ub(\psi) \) can therefore be linear functions in the induction variables \( I_1, I_2, \ldots, I_{k-1} \). Equation (2.1) can be applied again to compute the lower and upper bounds of these new linear functions. We can continue in this manner until we are eventually left with linear functions for \( lb(\psi) \) and \( ub(\psi) \) that involve only \( I_1 \). Since both \( L_1 \) and \( U_1 \) are constants, these linear functions will evaluate to constants. These constants are the lower and upper bounds of the function \( \psi \). In the worst case, the procedure to compute the lower and upper bounds of \( \psi \) will require at most \( m \) steps to terminate. Algorithm SIMPLE-SECTION in Figure 2.10 extends the above procedure to com-
Algorithm SIMPLE_SECTION

Input: dimension $n$ of the array, the nesting depth $m$ of the array reference, the bounds $L_h, U_h, \ldots, L_m, U_m$ of the loop induction variables $I_h, I_{h+1}, \ldots, I_m$ and the subscript expressions $x_1, x_2, \ldots, x_n$ of the array reference.

Output: the simple section $S$ that approximates the portion of the array referenced in the region enclosed by the $I_h$ loop.

begin
1. let $\vartheta_{I_h}(x_1, x_2, \ldots, x_n) = \{x_i | \text{DOM}_{I_h}(i) \neq 0\}$, and let $\| \vartheta_{I_h} \| = l$;
   /* $\vartheta_{I_h}$ is the set of $x_i$ that are variant with respect to the $I_h$ loop.
   The simple section consists of $l^2$ simple boundary pairs of the form $\alpha \leq \psi(\vartheta_{I_h}(x_1, \ldots, x_n)) \leq \beta$ (refer to Lemma 2.1, Definition 2.4 and Definition 2.7) */
2. for each possible expression $\psi$ do
3.     express $\psi$ in the form $\psi = a_0 + a_h I_h + \cdots + a_m I_m$, where the $a_k$, $h \leq k \leq m$
   are signed constants;
4.     define $a^L_k = \max(a_k, 0)$ and $a^U_k = \max(-a_k, 0)$, $h \leq k \leq m$;
5.     add the simple boundary pair $LBOUND(h, \psi) \leq \psi \leq UBOUND(h, \psi)$ to $S$;
endfor
6. return $S$
end

procedure LBOUND ($h, \xi$)
begin
1. if $\xi$ does not contain any terms in $I_h, I_{h+1}, \ldots, I_m$
2.     then return $\xi$
else
3.     replace each $I_k, h \leq k \leq m$, in $\xi$ by $(a^L_k L_k - a^U_k U_k)$;
4.     let $\xi'$ be the resulting expression;
5.     return LBOUND($h, \xi'$)
endelse
end

procedure UBOUND ($h, \xi$)
begin
1. if $\xi$ does not contain any terms in $I_h, I_{h+1}, \ldots, I_m$
2.     then return $\xi$
else
3.     replace each $I_k, h \leq k \leq m$, in $\xi$ by $(a^L_k U_k - a^U_k L_k)$;
4.     let $\xi'$ be the resulting expression;
5.     return UBOUND($h, \xi'$)
endelse
end

Figure 2.10 Algorithm to compute the simple section of an array accessed within a loop.
pute the simple section of an array referenced in a region enclosed by any $I_h$ loop, $1 \leq h \leq m$.

Algorithm SIMPLE_SECTION calls procedures LBOUND and UBOUND to compute the lower bound and upper bound of the simple boundary pairs of the simple section. Note that after each recursive call to the procedures LBOUND and UBOUND, the expression $\xi$ has at least one less induction variable than it did just prior to that call. Eventually, $\xi$ will only contain terms in $I_1, \ldots, I_{h-1}$, and the recursion will terminate. In the worst case, if each induction variable is triangular with respect to the induction variable of its containing loop, procedures LBOUND and UBOUND would require $O($nesting depth$)$ recursive calls to complete. For fixed dimensions, the number of possible $\psi$ expressions for a simple section is a small constant (by Lemma 2.1 and Definition 2.4). Algorithm SIMPLE_SECTION thus has a worst-case time complexity of $O($nesting depth$)$ of the array reference with respect to the $I_h$ loop.

2.4.1 Dealing with auxiliary induction variables

An auxiliary induction variable is a variable whose value is some linear function of the basic loop induction variable. Standard techniques can be used to identify families of induction variables for each loop [ASU86]. A family of induction variables consists of the basic loop induction variable and its associated auxiliary induction variables.

For each subscript position of an array reference in a loop, the family of dominant induction variables for that subscript can be determined. IDOM($i$) can be modified to contain the family of dominant induction variables for the $i$th subscript position. This modification will affect the computation of traversal order and reference template fields of the Data Access Descriptor in obvious ways. Henceforth, no distinction is made between the basic and the auxiliary induction variables of a loop. Both kinds will be referred to as “loop induction variables”.
2.4.2 Dealing with symbolic variables

Symbolic variables can occur in the lower and upper bounds of the loop induction variables as well as in the subscript expressions of an array reference. Although the techniques presented here only attempt to be precise up to symbolic execution, it is possible to extend the Data Access Descriptor representation to keep a limited number of symbolic variables. We must deal with two kinds of symbolic variables: inductive symbolic variables and non-inductive (or regular) symbolic variables. Non-inductive symbolic variables are those with values unknown at compile time, or those passed as parameters to the procedure containing the array reference. Inductive symbolic variables are loop induction variables of loops that enclose the region for which the Data Access Descriptor is being computed. These loop induction variables are symbolic within the region. For instance, consider a nest of two loops, with induction variables I and J, with the I loop enclosing the J loop. I and J are inductive symbolic variables within the body of the J loop, I is an inductive symbolic variable in the region described by the J loop, and the region described by the I loop has no inductive symbolic variables.

Symbolic variables can occur in the reference template field and the simple section field of the Data Access Descriptor. Arbitrary expressions involving several symbolic variables need complex representations such as directed acyclic graphs (DAGs). In an actual implementation, this is not a feasible solution due to the need for more space and increased complexity involved in processing the symbolic expressions. Complicated subscript expressions and induction variable bounds are rarely encountered in actual numerical scientific programs. The reason may be that they are usually non-intuitive, making the task of debugging more difficult for the programmer. We can handle simple symbolic expressions in the following manner.

Let \(dllb(i)\) and \(dub(i)\) be the declared lower and upper bounds for the \(i\)th dimension of an array. Some languages, such as Fortran, require the values of these bounds to be explicitly specified in the program so their storage can be allocated statically. Others,
such as Pascal and C, allow arrays to be dynamically allocated. It is assumed that the maximum size of the dimensions of an array are available either from the program itself or supplied upon request by the programmer. We allow the bounds of each simple boundary pair to contain at most two symbolic variables, inductive or non-inductive. Thus, the lower bound $lb(\psi)$ and upper bound $ub(\psi)$ of each $\psi$ must be of the form $p + qY + rZ$, where $p, q, r$ are signed integer constants, and $Y, Z$ are symbolic variables. If some bound cannot be expressed in this form, we can approximate it as follows:

- If $\psi = x_i$, then $dlb(i) \leq x_i \leq dub(i)$ are conservative approximations for its lower and upper bounds.

- If $\psi = x_i + x_j$, then $dlb(i) + dlb(j) \leq x_i + x_j \leq dub(i) + dub(j)$ are conservative approximations for its lower and upper bounds.

- If $\psi = x_i - x_j$, then $dlb(i) - dub(j) \leq x_i - x_j \leq dub(i) - dlb(j)$ are the conservative approximations for its lower and upper bounds.

Algorithm SIMPLE_SECTION can be appropriately modified to compute the above approximations when necessary.

We must also deal with symbolic variables that occur in the reference template vector. If an invariant subscript position has a symbolic expression that cannot be expressed in the form $p + qY + rZ$ where $Y, Z$ are symbolic variables, its corresponding entry in the reference template vector is denoted by "$?$". This indicates that for the region in question, this subscript position is a symbolic constant whose value is unknown.

If symbolic variables are to be allowed in the simple section field of the Data Access Descriptor, algorithms INTERSECT and UNION should be modified to deal with symbolic quantities. In general, it is difficult to determine the minimum or the maximum of two symbolic expressions without resorting to some symbolic execution.
Algorithm INTERSECT can be modified to simply report a non-null intersection whenever symbolic expressions are encountered in the simple boundary pairs of the two simple sections. Since intersection is used primarily as a test for data dependence, this is a conservative decision. Algorithm UNION can be modified to use the conservative approximations described earlier for symbolic bounds of simple boundary pairs. If the two symbolic expressions being compared have the same symbolic variables, then the comparison can be performed without resorting to symbolic evaluation. Complicated symbolic expressions are rarely encountered in actual numerical scientific programs, and the loss of precision may be acceptable in most cases.

Example 2.3:

Figure 2.11 shows some loops $R_1, R_2, R_3$ that contain references to a 2 dimensional array $A$ and the corresponding simple sections described by them (dotted lines represent redundant boundaries). Let the array $A$ be $100 \times 100$ in size. The Data Access Descriptors for each of these regions are given by:

- $R_1$ (Entire array):
  $$\delta_{R_1}(A) = \begin{cases} x_1, x_2 & 1 \leq x_1 \leq 100 \\ 1 \leq x_2 \leq 100 \\ 2 \leq x_1 + x_2 \leq 200 \\ -99 \leq x_1 - x_2 \leq 99 \end{cases} x_1 > x_2$$

- $R'_1$ (Single column):
  $$\delta_{R'_1}(A) = \begin{cases} x_1, I & 1 \leq x_1 \leq 100 \\ x_1 \end{cases}$$

- $R_2$ (Single diagonal):
  $$\delta_{R_2}(A) = \begin{cases} x_1, x_2 & 1 \leq x_1 \leq 100 \\ 2 \leq x_1 + x_2 \leq 200 \\ 0 \leq x_1 - x_2 \leq 0 \end{cases} \{ x_1, x_2 \}$$

- $R'_2$ (Single element):
  $$\delta_{R'_2}(A) = \begin{cases} I, I & 1 \leq x_1 \leq 100 \\ x_1 \end{cases}$$

- $R_3$ (Triangular section):
  $$\delta_{R_3}(A) = \begin{cases} x_1, x_2 & 1 \leq x_2 \leq 100 \\ 2 \leq x_1 + x_2 \leq 200 \\ 0 \leq x_1 - x_2 \leq 99 \end{cases} x_2 > x_1$$

- $R'_3$ (Part of a single row):
  $$\delta_{R'_3}(A) = \begin{cases} I, x_2 & 1 \leq x_2 \leq 100 \\ x_2 \end{cases}$$
2.5 Translation of Data Access Descriptors

In an actual implementation, it may be expensive to compute and store the Data Access Descriptors for each variable with respect to every region that contains the variable. For instance, if a variable is reference is contained within \( m \) loops, the Data Access Descriptor for that variable with respect to each of the loops may be different, so that \( m \) different Data Access Descriptors may have to be computed.

Fortunately, it is sufficient to compute the Data Access Descriptor for a variable with respect to the innermost region that contains it. The Data Access Descriptor can be converted to give the Data Access Descriptor for the variable in terms of the immediately enclosing region. This can be done incrementally, without recomputing the reference template, simple section and traversal order, by a process called translation. The reference template and the simple section fields of the Data Access Descriptor contain sufficient information to perform the translation.

Two kinds of translation are discussed in this section. The first involves translation of the Data Access Descriptor of a region to a loop that contains the region. The
second kind involves translation of the Data Access Descriptor for a variable accessed within a subroutine to the call site of the subroutine.

2.5.1 Translation to a surrounding loop

Consider an array reference $A(x_1, x_2, \ldots, x_n)$. Let $\delta_{R'}(A)$ be the Data Access Descriptor for this reference to $A$ in the region $R'$. Let $I_k$ be the induction variable of the loop immediately surrounding the region $R'$. Let $R$ be the region described by the $I_k$ loop. Algorithm TRANSLATE_TO_LOOP translates $\delta_{R'}(A)$ to $\delta_R(A)$, the Data Access Descriptor for the reference with respect to the region $R$ described by the $I$ loop (Figure 2.12).

The algorithm translates a "$\exists$" in the reference template to a "$\forall$". The "$\exists$" in the reference template indicates that its corresponding subscript position is a symbolic constant whose value is unknown. When this is translated out to an outer region, it is conservative to assume that this subscript position can vary between its declared lower and upper bounds within the new region. This is indicated by the "$\forall$" symbol in the reference template.

2.5.2 Translation from a subroutine to a call site

Translation of a Data Access Descriptor from within a subroutine to its call site is more complex. The complexity of the translation process depends on the parameter passing rules defined by the programming language. In particular, array variables that are passed as parameters can pose some problems, especially if the formal has different dimensions than its corresponding actual parameter. Algorithm TRANSLATE_TO_CALL translates a Data Access Descriptor for an array reference within a subroutine to the call site of the subroutine. Translation of the Data Access Descriptor of a scalar variable is trivial; the name of the scalar is replaced with the actual name bound to it at the call site.

It is assumed that for every call site in the program, we are given a binding function $b$. The binding function maps formal parameter names in the subroutine to actual
Algorithm **TRANSLATE_TO_LOOP**

**Input:** The Data Access Descriptor $\delta_{R'}(A) = (\theta_{R'}|S_{R'}|\tau_{R'})$ for A in the region $R'$.

**Output:** The Data Access Descriptor $\delta_R(A) = (\theta_R|S_R|\tau_R)$ for A with respect to the region $R$ (i.e., the region described by the $I_k$ loop).

begin

/* Computation of $\theta_R$. Refer to Definition 2.7 */
for $i = 1$ to $n$

if $\theta_{R'}(i)$ contains an $I_k$ term then

/* the ith subscript is variant in $R$ */
replace $\theta_{R'}(i)$ with the variable "$x_i$" ;
add $x_i$ to NEW_VARIANTS ;
endif

else if $\theta_{R'}(i)$ is variant in $R'$
then add $x_i$ to OLD_VARIANTS ;
else if $\theta_R(i) = "3"$ or $\theta_R(i) = "6"$ then

if $\theta_R(i) = "3"$ then set $\theta_R(i) = "6"$ ;
add $x_i$ to UNKNOWN_SYMBOLICS ;
endelse

else add $x_i$ to ININVARIANTs ;
endfor

$\tau_R = \tau_{R'} \times \text{NEW_VARIANTS} ;$ /* Computation of $\tau_R$ */
$S_R = S_{R'} ;$ /* Computation of $S_R$ */
for all $x_i \in \text{OLD_VARIANTS}$ do

for all $x_j \in \text{NEW_VARIANTS}$ do

add to $S_R$ the simple boundary pairs

$\text{LBOUND}(k, (lb(x_i) \pm \theta_R(j))) \leq x_i \pm x_j \leq \text{UBOUND}(k, (ub(x_i) \pm \theta_R(j))) ;$
endfor
endfor

for all $x_j \in \text{NEW_VARIANTS}$ do

add $\text{LBOUND}(k, \theta_R(j)) \leq x_j \leq \text{UBOUND}(k, \theta_R(j))$ to $S_R ;$
endfor

for all $x_i, x_j, i \neq j \in \text{NEW_VARIANTS}$ do

add to $S_R$ the simple boundary pairs

$\text{LBOUND}(k, \theta_R(i) \pm \theta_R(j)) \leq x_i \pm x_j \leq \text{UBOUND}(k, \theta_R(i) \pm \theta_R(j)) ;$
endfor

for all simple boundary pairs $\alpha \leq \psi \leq \beta \in S_R$ do

if $\alpha$ contains an $I_R$ term then replace $\alpha$ with $\text{LBOUND}(k, \alpha) ;$
if $\beta$ contains an $I_R$ term then replace $\beta$ with $\text{UBOUND}(k, \beta) ;$
endfor

return $(\theta_R|S_R|\tau_R)$
end

**Figure 2.12** Algorithm to translate a Data Access Descriptor in terms of an enclosing loop.
Algorithm TRA\linebreak3NLATE\linebreak3TO\linebreak3C\linebreak3ALL

Input: The Data Access Descriptor \( \delta(A) = (\theta|S|\tau) \) for an \( n \) dimensional array \( A \), the binding function \( b \) that maps the formals to actuals at the call site, and the vector \( \text{OFFSET} \) of subscript expressions of \( b(A) \) passed in at the call site.

Output: The Data Access Descriptor \( \delta(b(A)) = (\theta'|S'|\tau') \).

begin

\[ */* \text{ Case 1: } A \text{ and } b(A) \text{ are of the same dimension and size. } */ \]
1 \( \theta' = \theta \) with any symbolic variable \( X \in \theta \) replaced by \( b(X) \);
2 for all simple boundaries \( \alpha \leq x_i \leq \beta \in S \)
3 \quad add \( \alpha + \text{OFFSET}(i) \leq x_i \leq \beta + \text{OFFSET}(i) \) to \( S' \);
endfor
4 for all simple boundaries \( \alpha \leq x_i \pm x_j \leq \beta \in S \)
5 \quad add \( \alpha + \text{OFFSET}(i) \pm \text{OFFSET}(j) \leq x_i \pm x_j \leq \beta + \text{OFFSET}(i) \pm \text{OFFSET}(j) \) to \( S' \);
endfor
6 \( \tau' = \tau \) with any symbolic strides replaced by their formal names;
7 \( */* \text{ Case 2: } A \text{ is a linear array (i.e., of dimension 1). } */ \)
8 let \( \delta(A) \) be of the form \( \{ x_1 \mid \alpha \leq x_1 \leq \beta \} \);
9 and let the size of \( b(A) \) be \( N \times N \times \cdots \times N \) (\( N^n \) elements in total);
10 \( \theta'(1) = "x_1"; \)
11 for \( i = 2 \) to \( n \) do
12 \quad if \( \beta - \alpha + 1 > N^{i-1} - 1 - \sum_{m=1}^{i-1} N^{m-1}(\text{OFFSET}(m) - 1) \)
13 \quad \quad then \( \theta'(i) = "x_i"; \)
14 \quad else \( \theta'(i) = \text{OFFSET}(i); \)
endfor
15 let \( \theta'(j), 1 \leq j \leq k \) be the variants in the reference template computed above;
16 for \( i = 1 \) to \( k \) do
17 \quad add to \( S' \) the simple boundary pair
18 \quad \quad \text{OFFSET}(i) \leq x_i \leq (\beta - \alpha + 1 - N^{i-1} - 1 - \sum_{m=1}^{i-1} N^{m-1}(\text{OFFSET}(m) - 1))/N
19 endfor
20 the diagonal boundary pairs of \( S' \) can be computed as follows:
21 \( lb(x_i) + lb(x_j) \leq x_i + x_j \leq ub(x_i) + ub(x_j) \) and
22 \( lb(x_i) - ub(x_j) \leq x_i - x_j \leq ub(x_i) - lb(x_j) \) \( \forall i, j, 1 \leq i, j \leq k, i \neq j; \)
23 \( \tau' = x_1 \gg x_2 \gg \cdots \gg x_k; \)
end

Figure 2.13 Algorithm to translate a Data Access
Descriptor from a subroutine to its call site.
parameter names at the call site. If \( X \) is a formal variable in the subroutine, the actual name bound to it is denoted by \( b(X) \). If \( X \) is a global or COMMON variable (or passed by reference), \( b(X) = X \). For every parameter that is an array variable it is assumed that a vector OFFSET is available of length equal to the dimension of the array. The \( i \)th element of OFFSET is the \( i \)th subscript expression of the array element that is passed as a parameter to the subroutine. If only the name of the array variable appears at the call site, the subscripts are assumed to be the declared lower bounds of the array (i.e., the array element passed is assumed to be the first element of the array).

Algorithm TRANSLATE_TO_CALL in Figure 2.13 performs the translation of a Data Access Descriptor in a subroutine to the call site. The algorithm only considers two cases:

- **Case 1:** The first case is when the array variable \( A \) and its actual parameter \( b(A) \) are of the same dimension and size. This implies that the shape of the accessed portion of \( A \) within the subroutine is preserved upon translation to the call site. In other words, the simple section of \( b(A) \) has the same shape as the simple section of \( A \). The translation is accomplished by replacing all symbolic variables \( X \in \delta(A) \) by \( b(X) \), and adding the appropriate OFFSETs to the lower and upper bounds of the simple boundary pairs describing the simple section of \( A \).

- **Case 2:** The second case is when the array \( A \) within the subroutine is a linear array (i.e., a vector), but \( b(A) \) is a higher dimensional array. Clearly, shapes are not preserved upon translation, because a one dimensional section of \( A \) must be translated to a multidimensional section of \( b(A) \).

Let the array \( b(A) \) be of size \( N \times N \times \cdots \times N \) (i.e., \( N^n \) elements). The translation done in algorithm TRANSLATE_TO_CALL is based on the following observation. Assuming column major storage for the multidimensional array
\( b(A) \), the first \( N \) elements of \( A \) will map to the first column of \( b(A) \), the first \( N^2 \) elements of \( A \) will map to the first \( x_1 x_2 \) plane of \( b(A) \), and so on. Thus, the first \( N \) elements of the vector \( A \) translate to a single column of \( b(A) \); access in \( b(A) \) is along dimension \( x_1 \) only, and the other dimensions are invariant. An access of more than \( N \) but less than \( N^2 \) consecutive elements of \( A \) translate to accesses along dimensions \( x_1 \) and \( x_2 \) in \( b(A) \). That is, we have “spilled over” onto the next dimension. The translation must take into account the element of \( A \) that the accesses start from; this is given by the OFFSET vector. Lines 8 through 21 of algorithm TRANSLATE_TO_CALL generalize this concept to perform the translation of \( \delta(A) \) to \( \delta(b(A)) \).

The TRANSLATE_TO_CALL algorithm shown in Figure 2.13 assumes that the strides along the dimensions of \( A \) within the subroutine are all unit strides. While this does not have any real consequence for the first case, the second case depends on this assumption being true. The problem with non-unit strides is discussed in detail in a subsequent section, and further discussion of this issue is postponed until then.

**Example 2.4:**

Let us now examine a realistic program segment and compute the Data Access Descriptors for variables at various points in the program. The program in Figure 2.14 is adapted from the LINPACK subroutine DGEFA, which factors a real matrix by Gaussian Elimination [DMBS79]. Subroutine LUDCOMP takes as input the 2 dimensional matrix \( A \) to be factored and the order \( N \) of the matrix. It returns the factored matrix \( A \) in the form of an upper triangular matrix and the multipliers that were used to obtain it. Also returned is an integer vector IPVT, of size \( N \), containing the pivot indices.

The Data Access Descriptors for the array variable \( A \) are computed at the points denoted by the \( * \) line numbers in subroutine LUDCOMP. The subscript on \( \delta \) is used to denote the line number for which it is computed. The region can be discerned from
subroutine LUDCOMP (A, N, IPVT)
1* DO K = 1, N-1
2    L = IDAMAX (N-K, A(K,K), 1)
3    IPVT(K) = L
4*   swap (A(IPVT(K),K), A(K,K))
5    C = 1/A(K,K)
6*   call DSCAL (N-K, C, A(K+1,K), 1)
7*   DO J = K+1, N
8*    swap (A(IPVT(K),J), A(K,J))
9    T = A(L,J)
10*   call DAXPY (N-K, T, A(K+1,K), 1, A(K+1,J), 1)
11 ENDDO
12 ENDDO
end

function IDAMAX (M, DX, INCX)
13* find P, such that |DX(P)| = max(|DX(1)|, ..., |DX(M)|)
14 RETURN P
end

subroutine DSCAL (M, DA, DX, INCX)
15    J = 1
16* DO I = 1, M
17    DX(J) = DX(J) * DA
18    J = J + INCX
19 ENDDO
end

subroutine DAXPY (M, DA, DX, INCX, DY, INCY)
20    J = 1
21    K = 1
22* DO I = 1, M
23    DY(J) = DY(J) - DA * DX(K)
24    J = J + INCY
25    K = K + INCX
26 ENDDO
end

Figure 2.14  LU Decomposition program.
the context. The Data Access Descriptors for the arrays DX and DY within function IDAMAX and the subroutines DSCAL and DAXPY are:

- In function IDAMAX:
  \[ \delta_{13}^r(DX) = \left\langle x_1 \mid 1 \leq x_1 \leq M \right| x_1 \right\rangle \]

- In subroutine DSCAL:
  \[ \delta_{16}^w(DX) = \delta_{16}^r(DX) = \left\langle x_1 \mid 1 \leq x_1 \leq M \right| x_1 \right\rangle \]

- In subroutine DAXPY:
  \[ \delta_{22}^w(DY) = \delta_{22}^r(DY) = \left\langle x_1 \mid 1 \leq x_1 \leq M \right| x_1 \right\rangle \]
  \[ \delta_{22}^r(DX) = \left\langle x_1 \mid 1 \leq x_1 \leq M \right| x_1 \right\rangle \]

The Data Access Descriptor for A at line 2 of LUDCOMP can be computed by translating \( \delta^r(DX) \) in function IDAMAX outwards to the call site at line 2.

- At line 2 of LUDCOMP:
  \[ \delta_2^s(A) = \text{ref}(\delta_{13}^r) = \left\langle x_1, K \mid K + 1 \leq x_1 \leq N \right| x_1 \right\rangle \]

Similarly, we can get the Data Access Descriptors for A at lines 6 and 10 by translating the Data Access Descriptors in DSCAL and DAXPY outwards to their call sites in LUDCOMP.

- At line 6 of LUDCOMP:
  \[ \delta_6^w(A) = \delta_6^s(A) = \text{ref}(\delta_{16}^w) = \left\langle x_1, K \mid K + 1 \leq x_1 \leq N \right| x_1 \right\rangle \]

- At line 10 of LUDCOMP:
  \[ \delta_{10}^w(A) = \text{ref}(\delta_{22}^w) = \left\langle x_1, J \mid K + 1 \leq x_1 \leq N \right| x_1 \right\rangle \]
  \[ \delta_{10}^r(A) = \text{ref}(\delta_{22}^r) = \left\langle x_1, K \mid K + 1 \leq x_1 \leq N \right| x_1 \right\rangle \cup \left\langle x_1, J \mid K + 1 \leq x_1 \leq N \right| x_1 \right\rangle \]

To compute the Data Access Descriptor for A in the region enclosed by the loop at line 7, we must translate \( \delta_8 \) and \( \delta_{10} \) outwards to the loop header at line 7. The derivation of the write component of the Data Access Descriptor (i.e., \( \delta_w(A) \)) is shown below.
The read component $\delta^r(A)$ can be derived in a similar manner.

\[
\begin{align*}
\delta^w_4(A) &= \langle \exists, K \mid - \mid - \rangle \cup \langle K, K \mid - \mid - \rangle \\
\delta^w_5(A) &= \langle \exists, J \mid - \mid - \rangle \cup \langle K, J \mid - \mid - \rangle \\
\delta^w_7(A) &= \uparrow (\delta^w_4(A) \cup \delta^w_5(A)) \\
&= \uparrow \left( \langle \exists, J \mid - \mid - \rangle \cup \langle K, J \mid - \mid - \rangle \cup \langle z_{1,J} \mid K+1 \leq z_{1,J} \leq N \mid z_{1,J} \rangle \right) \\
&= \langle \forall, x_2 \mid K+1 \leq x_2 \leq N \mid x_2 \rangle \cup \langle K, x_2 \mid K+1 \leq x_2 \leq N \mid x_2 \rangle \\
&\quad \cup \langle x_1, x_2 \mid 2(K+1) \leq x_1 + x_2 \leq 2N \mid x_1> x_2 \rangle \\
&\quad \cup \langle x_1, x_2 \mid K+1 \leq x_1 \leq x_2 \leq N \mid x_1> x_2 \rangle \\
&\quad \cup \langle x_1, x_2 \mid K+1 \leq x_1 \leq x_2 \leq N \mid x_1> x_2 \rangle \\
\end{align*}
\]

By translating $\delta^w_4(A), \delta^w_5(A)$ and $\delta^w_7(A)$ outwards again, the Data Access Descriptor for $A$ in the region enclosed by the "DO K" loop at line 1 of LUDCOMP can be computed.

\[
\begin{align*}
\delta^w_5(A) &= \uparrow (\delta^w_4(A) \cup \delta^w_5(A) \cup \delta^w_7(A)) \\
&= \uparrow \left( \langle \exists, K \mid - \mid - \rangle \cup \langle K, K \mid - \mid - \rangle \cup \langle z_{1,K} \mid K+1 \leq z_{1,K} \leq N \mid z_{1,K} \rangle \right) \\
&\quad \cup \langle \forall, x_2 \mid K+1 \leq x_2 \leq N \mid x_2 \rangle \cup \langle K, x_2 \mid K+1 \leq x_2 \leq N \mid x_2 \rangle \\
&\quad \cup \langle x_1, x_2 \mid 2(K+1) \leq x_1 + x_2 \leq 2N \mid x_1> x_2 \rangle \\
&\quad \cup \langle x_1, x_2 \mid K+1 \leq x_1 \leq x_2 \leq N \mid x_1> x_2 \rangle \\
&\quad \cup \langle x_1, x_2 \mid K+1 \leq x_1 \leq x_2 \leq N \mid x_1> x_2 \rangle \\
&\quad \cup \langle x_1, x_2 \mid 1 \leq x_2 \leq N-1 \mid x_2 \rangle \cup \langle x_{1,x_2} \mid 1 \leq x_1 \leq N-1, 1 \leq x_2 \leq N-1 \mid x_{1,x_2} \rangle \\
&\quad \cup \langle x_{1,x_2} \mid 2 \leq x_2 \leq N-1, 1 \leq x_2 \leq N-1 \mid x_{1,x_2} \rangle \cup \langle x_{1,x_2} \mid 2 \leq x_2 \leq N \mid x_{1,x_2} \rangle \\
&\quad \cup \langle x_{1,x_2} \mid 3 \leq x_2 \leq 2N \mid x_{1,x_2} \rangle \cup \langle \forall, x_2 \mid 2 \leq x_2 \leq N \mid x_2 \rangle \\
&\quad \cup \langle x_{1,x_2} \mid 1 \leq x_2 \leq N-1 \mid x_{1,x_2} \rangle \cup \langle x_{1,x_2} \mid 1 \leq x_2 \leq N \mid x_{1,x_2} \rangle \\
&\quad \cup \langle x_{1,x_2} \mid 1 \leq x_2 \leq N \mid x_{1,x_2} \rangle \cup \langle x_{1,x_2} \mid 2 \leq x_2 \leq N \mid x_{1,x_2} \rangle \\
&\quad \cup \langle x_{1,x_2} \mid 4 \leq x_2 \leq 2N \mid x_{1,x_2} \rangle \cup \langle x_{1,x_2} \mid 2 \leq x_2 \leq N \mid x_{1,x_2} \rangle \\
\end{align*}
\]

The sections represented by $\delta^w_5(A)$ are shown in Figure 2.15. The dotted boundaries are the boundaries of the complete array $A$. The dashed lines represent redundant simple boundaries. \(\square\)
\[ \delta^\nu(A) = \]

Figure 2.15 Data Access Descriptors for the stores to A within LUDCOMP.

2.6 Dealing with non-unit step sizes

In all of the examples thus far, the step sizes of loop induction variables were ignored when computing the Data Access Descriptor. We implicitly made the assumption that every point of the simple section described by a Data Access Descriptor is actually accessed. This assumption, though conservative in most cases, can give incorrect results for Data Access Descriptors that are translated from a subroutine to its call site. For example, consider the subroutine DAXPY in the LUDCOMP program. One of the parameters to DAXPY is INCY, which is used as a step size for the auxiliary loop induction variable J on line 22 (see Figure 2.15). When INCY = 1, the access to the M elements of DY within DAXPY translate to an access of N-K consecutive elements of the array A at line 10, starting from the A(K+1,K)th element. Assuming
column major storage for Fortran, this is an access of N-K elements along the Kth column of the array A.

Suppose the actual parameter bound to the formal INCY at the call site on line 10 is N instead of 1. Within DAXPY, M elements of the array DY are accessed at a distance of INCY apart. This translates to an access of N-K elements of the array A, starting at the A(K+1,K)th element, each separated by N elements from the previous access. This is an access of N-K elements along the (K+1)th row of A. If we ignore the step size INCY and assume the step size to be 1, the translation of the Data Access Descriptor from DAXPY to its call site at line 10 of LUDCOMP cannot detect this row-wise access. It translates the access to DY in DAXPY as an access of N-K elements along the Kth column of A, which is incorrect.

Thus, the strides of loop induction variables cannot be ignored. The Data Access Descriptor representation can be extended to keep information about strides. Consider a loop nest of the form:

```plaintext
DO I1 = L1, U1, S1
  DO I2 = L2, U2, S2
     ...                      
     DO Im = Lm, Um, Sm
     A(x1, x2, ..., xn) = ...  
  ENDDO                       
ENDDO
```

where the loop induction variables, their lower and upper bounds and the subscript expressions of the array reference are defined as in secton 2.4. S1, S2, ..., Sm are the loop strides associated with their respective loop induction variables. These loop strides must be signed integer constants. They should not be expressions involving more than one symbolic variable, and they may not involve another loop induction variable. These restrictions are reasonable because most loop strides encountered in numerical scientific programs conform to these restrictions. If a stride Sk is negative, the lower and upper bounds Lk and Uk of the induction variable Ik are swapped before applying algorithm SIMPLE_SECTION or algorithm TRANSLATE_TO_LOOP. The IDOM vectors for the reference are modified to keep the strides of the dominant
induction variables in addition to the dominant induction variables themselves. When computing the traversal order of a section, the superscript on \( x_i \) is used to denote the stride along the \( x_i \) dimension. The stride along a particular dimension \( x_i \) is taken to be the stride of the dominant induction variable of the \( i \)th subscript position in the current region. The default stride is 1. For example, the Data Access Descriptor for the store into DY in subroutine DAXPY is:

\[
\delta_{22}^{(n)}(DY) = \left\{ x_i \mid 1 \leq x_i \leq M \mid x_i^{NCY} \right\}
\]

If the stride along a particular dimension is an unknown symbolic constant, it is represented by a "*" in the superscript.

**Example 2.5:**

In the following reference to \( \mathbf{A} \),

\[
\begin{align*}
&\text{DO } l = 1, 100, 2 \\
&\text{DO } J = 100, 1, -4 \\
&\text{DO } K = 1, 100 \\
&\text{DO } L = 1, 100, P \\
&\text{A}(l+J, l, K+2, L) = \cdots \\
&\text{ENDDO}
\end{align*}
\]

the traversal order field of the Data Access Descriptor for \( \mathbf{A} \) with respect to the \( I \) loop is \( x_4^* \succ x_3 \succ x_1^{-4} \succ x_2^2 \). \(<\)

Algorithm TRANSLATE_TO_CALL must be modified to check if the size of the offset added to the stride along a dimension exceeds the declared upper bound of the array passed as an actual parameter. If so, then the translation must take this into account when computing the simple section accessed at the call site.

### 2.6.1 Intersection and union of Data Access Descriptors

Intersection of Data Access Descriptors is used as a test for data dependence. Procedure TESTDEPENDENCE in Figure 2.16 is a definition of Data Access Descriptor intersection. The procedure takes two Data Access Descriptors for a variable and returns their "intersection". If there is no data overlap between the two Data Access
Descriptors, the procedure returns $\phi$, indicating that the two Data Access Descriptors are independent.

If there are multiple references to the same array variable within a region, a Data Access Descriptor is computed for each reference and unioned together to give a single Data Access Descriptor that summarizes all accesses to the array within that region. Procedure DAD\_UNION in Figure 2.17 returns the union of two Data Access Descriptors for the same array variable. The time complexity of Data Access Descriptor union and intersection is determined by the complexity of simple section union and intersection, which is a constant time.

### 2.7 The Data Window revisited

The previous sections described the Data Access Descriptor representation, and how it can be used to summarize the portion of an array accessed within a region. We are now equipped to give the formal definition of a Data Window.

**Definition 2.9 (Data Window):** Given a region $R$ of a program, its Data Window, denoted by $\Delta(R)$ is given by

$$\Delta(R) = \{ \delta_R(X) \mid X \text{ is referenced in } R \text{ and is global to } R \}$$

where $\delta_R(X)$ is the Data Access Descriptor for the variable $X$ in $R$.

The above definition indicates that variables that are “local” to $R$ do not contribute to its Data Window since they are not visible outside the region. $\delta_R(X)$, for each variable $X$, consists of two components, $\delta^w_R(X)$ and $\delta^r_R(X)$. Thus, the Data Window also has two components, the write component $\Delta^w(R) = \{ \delta^w_R(X) \}$ and the read component $\Delta^r(R) = \{ \delta^r_R(X) \}$.

#### 2.7.1 Intersection and union of Data Windows

A data dependence exists between two regions if the intersection of their Data Windows is non-empty. Thus, intersection of Data Windows can be used to check for depen-
Procedure \textsc{Test Dependence} ($\delta_1(A)$, $\delta_2(A)$)

/* compute $\delta(A) = \delta_1(A) \cap \delta_2(A)$. Return $\phi$ if the
intersection is empty (i.e., $\delta_1(A)$ and $\delta_2(A)$ are independent). */
1 let $\delta_1(A) = \left< e_1 \mid s_1 \mid r_1 \right>$ and $\delta_2(A) = \left< e_2 \mid s_2 \mid r_2 \right>$;
2 let the dimension of $A$ be $n$, and let $\delta(A) = \left< e \mid s \mid r \right>$;
/* Computation of $\theta$ */
3 for $i = 1$ to $n$ do
4 if $\theta_1(i) = \theta_2(i)$
5 then $\theta(i) = \theta_1(i)$
6 else if both $\theta_1(i)$ and $\theta_2(i)$ are invariant
7 then $\theta(i) = \text{"\text{3}"}$;
8 else if only one of them is invariant then
9 suppose $\theta_1(i)$ is invariant and $\theta_2(i)$ is variant;
10 there must be a simple boundary pair $\alpha \leq \beta \in S_2$;
11 check if $\theta_1(i)$ lies between $\alpha$ and $\beta$;
12 if $\theta_1(i)$ is outside this range
13 then return $\phi$;
14 else if $\theta_1(i)$ is within this range
15 then $\theta(i) = \theta_1(i)$;
16 else if it is not possible to determine whether or not $\alpha \leq \theta_1(i) \leq \beta$
17 then $\theta(i) = \text{"\text{3}"}$;
endelse
18 endfor
/* Computation of $S$ */
19 $S = S_1 \cap S_2$; (see algorithm \textsc{Intersect})
20 if $S = \phi$ then return $\phi$;
/* Computation of $\tau$ */
21 if $\tau_1 \equiv \tau_2$
22 then $\tau$ = the "matched portion" of $\tau_1$ and $\tau_2$;
23 else $\tau = \text{"*"}$;
24 return $\delta(A)$

\textbf{Figure 2.16} Procedure to test for data dependence (Data Access Descriptor intersection).
Procedure DAD_UNION ($\delta_1(A)$, $\delta_2(A)$)

/* compute $\delta(A) = \delta_1(A) \cup \delta_2(A)$. */
1 let $\delta_1(A) = \langle \theta_1 \mid s_1 \mid \tau_1 \rangle$ and $\delta_2(A) = \langle \theta_2 \mid s_2 \mid \tau_2 \rangle$;
2 let the dimension of $A$ be $n$, and let $\delta(A) = \langle \theta \mid s \mid \tau \rangle$;
3 /* Computation of $S$ */
4 $S = \text{simp}(S_1 \cup S_2)$; (see algorithm UNION)
5 /* Computation of $\theta$ */
6 for $i = 1$ to $n$ do
7      if $\theta_1(i) = \theta_2(i)$
8          then $\theta(i) = \theta_1(i)$
9      else if both $\theta_1(i)$ and $\theta_2(i)$ are invariant
10          then $\theta(i) = \text{"\textbackslash"}$;
11      else if only one of them is invariant then
12          suppose $\theta_1(i)$ is invariant and $\theta_2(i)$ is variant;
13          there must be a simple boundary pair $\alpha \leq x_i \leq \beta \in S_2$;
14          if $\theta_1(i)$ lies outside $\alpha$ and $\beta$ then
15              extend the range of the boundary pair $x_i$ in $S$ to include $\theta_1(i)$;
16              recompute the lower and upper bounds of all simple boundary
17              pairs in $S$ that involve $x_i$;
18          endif
19      else if it cannot be determined whether or not $\alpha \leq \theta_1(i) \leq \beta$
20          then $\theta(i) = \text{"\textbackslash"}$;
21     endelse
22 endfor
3 /* Computation of $\tau$ */
4 if $\tau_1 \equiv \tau_2$
5     then $\tau$ = the "matched portion" of $\tau_1$ and $\tau_2$;
6 else $\tau$ = "*";
7 return $\delta(A)$

Figure 2.17 Procedure to compute the union of two Data Access Descriptors.
dence between regions. The intersection of two Data Windows is defined as follows:

**Definition 2.10 (Data Window Intersection):** Given two regions $R_1$ and $R_2$ and their corresponding Data Windows $\Delta(R_1)$ and $\Delta(R_2)$, the intersection of the Data Windows is given by

$$\Delta(R_1 R_2) = \Delta(R_1) \cap \Delta(R_2) = \{\delta_{R_1}(X) \cap \delta_{R_2}(X)\}$$

where each component of $\Delta(R_1)$ is intersected with both the read and the write components of $\Delta(R_2)$. $\Delta(R_1 R_2)$ is called the *data overlap* between $R_1$ and $R_2$.

If the data overlap $\Delta(R_1 R_2)$ is empty, it means there is no data dependence between regions $R_1$ and $R_2$. Since each Data Window consists of two components, and intersection between dissimilar components are allowed, the Data Overlap $\Delta(R_1 R_2)$ will consist of four components:

- $\Delta^w^w(R_1 R_2) = \Delta^w(R_1) \cap \Delta^w(R_2)$, called the *write-read* dependence.
- $\Delta^r^w(R_1 R_2) = \Delta^r(R_1) \cap \Delta^w(R_2)$, called the *read-write* dependence.
- $\Delta^w^r(R_1 R_2) = \Delta^w(R_1) \cap \Delta^r(R_2)$, called the *write-write* dependence.
- $\Delta^r^r(R_1 R_2) = \Delta^r(R_1) \cap \Delta^r(R_2)$, called the *read-read* dependence.

When both regions consist of single statements, the above dependences are called *true, anti, output* and *input* dependences respectively.

If the two regions are two different iterations of the same loop, a dependence between them is referred to as a *loop-carried dependence*. A region can be constructed by merging together a sequence of statements. The Data Window of the resulting region will be the union of the Data Windows of its component statements. We can take this a step further and merge two regions to form a larger region. The Data Window of this larger region will be the union of the Data Windows of its component regions. Union of Data Windows is defined as follows:
Definition 2.11 (Data Window union): Given two regions $R_1$ and $R_2$ and their corresponding Data Windows $\Delta(R_1)$ and $\Delta(R_2)$, the union of the Data Windows is a Data Window

$$\Delta(R) = \Delta(R_1) \cup \Delta(R_2) = \{\delta^{R_1}(X) \cup \delta^{R_2}(X)\}$$

where each component of $\Delta(R_1)$ is unioned with its corresponding component of $\Delta(R_2)$.

$\Delta(R)$ is the Data Window of the region $R$ obtained by merging the regions $R_1$ and $R_2$. $\Delta(R)$ consists of two components, the write and the read component.

2.8 Dependence testing using Data Windows

A non-empty intersection between two Data Access Descriptors of an array variable is only a sufficient condition for establishing the existence of a data dependence. The necessary condition requires that the intersecting region contain at least one point with integer coordinates, since the subscripts of an array reference can only have integral values. Although the intersecting region describes the space of common array references, the points with non-integer coordinates in the intersection do not correspond to a reference. Thus, the intersection is empty unless it contains a point with integer coordinates.

However, algorithm INTERSECT is correct in reporting a non-empty intersection without actually checking to see if the intersection contains any points with integer coordinates. The following important property about simple sections allows us to treat simple section intersection as a sufficient and necessary criterion for assuming the existence of a dependence:

Lemma 2.4 A non-null intersection between any two simple sections $S_1$ and $S_2$ of the same dimension must contain at least one point with integer coordinates.

Proof: A simple section consists of simple boundary pairs of the form $\alpha \leq x_i \leq \beta$ and $\alpha \leq x_i \pm x_j \leq \beta$, where $\alpha$ and $\beta$ are integer constants or expressions that
evaluate to integers. Consider a point $P$ at the vertex of an $n$ dimensional simple section. Clearly, the point $P$ must have integer coordinates because it must satisfy all of the simple boundaries that intersect at $P$. Thus, all vertices of a simple section have integer coordinates. By definition, every simple section is a convex polytope. Intersecting convex polytopes have the property that one or more vertices of one will be contained in the other. Hence, when two simple sections $S_1$ and $S_2$ intersect, one or more vertices of $S_1$ will be contained in $S_2$ and vice versa. The vertices of the intersecting region will therefore have integer coordinates. This proves the statement of the lemma. □

2.8.1 Programmer supplied assertions

Often, it is necessary to make conservative assumptions when complex subscript expressions or loop induction variable bounds are encountered. Earlier, we discussed how to handle symbolic variables in the Data Access Descriptor representation. When an invariant symbolic expression is encountered at a particular subscript position, a "∃" is put in its corresponding position to denote that the constant value at this subscript position is unknown. When a Data Access Descriptor containing a "∃" is translated outwards to a containing region, the subscript with the "∃" is assumed to vary between the declared lower and upper bounds for that subscript position of the array, and this is represented by a "∀" symbol. This is illustrated in the Data Access Descriptor $δ_8^y(A)$ in the LUDCOMP program (Example 2.4). The reference to $A(IPVT(K),J)$ on line 8 of LUDCOMP has the Data Access Descriptor $⟨∃,J | - | -⟩$, indicating that we have no information about the value of IPVT(K). When this is translated outwards to the loop at line 7, the resulting Data Access Descriptor is $⟨∀,z_2 | K + 1 ≤ z_2 ≤ N | z_2⟩$. We could assume that the first subscript position can have values anywhere between the declared lower and upper bounds of $A$. This is done when translating the Data Access Descriptor outwards again to the loop at line 1.
However, if we had the information that IPVT is actually a permutation of the values of K, the translation would be more precise. For example, if the programmer gave the information that the first subscript position can be considered as "K", then the outward translation of this Data Access Descriptor would give a triangular section instead of the rectangular section computed as a result of conservative translation. Thus, the accuracy of dependence testing can be improved in some cases by programmer interaction.

2.8.2 Implementing dependence testing

In an actual implementation, there is a choice between keeping a Data Access Descriptor for each reference of an array in a region and unioning all of them into a single Data Access Descriptor for that array over the whole region. In general, it may not be a good idea to union the Data Access Descriptors for all references to the same variable within a region, because the imprecision introduced during successive unions has a cumulative effect. This may lower the accuracy of dependence testing between two regions. Maintaining the Data Access Descriptor for every reference to a variable in a region will improve the accuracy of dependence testing, at the expense of increased storage requirements. An effective compromise between these two extremes is to maintain only those Data Access Descriptors for an array variable which describe accesses to distinct sections. This information can be maintained as a list. Before a Data Access Descriptor for another reference to the same variable is added to the list, we check to see if the section described by it is already included in the list. It is added to the list only if it describes a section that is not already in the list. Thus, if two different references to the same array in a region access the same section, only one Data Access Descriptor is maintained to represent both references. To check for a dependence between two regions on a given array variable, the list of Data Access Descriptors for the variable in both regions are compared. If some Data
Access Descriptor on one list intersects with any Data Access Descriptor of the other list, a dependence on the variable is assumed to exist between the two regions.

2.9 Inadequacies of the Data Access Descriptor representation

The Data Access Descriptor is tuned to be an efficient mechanism for summarizing data access information in numerical scientific programs. It may not be very suitable for other kinds of applications. In designing this representation, we had to sacrifice some precision for speed and easy implementation. The imprecision does not give incorrect results because we always err on the conservative side. This imprecision should be acceptable in a large majority of numerical scientific programs, although no empirical studies have been performed. This section discusses the nature of the imprecision in the Data Access Descriptor representation.

The first source of imprecision is in the handling of symbolic variables in the Data Access Descriptor (section 2.4.2). Any system that does a static analysis of the program must deal with symbolic variables. Comparing symbolic variables and performing arithmetic operations on them almost always requires some form of symbolic execution. This is a complex procedure that is difficult to realize in a real system and often produces unsatisfactory results. Thus, static analysis tools seldom use symbolic evaluation techniques, opting instead to make conservative assumptions when symbolic expressions are encountered. In this respect, the decision to make the Data Access Descriptor representation be precise only up to symbolic execution seems justifiable. When a complex symbolic expression is encountered in the subscript position of an array variable, we assume that it can have any value between its declared lower and upper bounds. In languages where these bounds are unknown at compile-time, we must assume that any integer value can occur in that subscript position. These assumptions severely limit the precision of the algorithms that compute intersection and hence the accuracy with which we can establish the existence of a dependence.
However, to be conservative, we always assume that a dependence exists, unless the intersecting region can be shown to be non-null with absolute certainty.

The other source of imprecision is the control flow within a region. If we union together all of the Data Access Descriptors for a given variable in a region to a single descriptor for the variable, it represents may information. That is, it is assumed that all control paths through the region will result in the access described by the single Data Access Descriptor for the variable. To keep must information, it would be necessary to keep Data Access Descriptors for each reference of a variable in a region. However, keeping only distinct Data Access Descriptors of the variable, as discussed earlier, may be a good compromise between overly conservative may information and storage intensive must information.

Some precision is also lost every time two Data Access Descriptors are unioned using algorithm UNION. This is because algorithm UNION computes an approximation simp(actual union), which is the smallest simple section containing the actual union. While this may be a good reason for not unioning the Data Access Descriptors for all references to an array variable in a region, the space cost can be prohibitive if they are not unioned into a single descriptor. Again, the solution suggested above may be a fairly good compromise between the two methods.

The representation of the simple section is also a source of imprecision. Although a large majority of frequently encountered access shapes in numerical scientific programs can be represented exactly by a simple section, there are some shapes that do not have exact representations in the simple section framework. Figure 2.18 shows some examples of non-simple sections in 2 and 3 dimensions. For such access shapes, algorithm SIMPLE.SECTION computes a simple section that contains the accessed section. Figure 2.19 shows a 3 dimensional non-simple section and its simple section approximation as computed by algorithm SIMPLE.SECTION. The shaded plane in Figure 2.19 is not a simple boundary of the section, because it is of the form $x_1 + x_2 + x_3 = c$. Simple boundaries can only be of the form $x_i = c$
Figure 2.18  Some non-simple sections in 2 and 3 dimensions.

Figure 2.19  (a) A non-simple section, and (b) its simple section approximation.

or \( x_i \pm x_j = c \) by definition (Definition 2.2). Algorithm SIMPLE_SECTION will approximate the shaded plane by the three planes marked A, B and C, all of which are simple boundaries. Plane A is of the form \( x_1 + x_3 = c_1 \), plane B is of the form \( x_1 + x_2 = c_2 \) and plane C is of the form \( x_2 + x_3 = c_3 \). It can be shown that algorithm SIMPLE_SECTION always computes the smallest simple section that contains the accessed section. Thus, although some imprecision is introduced in making this approximation, the loss of precision is kept as minimal as possible.
2.10 Dependence as a measure of inter-process communication

Thus far, we have conceptually viewed data dependence as arising due to memory overlap. An alternative view of data dependence is the value flow concept that views data dependence between two statements as a flow of some value computed by one to the other. Extending this concept to regions, a dependence between two regions implies that one produces data that is consumed by the other region. The amount of data that is communicated between the regions is given by the data overlap (i.e., intersection of their Data Windows). If the regions are tasks or independently schedulable processes, the size of the data overlap can be viewed as a measure of the amount of inter-process communication between the tasks. This notion of data dependence allows us to perform several useful optimizations that deal with process granularity. For example, in a shared-memory system, one of the concerns is whether the parallel processes have a large enough computation to communication ratio to give the desired speedup during parallel execution. In a distributed memory system, the goal is to minimize communication between processes that are bound to processors not directly connected by physical communication links. We can now address these concerns by computing the data overlap between the regions of code that constitute each process, using this information to tune the granularity of the processes for a particular architecture.

2.11 Comparison with related work

The work of Callahan and Kennedy on Regular Sections formed the basis of the simple section representation. In [Cal86], Callahan introduced the Regular Section Descriptor (RSD) as a method of representing the shapes of arrays accessed within a subroutine. The RSD is a very general representation; it keeps information about the bounds of loop induction variables and the actual subscript expressions in each
reference. From the loop bounds and the subscript expressions, the boundaries of
the section accessed by that reference can be derived. Unfortunately, this generality
makes several operations involving RSDs too expensive to be of practical use. In par-
ticular, two different RSDs could describe the same access shape, so that a procedure
called standardization, which attempts to make all RSDs have a standard traversal
order, must be applied before union or intersection operations can be performed.
Callahan and Kennedy later proposed a more restricted version of the RSD which
limited the describable shapes to single rows, columns and diagonals or the entire ar-
ray [CK87], and avoided the need for standardization. The Data Access Descriptor is
a compromise between the generality of the RSD and the simplicity of the restricted
RSD because the class of regions describable by a Data Access Descriptor are be-
tween those describable by the RSD and the restricted RSD. However, by isolating
the traversal order information from the description of the array section accessed,
the need for standardization is avoided. Every simple section of an array can be
described by a unique Data Access Descriptor. This makes union and intersection
operations involving Data Access Descriptors very fast and relatively easy to imple-
ment. Figure 2.3 shows some simple sections in 2 and 3 dimensions which can be
accurately represented by a Data Access Descriptor.

Li and Yew have taken a different approach to representing array accesses in
subroutines [LY88]. They maintain a list of atoms, that encode each array reference
in a subroutine. No attempt is made to summarize the effect of all the references into a
single descriptor. There are many similarities between the Regular Section Descriptor
and the information encoded in these lists. However, because the information is not
summarized, their technique gains some precision in dependence testing at the cost of
increased space requirements. The problem of summarizing these atoms into a single
representation for each variable is equivalent to the problem of constructing Regular
Section Descriptors.
The technique proposed by Triolet is also more general than the simple section. In his technique, all array accesses in a region are summarized by keeping a set of linear inequalities that describe the boundaries of the accessed portion [Tri86]. There is no restriction on these inequalities. Thus arbitrary convex shapes can be described. Unfortunately, this generality also makes the representation expensive. Such arbitrary access shapes are rarely encountered in practice, so such a general representation seems unnecessary.

2.12 Summary

This chapter has presented a new representation for array accesses in regions, called the Data Access Descriptor, and shown how it can be used to treat dependence testing between statements and larger regions in a uniform manner. The Data Access Descriptor has the advantage of separating the description of the accessed region from the specification of access order. This permits a dependence test that is independent of the ordering of the loops in the loop nest, yet retains the order information for determining when pipelining is possible. In addition, union and intersection, which are used for region merging and dependence testing respectively, are very efficient. The generalization of the concept of dependence from the statement-level to the region-level provides the flexibility to move from low level details to high level summary information. This can be exploited in an interactive system, where it is important to be able to provide high-bandwidth conceptual information to the programmer, while giving him the option of examining lower level details about data dependences between pairs of statements.
Chapter 3

Parallelism Detection and Enhancement

The traditional data dependence graph is useful in detecting vectorizable statements and parallel loops. However, when looking for parallelism between much larger regions of the program, statement level data dependence information can be cumbersome to use and expensive to manipulate. A representation is needed to deal with the interactions between regions of the program without having to examine all the statement level data dependences between regions. We have already seen how data access information can be summarized for regions using Data Windows. To avoid keeping statement level data dependences between regions, dependence can be represented as an intersection of Data Windows. In this way, a “region level” data dependence graph can be constructed, in much the same way as a statement level traditional data dependence graph would be constructed. This graph can then be used to find parallelism between entire regions. The construction of such a graph, its properties and its use in parallelism detection is the subject of this chapter.

3.1 Task graphs

A task can be loosely defined as an independently schedulable unit of computation. Let $T = T_1, T_2, \ldots, T_n$ be a set of tasks, and let $<$ be a precedence relation on $T$. The precedence relation $<$ imposes a partial order on $T$ which arises as a direct consequence of the control flow between the components of $T$. The notation $T_i < T_j$ denotes that task $T_i$ must finish execution before task $T_j$ can begin. $T_i$ is called a predecessor of $T_j$, and $T_j$ is called a successor of $T_i$. The set of tasks $T$ and the precedence relation $<$ can be described by a directed graph $G = (T, E_T)$, where
the set of edges \( E_T = \{(T_i, T_j) \mid T_i < T_j\} \). This graph is called a task graph. If we consider each basic block of a program to be a schedulable unit of computation, then the task graph is the traditional control flow graph of the program.

The simplest kind of task graph is the chain graph:

**Definition 3.1 (Chain graph):** A set of tasks \( T_1, T_2, \ldots, T_k \) in \( G \) is called a chain if

- The tasks are all distinct,
- \( T_i \) is a predecessor of \( T_{i+1} \) (\( i = 1, 2, \ldots, k - 1 \)),
- \( T_1 \) has no predecessor, and
- \( T_k \) has no successor.

A task graph \( G \) is called a chain graph if it consists of a single chain.

A chain of \( k \) tasks will be denoted as \( T_1T_2 \cdots T_k \). The length of a chain is the number of tasks in the chain. \( T_1 \) is the initial node, and \( T_k \) is the final node of the chain. Chains that have no nodes in common are called independent chains. A useful way to represent a task graph is by specifying its chain expression, which lists all of the chains in the graph.

**Definition 3.2 (Chain expression):** Let \( G \) be a task graph with \( m \) chains \( C_1, C_2, \ldots, C_m \). The chain expression for \( G \) is given by:

\[
C(G) = \sum_{C_j \in G} \left( \prod_{1 \leq i \leq m} T_i \right).
\]

Figure 3.1 shows a task graph and its corresponding chain expression. Each chain contributes one term to the sum. A task node may belong to several chains. The chain expression of a graph lists all the paths in the graph from each initial node to the final nodes (i.e., all of the chains of the graph).
3.2 Structured and non-structured task graphs

Task graphs can be broadly classified into two categories: structured task graphs and non-structured task graphs. The significance of structured and non-structured graphs for real programs will be discussed later. The following definition of structured graphs is adapted from the notion of “simple graphs”, first introduced by Robinson [Rob79].

**Definition 3.3 (structured graph):** A task graph $G$ is structured if its chain expression $C(G)$ can be factored so that each task node appears only once in the factored expression.

Figure 3.2 shows some structured and non-structured graphs, and their corresponding chain expressions. The chain expressions for the structured graphs after factoring contain only one occurrence of each task node. This is not true for the non-structured graphs.

3.2.1 FJ-graphs

Structured graphs contain an important class of graphs called FJ-graphs (or fork-join graphs). FJ-graphs are those task graphs that can be generated by the productions
$C(G) = T_1T_2 + T_1T_3T_4 + T_1T_3T_5$
$= T_1(T_2 + T_5(T_4 + T_3))$
$\Rightarrow$ structured.

$C(G) = T_1T_4 + T_1T_2T_4 + T_1T_2T_3T_4$
$= T_1T_4(1 + T_2(1 + T_3))$
$\Rightarrow$ structured.

$C(G) = T_1T_2T_4 + T_1T_3T_4 + T_1T_3T_5$
$= T_1T_3(T_4 + T_5 + T_2T_4)$
$\Rightarrow$ non-structured.

$C(G) = T_1T_2T_3T_4 + T_1T_2T_4 + T_1T_3T_4$
$= T_1T_4(T_2T_3 + T_2 + T_3)$
$\Rightarrow$ non-structured.

Figure 3.2 Some structured and non-structured graphs and their chain expressions.
Figure 3.3 Graph grammars for (a) FJ-graphs and (b) FJ-reducible graphs.
of the graph grammar shown in Figure 3.3(a). The set of productions of this graph grammar correspond to the Böhm and Jacopini definition for structured programs [BJ66, Ken81]. The chain expression $C(G)$ for a given FJ-graph $G$ can be derived in the following manner:

1. Extend the productions of the graph grammar with a label attribute, denoted by $l$ (the other attribute, denoted by $\Delta$, is the Data Window attribute, and will be discussed later). Initially, set $l(x) = x$ for each task node.

2. Parse the graph $G$ using this attribute grammar. When the right hand side of a production is recognized, its corresponding subgraph is reduced to a single task node (henceforth referred to as a compound node). Each time a reduction is performed, the label attribute of the compound node is computed as indicated in Figure 3.3(a).

3. If $G$ is an FJ-graph, it can be reduced to a single compound task node. The label attribute of this node is the chain expression for $G$ in its factored form.

4. If $G$ is not an FJ-graph, it cannot be reduced to a single node.

The sequence of reductions produced as a by-product of the above procedure is called the parse of the task graph. It can be shown that the attribute grammar of Figure 3.3(a) parses a task graph in time linear in the number of task nodes [Ken81].

An FJ-graph has a unique initial node (i.e., a node with no predecessors) and a unique final node (i.e., a node with no successors). Task nodes with out-degree greater than 1 are called fork nodes and those with in-degree greater than 1 are called join nodes. Each fork node has a corresponding join node in an FJ-graph. A fork node dominates all of its descendant nodes up to and including its corresponding join node. The join node postdominates all its ancestor nodes up to and including its corresponding fork node. Figure 3.4 shows examples of FJ-graphs.
3.2.2 Chain production

The production of the FJ-graph grammar that expands a single node to a chain graph (see Figure 3.3(a)) is an especially important one. It is used in the parallelism detection procedure that will be discussed shortly. We refer to this production as the chain production of the graph grammar. If we start from a single compound task node, and repeatedly apply chain productions to expand the graph until further expansion is no longer possible, we obtain a chain graph.

3.2.3 FJ-reducible graphs

For our purposes, the Böhm and Jacopini grammar of Figure 3.3(a) is insufficient, because it only describes a subset of the class of structured task graphs, namely the FJ-graphs. FJ-graphs are contained in the more general class of task graphs called FJ-reducible graphs. FJ-reducible graphs are those that can be generated by the productions of the grammar shown in Figure 3.3(b). The three extra productions of the FJ-reducible graph involve the recognition of subgraphs that can be transformed into
FJ-graphs by the addition of dummy task nodes. Figure 3.5 shows these subgraphs and their equivalent FJ-graph forms. The nodes labeled “D” represent dummy task nodes.

The following theorem proves that the task graphs that are generated by the FJ-reducible graph grammar are structured task graphs:

**Theorem 3.1** The class of task graphs generated by the productions of the FJ-reducible graph grammar are structured (i.e., their chain expressions can be factored so that each task node appears only once in the factored expression).

**Proof:** If a task graph $G$ can be generated by the productions of the FJ-reducible graph grammar, then starting from $G$, we can parse the graph by reversing the sequence of productions (i.e., apply reductions) until $G$ is reduced to a single compound graph.
node. The label attribute of this compound node is the factored chain expression \( C(G) \) of the graph \( G \). We must show that \( C(G) \) contains only one occurrence of each node of \( G \).

This is immediate from the construction of the label attribute of a compound node from the synthesized label attributes of its component nodes. For every reduction shown in Figure 3.3(a) and (b), the label of the compound node has only one occurrence of the label of each of its component nodes. Since each task node \( x \) of \( G \) initially has its label \( l(x) \) set to \( x \), this invariant is preserved in every reduction of the parse. The statement of the theorem follows. \( \Box \)

The combined graph grammar of Figure 3.3(a) and (b) is called a structured graph grammar.

3.3 Significance of structured and non-structured task graphs

We have shown that the class of structured task graphs contains a hierarchy of three sub-classes of task graphs: chain graph \( \subseteq \) FJ-graph \( \subseteq \) FJ-reducible graph (see Figure 3.6). The programs that produce these kinds of task graphs are discussed below.

Structured task graphs are produced by structured programs, written using a disciplined programming style. The productions of the FJ-graph grammar and the FJ-reducible graph grammar define the program constructs that are considered to be "structured". They include sequencing, if-then and if-then-else conditionals, single entry - single exit loops which may be sequential DO loops or PARALLEL LOOPS, and case statements with their parallel equivalent, the PARALLEL SECTIONS statements.

The productions of the structured graph grammar reveal the intuition behind the definition of structured graphs in terms of factorability of chain expressions (definition 3.3). If a node can be factored out from the chain expression, then that node must be
common to all of the chains. If after factoring, an independent set of chains remain, then the graph must have the fork-join property. That is, such a graph is either an FJ-graph or is reducible to an FJ-graph.

Non-structured task graphs are produced by badly structured programs, which contain complex control flow. Undisciplined use of branch statements like the GOTO usually produce unstructured programs. In the discussions that follow, it is assumed that the input program is well structured, and its task graph is a structured task graph.

3.4 A hierarchical parallelism detection procedure

This section presents a parallelism detection procedure that traverses the task graph in a hierarchical fashion using the absence of data dependences between the task nodes to uncover parallelism in the program. The parallelism is identified by topologically partitioning the nodes of the graph, so nodes that have no data dependences between them are grouped together into a single partition.

**Definition 3.4 (topological partition):** A topological partition of the nodes of a directed acyclic graph is a grouping of the nodes into sets $S_1, S_2, \ldots, S_n$, such that there is no edge between the nodes grouped in the same set, and for any two nodes $x \in S_r$ and $y \in S_t$, $r < t \Leftrightarrow x$ precedes $y$ in any topological ordering of the nodes.

---

![structured task graphs](image)

**Figure 3.6** Classes of structured task graphs.
A topological partition of the nodes of a graph can be determined by a straightforward
modification of the standard topological sort algorithm [AUH74]. The topological
partition of a directed acyclic graph may not be unique.

Given a program $\mathcal{P}$, the traditional control flow graph is constructed, where each
node represents a basic block, except that a subroutine call statement is also treated
as a separate basic block. The control flow graph is the initial task graph $G$ of the
program, and each basic block represents a "task" node. Data Windows are computed
for each basic block, using the techniques discussed in the previous chapter.

The graph $G$ is parsed using the structured graph grammar, and the Data Window
attribute is used to collect the Data Window information from the basic blocks,
merging them together with each reduction of the parse. The parse will terminate
when the entire graph has been collapsed into a single compound node. This node
is the task graph at level 0, denoted by $G^0$. The parse is then reversed, and $G^0$ is
expanded in steps. The idea is to look for parallelism after each step of the expansion,
so that coarse grain parallelism can be identified before going further on to uncover
relatively finer grain parallelism. With each expansion step, the level of the task graph
increases. Parallelism uncovered in the task graph $G^k$ is called level $k$ parallelism.

The following steps outline such a technique for methodically uncovering parallel-
ism in the program.

1. Parse the task graph $G$ using the structured graph grammar. During the parse,
   Data Window information is computed at each reduction step using the Data
   Window attribute of the structured graph grammar (see Figure 3.3). As the
structured graph grammar of Figure 3.3(a) indicates, the Data Window for a
loop is the Data Window of the loop body translated outwards, and the Data
Window of a call site is the Data Window of the called subroutine translated
outwards to the calling routine. Outwards translation of a Data Window is
denoted by the "$\dagger$" symbol. The details of the translation procedure were
discussed in chapter 2. If the program is structured, the parse will yield a
sequence of reductions that reduce $G$ to a single compound node, which is the level 0 task graph. The sequence of reductions can be reversed to get a sequence of productions, that will be used to generate the task graph $G$ starting from the final compound node. Set the level counter $k$ to 1.

2. Expand the single compound node created in step 1 by applying successive chain productions, until no more nodes can be expanded by applying the chain production. Let the initial node of the resulting chain graph be $x$. This chain graph is the level $k$ task graph rooted at $x$, denoted by $G^k(x)$.

3. Construct the Task Dependence Graph of $G^k(x)$, denoted by $TDG^k(x)$. The nodes of $TDG^k(x)$ are the same as $G^k(x)$. For every pair of nodes $u, v \in TDG^k(x), u < v$ in $G^k(x)$, compute the data overlap $\Delta(uv) = \Delta(u) \cap \Delta(v)$. If the data overlap is nonempty, add the directed edge $(u, v)$ to $TDG^k(x)$. $TDG^k(x)$ will be a directed acyclic graph by construction, where the edges represent data dependences.

4. Topologically partition the nodes of $TDG^k(x)$. All independent sets of nodes (i.e., nodes that have no data dependences between them) will be grouped together in the same partition. A partition that contains more than one node indicates an opportunity for parallelism; all nodes in the same partition can be executed concurrently without violating any data dependences in the program.

5. Visit each topological partition in order. If a partition contains more than one node, mark the nodes as separate SECTION clauses of a PARALLEL SECTIONS statement.

6. If a partition contains a single compound node, expand it by using the appropriate production of the structured graph grammar. The production cannot be
Algorithm \textsc{Find\_Parallelism}(G)

\textit{Input}: A task graph $G$.
\textit{Output}: Parallelism identified in the task graph $G$.

begin
1. compute the Data Window $\Delta(v)$ for each task node $v \in G$;
2. parse $G$ using the structured graph grammar;
3. if the parse does not reduce $G$ to a single node then quit;
   else
4. let $x$ be the single compound node produced by the parse;
5. \textsc{Par\_Search}(x, 1);
6. endelse
endif
end

\textbf{Figure 3.7} Algorithm to uncover parallelism in the program.

a chain production, because after step 2, $G^k(x)$ cannot be expanded any further by using the chain production.

7. If the production in step 6 yields a programmer specified parallel region, such as a PARALLEL LOOP or PARALLEL SECTIONS, expand this node further only if nested parallelism is desired. Otherwise, do not proceed further into this node.

8. If the production in step 6 yields a DO loop, check for parallelism between the iterations of the loop. This can be done by taking the Data Window of a single iteration of the loop, and checking to see if changing the value of the loop induction variable creates a non-intersecting Data Window. The details of this are given later.
Procedure PAR_SEARCH (x, k)

/* search for level k parallelism in the compound node x. */
begin
1   applying only chain productions, expand x as much as possible;
2   let v be the initial node of the resulting chain graph G^k(v);
3   construct TDG^k(v);
4   topologically partition TDG^k(v);
5   for each partition do
6       if there is only 1 task node z in the partition then
7           if z is a compound node then
8               expand z by applying the appropriate production;
9               if z expands to a loop with body w then
10                  /* Check for loop parallelism (refer to section 3.4.1). */
11                     if the loop is parallel
12                         then mark z as a "parallel loop";
13                         else PAR_SEARCH(w, k+1);
14                     endif
15                 else if z expands to a subroutine call then
16                     let S be the local task graph of the subroutine body;
17                     invoke algorithm FIND_PARALLELISM on S to uncover
18                         parallelism within the subroutine;
19                     endelse
20                 else if z expands to an FJ-reducible graph with fork node f then
21                    /* This is an if-then-else or case statement or PARALLEL SECTIONS */
22                       for all successors w_i of f do
23                          /* Look for parallelism at deeper levels */
24                          PAR_SEARCH(w_i, k+1);
25                      endfor
26                 endif
27             endif
28           endif
29       else if the partition has more than one node then
30           group all nodes in the partition into a single node z';
31           mark z' as "parallel sections";
32       endelse
33     endfor
34 end

Figure 3.8 Procedure to recursively search for parallelism at deeper levels.
9. If the production in step 6 yields a subroutine call, then let $S$ be the local task graph of the subroutine. Invoke algorithm FIND_PARALLELISM recursively on the task graph $S$. This will uncover parallelism within the subroutine.

10. If the production in step 6 yields a graph that consists of a fork node followed by a set of nodes dominated by the fork node, and possibly a join node that postdominates the same set of nodes, then ignore the fork and join nodes and visit each of the remaining nodes in order. For each node visited, increment the level counter $k$ by 1, and go to step 2, where this node is recursively expanded to look for parallelism at level $k + 1$.

11. The recursion will terminate when a node that cannot be expanded any further is visited. This will be on the level of a basic block. The process now returns to the level of the topological partition visited at step 5.

Algorithm FIND_PARALLELISM in Figure 3.7 performs the hierarchical parallelism detection described above. Procedure PAR SEARCH in Figure 3.8 expands each node, and recursively looks for parallelism at deeper levels (i.e., at progressively greater values of the level counter). When checking for dependence (i.e., non-empty data overlap), the dependences of interest are write-read dependences $\Delta^{wr}(T_i; T_j)$, write-write dependences $\Delta^{ww}(T_i; T_j)$ and read-write dependences $\Delta^{rw}(T_i; T_j)$ between the two task nodes. read-read dependences $\Delta^{rr}(T_i; T_j)$ are not taken into consideration because they do not inhibit safe parallelization of the two regions.

Example 3.1:

Figure 3.10 shows a task graph and the output of algorithm FIND_PARALLELISM. When algorithm FIND_PARALLELISM is called on the task graph $G$, line 2 of the algorithm parses $G$ using the structured graph grammar. The sequence of reductions produced by the parse are shown in Figure 3.11. The Data Window attribute of the structured graph grammar is used to propagate Data Window information upwards from the deeper levels with each reduction of the parse.
Reversing the sequence of reductions in Figure 3.11, we obtain a sequence of productions, which is used by procedure PAR_SEARCH to traverse $G$ in a hierarchical fashion, searching for parallelism at each level. The steps of the procedure PAR_SEARCH are illustrated in Figure 3.9. Dependence edges in the Task Dependence Graphs are hypothetical and have been inserted for illustrative purposes only.

\[<\]

### 3.4.1 Detection of loop parallelism

When a node expands to a loop in line 9 of procedure PAR_SEARCH, a check is done to see if the iterations of the loop have any dependences between them. Such dependences are called loop-carried dependences, and their absence indicates that the iterations can be safely executed in parallel. Let $w$ be the node that represents the body of the loop, and $I$ be the loop induction variable. $I$ is a symbolic variable within the body of the loop, represented by the node $w$.

Let $\Delta(w)$ be the Data Window for the region represented by $w$. For illustration, let us assume that $\Delta(w)$ consists of a single Data Access Descriptor, $\delta(A)$, for the reference to the array $A$ within the loop. Let $\delta(A)_{|I-I+p}$ be the Data Access Descriptor obtained by replacing all occurrences of $I$ in $\delta(A)$ by $I + p$, where $p$ is some constant value within the range of $I$. Compute $\delta(A) \cap \delta(A)_{|I-I+p}$ using the method described in Chapter 2. The loop is parallel if the intersection is empty. Otherwise, assume that there is a loop-carried dependence and the loop cannot be made parallel.

### 3.4.2 Finding nested parallelism

Algorithm FIND_PARALLELISM shown in Figure 3.7 does not search for parallelism within a node that has been identified as part of a parallel region. Specifically, in line 19 of procedure PAR_SEARCH, when a set of nodes in the same topological partition are grouped together into “parallel sections”, the nodes are not expanded any further.
Figure 3.9 Illustration of procedure PAR.SEARCH.
Similarly, in line 11, if a loop is found to be parallel, PAR\_SEARCH is not invoked on the loop body. Thus, the algorithm shown in Figure 3.7 only finds parallelism at the highest possible level in the hierarchical task graph.

It is easy to extend algorithm FIND\_PARALLELISM to search for nested parallelism, by making two changes:

- Immediately after line 21 in procedure PAR\_SEARCH, insert the following:

```plaintext
for each component w ∈ z' do
    PAR\_SEARCH(w, k + 1);
endfor
```

This invokes PAR\_SEARCH on the nodes that are part of a parallel section at level k, so that nested parallelism can be uncovered within them.

- Immediately after line 11 in procedure PAR\_SEARCH, insert the following:

```plaintext
PAR\_SEARCH(w, k + 1);
```

This will uncover nested parallelism within a parallel loop.

### 3.4.3 Dealing with programmer specified parallelism

Algorithm FIND\_PARALLELISM can also handle programmer specified parallelism very naturally. Recall from Chapter 1, that programmer specified parallelism is limited to PARALLEL LOOPS and PARALLEL SECTIONS. These constructs yield FJ-reducible task graphs, which can be parsed by the structured graph grammar. Within procedure PAR\_SEARCH, when a compound node expands to an already parallelized compound node, we can either go to the next topological partition, or expand the node one more step and look for nested parallelism.

When a programmer specified parallel node is encountered, we can also check to ensure that the parallel segment is deterministic. Nondeterminism arises when there is incorrect synchronization or lack of synchronization between parallel regions.
that have a data dependence between them. The sources of such nondeterminacy can be identified and reported to the programmer as a potential anomaly. The synchronization within the programmer specified parallel segment, such as between the SECTIONs of a PARALLEL SECTIONS statement, can be analyzed in this manner. Analysis of synchronization is the subject of the next chapter.

3.4.4 Time complexity

It can be shown that parsing the graph \( G = (V, E) \) using the structured graph grammar and generating the sequence of reductions takes \( O(V) \) time [Ken81]. The topological partition can be done in \( O(V + E) \) time. Procedure PAR_SEARCH visits each node of \( G \) only once. The most expensive step in the algorithm is line 3, when the Task Dependence Graph \( TDG^k(v) \) is constructed. Since every pair of nodes in \( G^k(v) \) is compared to obtain the dependence edges, this procedure has a worst case time complexity of \( O(V^2) \).

Observe that \( G^k(v) \) is always a chain graph by construction. We can take advantage of this fact, and combine the computation of dependence edges with the topological partitioning process, accomplishing both of them in time linear in the number of nodes of \( G^k(v) \). In particular, consider a chain graph \( F = x_1 < x_2 < \cdots < x_n \). The Task Dependence Graph of \( F \) will have the same set of nodes \( x_1, x_2, \ldots, x_n \) and data dependence edges connecting pairs of nodes. A topological partition of the nodes in the Task Dependence Graph can be computed as follows:

- Put the initial node \( x_1 \) in a set \( S_1 \), and let \( \Delta(S_1) = \Delta(x_1) \).
- If \( \Delta(S_1) \cap \Delta(x_2) \neq \phi \), add \( x_2 \) to the set \( S_1 \), and let \( \Delta(S_1) = \Delta(S_1) \cup \Delta(x_2) \).
- Otherwise, if the intersection is non-empty, create a new set \( S_2 \) and add \( x_2 \) to \( S_2 \).
- Continuing in this manner, when node \( x_i \) is visited, let \( S_h, h < i \) be the set containing \( x_{i-1} \). Compute \( \Delta(S_h) \cap \Delta(x_i) \).
• If the intersection is $\phi$, add $x_i$ to set $S_h$. This indicates that $x_i$ can be executed concurrently with all nodes in $S_h$, since there is no data dependence from any node in $S_h$ to $x_i$.

• Else if the intersection is non-empty (i.e., there is a data dependence from some node in $S_h$ to node $x_i$), create a new set $S_{h+1}$, and add $x_i$ to $S_{h+1}$.

The sequence of sets $S_1, S_2, \ldots S_t$ produced by the above method represents a topological partition of the nodes of the Task Dependence Graph of $F$. Since this method visits each node of $F$ only once, its time complexity is linear in the number of nodes. Using this technique, lines 3 and 4 of procedure PAR.SEARCH can be done in time $O(V)$.

Thus, the entire algorithm FIND_PARALLELISM has a theoretical worst case time complexity of $O(V)$. The algorithm finds parallelism of different granularities depending on the level at which the parallelism is uncovered. Parallelism found at upper levels represents relatively coarse grain parallelism, while parallelism found at deeper levels, such as between basic blocks, represents fine grain parallelism. Procedure PAR.SEARCH can be modified to continue searching for parallelism within a basic block (i.e., between the statements of a basic block) to uncover much much finer granularities of parallelism if desired.

3.4.5 Accuracy of dependence testing

The theoretical time complexity of $O(V)$ is achieved because the construction of the Task Dependence Graph can be combined with the topological partitioning. Although this lowers the theoretical time complexity of algorithm FIND_PARALLELISM, it could also considerably lower the accuracy of dependence testing. This is because union of Data Windows introduces some imprecision, which tends to accumulate as more unions are performed (see chapter 2). The technique described in the previous subsection for the construction of the Task Dependence Graph may not be very suitable in practice. It may be better to use the $O(V^2)$ greedy method, comparing
Figure 3.10  (a) The input task graph $G$ and (b) the output of algorithm FIND_PARALLELISM.

every pair of nodes, to compute the data dependence edges of the Task Dependence Graph in order to avoid too much imprecision. We can then topologically partition the nodes to locate parallelism on a second pass.

The precision of the Data Window information is highest at the basic block level, where it is initially computed. As we parse the graph, reducing nodes into compound nodes, the unioning of Data Windows lowers the precision. The parallelism detection procedure PAR_SEARCH starts at the uppermost level, from the completely reduced task graph, and gradually expands the graph, working towards the deeper levels. Thus, greater values of the parameter $k$ in procedure PAR_SEARCH represent deeper levels at which parallelism is uncovered and finer granularity of parallelism being identified. Also, at larger values of $k$, the Data Window information gets more precise, the accuracy of dependence testing improves, and consequently the possibility of detecting finer grain parallelism increases.
3.5 Task Pipelining

The parallelism detection procedure has one major drawback. It cannot identify parallelism that may exist between a node within an FJ-reducible subgraph, such as an if-then-else, case or PARALLEL SECTIONS statement, and a node outside such a subgraph. This deficiency arises because algorithm FIND_PARALLELISM only identifies parallelism between the nodes of a chain graph, by using the topological partitioning procedure. Some of the nodes in the chain graph may be compound nodes.

In this section, a pipelining technique is presented, that allows us to correct the deficiency in the parallelism detection process by identifying parallelism between nodes that are within compound statements and nodes that are outside. Pipelining involves synchronizing the data overlap between the regions, so that they can execute concurrently in lock-step, without violating any data dependences between them.
Consider two regions $R_1$ and $R_2$ that have a data dependence $\Delta(R_1R_2)$ between them. This implies that $R_2$ must wait until $R_1$ has completed all accesses to the data described by $\Delta(R_1R_2)$, before starting execution. If several free processors available in the system, $R_2$ need not wait until $R_1$ completes. $R_2$ can be scheduled for execution as soon as $R_1$ has completed accesses to $\Delta(R_1R_2)$. $R_1$ and $R_2$ can thus be pipelined in the following manner:

1. Starting from the end of $R_1$, locate the earliest point at which the data described by $\Delta(R_1R_2)$ is completely accessed (i.e., the point in $R_1$ beyond which no part of $\Delta(R_1R_2)$ is referenced). If this point is not the end of $R_1$, insert a "POST EVENT evar" statement at that point, where "evar" is an EVENT variable (see chapter 1 for the definition and semantics of event synchronization).

2. At the beginning of $R_2$, insert a "WAIT EVENT evar" statement.

3. Group $R_1$ and $R_2$ into a PARALLEL SECTIONS statement. The event synchronization will ensure that the data dependence between $R_1$ and $R_2$ is always preserved.

For the above procedure to work, a method is devised to "locate the earliest point" within $R_1$ at which the data overlap $\Delta(R_1R_2)$ is completely accessed. This can be done in the following manner:

1. Let the region $R_1$ be represented by the compound node $x$, and the region $R_2$ by the compound node $y$ in the level $k$ task graph $G^k(v)$ rooted at node $v$. $G^k(v)$ is constructed during the parallelism detection process, in algorithm FIND_PARALLELISM. The data overlap (or data dependence) between $x$ to $y$ is given by $\Delta(xy) = \Delta(x) \cap \Delta(y)$.

2. Expand the compound node $x$. Note that $x$ cannot be expanded any further using chain productions, due to the way $G^k(v)$ is constructed (see Figure 3.9 and algorithm FIND_PARALLELISM).
3. If \( x \) expands into an FJ-reducible graph with fork node \( f \), then for each of the successors \( w_1, w_2, \ldots, w_m \) of \( f \), we locate the "earliest point" at which \( \Delta(xy) \) is completely accessed.

4. If \( x \) expands to a subroutine call, we traverse the body of the subroutine to find the "earliest point" at which \( \Delta(xy) \) is completely accessed.

5. Similarly, if \( x \) expands into a loop, we traverse the body of the loop, looking for the "earliest point".

Algorithm PIPELINE in Figure 3.12 uses such a strategy to pipeline \( x \) and \( y \). Given a node \( z \) that is produced as a result of expanding \( x \) above, we can find the "earliest point" at which \( \Delta(xy) \) is accessed in the following manner:

1. Expand \( z \) using only chain productions until no further expansion is possible. Suppose that the resulting chain graph is \( z_1 < z_2 < \cdots < z_n \).

2. Starting from the node \( w_n \), and working towards \( w_1 \), locate the first node \( w_k \) for which \( \Delta(w_k) \cap \Delta(xy) \neq \emptyset \).

3. If \( w_k \) is not a compound node, insert a POST EVENT at the end of \( w_k \). Then, node \( y \) can start execution as soon as node \( w_k \) has completed.

4. If \( w_k \) is a compound node, expand \( w_k \) and repeat the above procedure, attempting to push the POST EVENT as close to the beginning of \( w_k \) as possible.

Procedure SYNCHRONIZE, called by algorithm PIPELINE, uses this technique to locate the earliest point in \( x \) at which to post the event variable (see Figure 3.13).

There are two complications that arise in algorithm PIPELINE, which have not been discussed. These complications occur when node \( x \) expands to a loop, and when \( x \) expands to a PARALLEL SECTIONS.
Algorithm PIPELINE

Input: a task node \( y \), its (unique) predecessor \( x \) and data overlap \( \Delta(xy) \).
Output: EVENT synchronization inserted to pipeline \( x \) and \( y \).

begin
  1 if \( x \) expands to an FJ-reducible graph with fork node \( f \) then
    /* \( x \) is a case statement (PARALLEL SECTIONS can be handled similarly).
    See sections 3.5.1 and 3.5.2 */
    2 create a new EVENT variable \( evar \);
    3 for all successors \( w_i \) of \( f \) do
      4 SYNCHRONIZE (“\( f \)”, \( w_i \), \( \Delta(xy) \), \( evar \)); /* insert POST EVENT */
      5 at the beginning of \( y \) insert the statement “\( \text{WAIT EVENT } evar \)”;
    endfor
  endif
  6 else if \( x \) expands to a loop with induction variable \( I \) and loop body \( w \) then
    7 let \( L_I, U_I, S_I \) be the lower bound, upper bound and stride of \( I \) respectively;
    8 if \( y \) is also a loop, with induction variable \( J \) then
      9 if the traversal order of any of the \( \delta \in \Delta(xy) \) is “*”
      10 then report that \( x \) and \( y \) cannot be pipelined profitably, and quit;
      11 create a new EVENT array variable \( evar(I) : (U_I - L_I)/S_I \);
      12 SYNCHRONIZE (“loop”, \( w \), \( \Delta(xy) \), \( evar(I) \)); /* insert POST EVENT */
      13 create a new integer variable \( $I$ \);
      14 at the beginning of \( y \) insert the statement “\( $I = L_I \)”;
      15 at the top of the body of the loop in \( y \) insert the statement
      16 “IF \( J \geq $I \) AND \( $I \leq U_I \) THEN
        \( \text{WAIT EVENT } evar($I) \)
        \( $I = $I + S_I \)
    ENDIF”
    endif
    17 else if \( y \) is not also a loop
    18 then report that \( x \) and \( y \) cannot be pipelined profitably, and quit;
  endelse
  19 else if \( x \) expands to a call to a subroutine \( S \) then
    20 parse \( S \), and let \( w \) be the final reduced node;
    21 create a new EVENT variable \( evar \);
    22 SYNCHRONIZE (“callsite”, \( w \), \( \Delta(xy) \), \( evar \)); /* insert POST EVENT */
  endelse
  23 make \( x \) and \( y \) separate SECTIONS in a PARALLEL SECTIONS statement;
end

Figure 3.12 Algorithm to pipeline two tasks.
Procedure SYNCHRONIZE (type, z, Δ', evar)

begin
/* insert a POST at the point in z, before which Δ' is completely accessed */
1 expand z using only chain productions to get a chain graph z₁ < z₂ < ⋯ < zₙ;
2 Found ← false; i ← n;
3 while i ≥ 1 and not Found do
4    if type = "callsite" then overlap = ↑ Δ(zᵢ) ⋂ Δ'
5        else overlap = Δ(zᵢ) ⋂ Δ'
6    if overlap ≠ φ then Found ← true;
7        else i ← i - 1;
8          endwhile
9    if Found then
10       if type ≠ "callsite" then
11          if zᵢ is a compound node that expands to an FJ-reducible graph with fork node f
12              then for all successors wᵢ of f do
13                 SYNCHRONIZE (type, wᵢ, Δ', evar);
14                 end
15              endif
16          endif
17    endif
18 if not Found and i = 1 then
19    at the end of node z₁ insert "POST EVENT evar";
20 endif
21 end

Figure 3.13 Procedure to insert synchronization for pipelining.

3.5.1 Pipelining two loops

If node x expands into a loop, then the portion of the data represented by Δ(x) is accessed iteratively, where each iteration of the loop accesses a chunk of this data. If node y is also a loop, then y also accesses its portion of the data, given by Δ(y), in an iterative manner. Pipelining between x and y can be achieved if we can ensure that each time a particular iteration of the y loop starts, the chunk of Δ(xy) accessed by this iteration has already been accessed by the iterations of the x loop. That is, the x and y loops are staggered such that as soon as an iteration of x completes access to
its chunk of $\Delta(xy)$, the iteration of $y$ that accesses this same chunk starts executing. To stagger the loops in this way, an event array variable is used. An element of the event array is posted by each iteration of the $x$ loop. An iteration of the $y$ loop must wait until its counterpart in the $x$ loop completes access to the chunk of $\Delta(xy)$ that is common to both iterations.

Such a staggering is profitable only when both loops $x$ and $y$ access all the common array variables in the same order. For instance, if loop $x$ traversed a 1 dimensional array $A$ from lower to higher indices, while loop $y$ traversed the same array $A$ from higher to lower indices, their iterations cannot be staggered. Similarly, if loop $x$ traversed a 2 dimensional array in a columnwise manner and loop $y$ traversed the same array in a rowwise manner, staggering of their iterations is impossible. The information about access order is kept in the traversal order field of the Data Access Descriptor for each array variable (see chapter 2). Pipelining is profitable only if the traversal orders of all array variables accessed by both $x$ and $y$ loops match. Algorithm PIPELINE checks for a match and quits if a mismatch is detected in the traversal order fields of any Data Access Descriptor in $\Delta(xy)$.

An important feature of the algorithm is that the event array variable that is posted within the loop in node $x$ is indexed by the loop induction variable of the outermost loop in $x$. This is true regardless of how deeply nested the POST EVENT is actually inserted within $x$. This is necessary because the data overlap $\Delta(xy)$ is computed by intersecting the Data Window for the outermost loop in $x$ with $\Delta(y)$. The algorithm must take this fact into account, otherwise an incorrect pipelining may result.

**Example 3.2:**
Figure 3.14 illustrates how two tasks are pipelined by algorithm PIPELINE. The Data Windows for the tasks $T_1$ and $T_2$ consist of only one Data Access Descriptor, for the array variable $A$. The Data Windows for task $T_1$ and the body of the I loop of $T_1$ are:
\( T_1: \)

\[
\begin{align*}
\text{PARALLEL SECTIONS} \\
\text{SECTION} \\
\begin{align*}
\text{DO } & I = 1,100,2 \\
\text{DO } & J = 1,100 \\
A(I,J,1) & = \ldots \\
\text{ENDDO} \\
\text{ENDDO} \\
\text{POST EVENT evar(I)} \\
\text{ENDDO}
\end{align*}
\end{align*}
\]

\( T_2: \)

\[
\begin{align*}
\text{SECTION} \\
\begin{align*}
\text{DO } & I = 1,100 \\
\text{DO } & J = 1,100 \\
\text{DO } & K = 1,100 \\
\text{IF } (I \geq I \text{ AND } I \leq 100) & \text{ THEN} \\
\text{WAIT EVENT evar(I)} \\
\text{ENDIF} \\
\text{DO } & J = 1,100 \\
\text{DO } & K = 1,100 \\
\ldots & = A(I,J,K) \\
\text{ENDDO} \\
\text{ENDDO} \\
\text{ENDDO}
\end{align*}
\end{align*}
\]

\[ \text{END PARALLEL SECTIONS} \]

(a) Before pipelining  \hspace{1cm} (b) After pipelining

\textbf{Figure 3.14} Illustration of task pipelining.

I loop:  \( \Delta(T_1) = \delta_{T_1}(A) = \left\{ x_1, x_2, 1 \right\} \)

\[
\begin{align*}
1 \leq x_1 & \leq 100, 1 \leq x_2 \leq 100 \\
2 \leq x_1 + x_2 & \leq 200 \\
-99 \leq x_1 - x_2 & \leq 99 \\
& x_2 \succ x_1
\end{align*}
\]

J loop:  \( \Delta(T_1^*) = \delta_{T_1^*}(A) = \left\{ I, x_2, 1 \right\} \)

\[
\begin{align*}
1 \leq x_1 & \leq 100 \\
2 \leq x_1 + x_2 + z_3 & \leq 200 \\
-99 \leq x_1 - x_2 - z_3 & \leq 99 \\
& x_3 \succ x_2 \succ x_1
\end{align*}
\]

The Data Window for task \( T_2 \) is:

\[
\Delta(T_2) = \delta_{T_2}(A) = \left\{ x_1, x_2, x_3 \right\}
\]

\[
\begin{align*}
1 \leq x_1, x_2, x_3 & \leq 100 \\
2 \leq x_1 + x_2 + x_3 & \leq 200 \\
-99 \leq x_1 - x_2 & \leq 99 \\
& x_3 \succ x_2 \succ x_1
\end{align*}
\]

and the data overlap (or data dependence) \( \Delta(T_1 T_2) = \Delta(T_1) \cap \Delta(T_2) \) is given by:

\[
\Delta(T_1) \cap \Delta(T_2) = \delta_{T_1}(A) \cap \delta_{T_2}(A) = \left\{ x_1, x_2, 1 \right\}
\]

\[
\begin{align*}
1 \leq x_1 & \leq 100, 1 \leq x_2 \leq 100 \\
2 \leq x_1 + x_2 & \leq 200 \\
-99 \leq x_1 - x_2 & \leq 99 \\
& x_2 \succ x_1
\end{align*}
\]
3.5.2 Pipelining with a PARALLEL SECTIONS

The second complication in algorithm PIPELINE occurs when a node $x$ expands to a PARALLEL SECTIONS statement. The node can be treated as if it were a case statement, with one modification. The discussion about structured programs indicated that both case and PARALLEL SECTIONS constructs are represented by an FJ-reducible graph with a fork node $f$, a join node, and a set of nodes (successors of $f$) that are the clauses of the case statement or the SECTIONS of the PARALLEL SECTIONS statement. When $x$ expands to a case statement, only one of the clauses will actually execute at run-time, so the same event variable can be posted by all the clauses. The node $y$ must wait for this single event variable to be posted before it starts execution. Algorithm PIPELINE handles this case. However, if $x$ expands to a PARALLEL SECTIONS statement, all of the sections may execute simultaneously at run-time, so they cannot all post the same event variable. We therefore need a separate event variable for each section, and node $y$ can start execution only when all of the event variables have been posted. Although algorithm PIPELINE as shown in Figure 3.12 does not handle this case, it can be easily extended to do so.

Example 3.3:

The example in Figure 3.15 illustrates the expansion of a node to a PARALLEL SECTIONS. Figure 3.15(a) represents the original program with its level 1 Task Dependence Graph shown below. Figure 3.15(b) shows the program with parallelism identified by algorithm FIND_PARALLELISM. Tasks $T_1$ and $T_2$ have been grouped together into a PARALLEL SECTIONS statement. Algorithm PIPELINE is called to pipeline the PARALLEL SECTIONS node with task $T_3$. The program with the specified pipelining is shown in Figure 3.15(c).
PARALLEL SECTIONS

SECTION
DO I = 1,100
DO J = 1,100
A(I,J,1) = ...
ENDDO
POST EVENT evar1(I)
ENDDO

SECTION
DO I = 1,100
DO J = 1,100
A(I,J,100) = ...
ENDDO
POST EVENT evar2(I)
ENDDO

SECTION
S$1 = 1
S$J = 1
DO I = 1,100
IF (I$S$1 AND $I$S$1$100) THEN
WAIT EVENT evar1($S$I)
$S$I = $S$I + 1
ENDIF
IF (I$S$J AND $S$J$S$1$100) THEN
WAIT EVENT evar2($S$J)
$S$J = $S$J + 1
ENDIF
DO J = 1,100
DO K = 1,100
... = A(I,J,K)
... = A(I,J,K)
ENDDO
ENDDO
ENDDO

PARALLEL SECTIONS

T1:
DO I = 1,100
DO J = 1,100
A(I,J,1) = ...
ENDDO

T2:
DO I = 1,100
DO J = 1,100
A(I,J,100) = ...
ENDDO

T3:
DO I = 1,100
DO J = 1,100
DO K = 1,100
... = A(I,J,K)
... = A(I,J,K)
ENDDO

(a) original program  (b) task parallelism  (c) with pipelining

Figure 3.15 Parallelism detection and enhancement using the Task Dependence Graph.
Program segments such as the one illustrated here are often encountered in numerical scientific programs. The first two loops initialize two of the boundary planes of a 3 dimensional mesh, modeled by the 3 dimensional array A. The third loop then sweeps the entire mesh.

3.6 Automatic pipelining

Algorithm PIPELINE can be incorporated into algorithm FIND_PARALLELISM so that pipelining is automatically performed when possible. Procedure PAR_SEARCH can be modified so that when it visits a node that is the only one in the current topological partition, algorithm PIPELINE is invoked to pipeline this node with its predecessor if possible.

3.7 Programmer guided pipelining

In general, it is difficult for a programming tool to statically decide when pipelining two tasks will improve run-time performance. This problem is dealt with in the same way that we dealt with the problem of choosing a parallelization strategy for good performance. We assume that the programmer understands the nature of the target machine and is capable of making these decisions. We provide an option by which algorithm PIPELINE can be selectively applied on pairs of task nodes indicated by the programmer. If pipelining is unsuccessful, the tool can report the mismatched traversal orders to the programmer. The programmer can either decide not to pipeline, or ask the tool to check if some traversal changing transformations such as loop interchange or loop reversal can be applied to one of the tasks to make the traversal orders match. A study of the kinds of transformations that can enhance pipelining is outside the scope of this research, and is left as an open problem.
3.8 Related work

The work of the PTRAN group is closely related to the techniques discussed in this chapter [ABC+88, ABC+87]. Similar methods were also developed by Hsieh [Hsi88]. Their technique is restricted to “structured programs”, whose graphs can be parsed by the Böhm and Jacopini grammar [BJ66]. They also uncover parallelism of different granularities, but there is a fundamental difference between their approach and the one presented here. They construct a process tree of the program, using control dependence information. A node $y$ is control dependent on another node $x$ if the execution of node $y$ depends on whether the flow of control leaves $x$ along a path that must go through $y$. A process tree represents all the parallelism that is available in the program if only the order imposed by control flow is preserved. Once such parallelism is uncovered, all data dependences (i.e., traditional data dependences between pairs of statements) are satisfied by inserting the necessary synchronization between the parallel sections.

Algorithm FIND_PARALLELISM does not use any control dependence information to uncover parallelism. The hierarchical method always uncovers parallelism between task nodes that are totally ordered by the control flow (i.e., task nodes that form a chain graph). Some of the task nodes in the chain graph may be compound nodes; parallelism within such nodes is recursively uncovered. Since the task graph of the program is parsed by the structured graph grammar, every compound node must be an FJ-reducible subgraph. Such graphs have the property that the fork node dominates all nodes in the graph and the join node postdominates all the nodes in the graph. This implies that the nodes in an FJ-reducible subgraph can only be control dependent on the fork node, and not on any other node outside the subgraph. Thus, algorithm FIND_PARALLELISM implicitly satisfies all control dependences by visiting the nodes of the task graph in a hierarchical manner, since an FJ-reducible subgraph at a given level is always represented as a single compound node.
Pipelining the iterations of a loop is a special case of the pipelining technique discussed here, and was first proposed and studied by Cytron [Cyt86]. Task parallelism and piping techniques were also developed by Polychronopoulos [Pol86]. His techniques are guided by static estimates of run-time performance.

3.9 Summary

In this chapter the Data Window representation was used to compute dependences, which were used for detection of parallelism. The parallelism detection procedure uses a structured graph grammar to parse the task graph of the program, and reverses the reductions of the parse to methodically traverse the nodes. Parallelism uncovered at upper levels is of a coarse granularity, while relatively finer grain parallelism is uncovered at deeper levels. A technique for pipelining tasks was also presented, and used to enhance the parallelism detected in the program.
Chapter 4

Analyzing synchronization

To support parallel programming, languages are often extended to include constructs that allow partitioning and synchronization to be specified explicitly in the program. Unfortunately, there is no guarantee that a parallel program expressed in an extended language is free of errors. Particularly difficult to diagnose are errors that introduce nondeterminacy in the program's execution, where different runs of the program on the same input data may yield different results. Errors of nondeterminism usually arise from race conditions that have been inadvertently introduced by the programmer. A race condition exists between two statements if they can simultaneously access the same memory location, and at least one of them writes into that location. This can occur due to the absence of synchronization or synchronization that is specified incorrectly by the programmer.

This chapter presents a static analysis technique that can be used to determine if a given parallel program with synchronization is free from race conditions, and if not, to identify all potential race conditions in the program.

Since nondeterministic race conditions can occur only in the "parallel" parts of a parallel program, we concentrate only on these parallel segments. A parallel program segment is represented as parallel threads of control, all executing simultaneously. Each thread is referred to as a task. The PARALLEL SECTIONS construct is an example of a parallel program segment. The individual SECTIONs constitute the tasks.

Tasks are synchronized using EVENT variables. Only three operations can be performed on EVENT variables. They are: POST EVENT, WAIT EVENT and
CLEAR EVENT. The semantics of these constructs were discussed in chapter 1. For brevity, these are denoted as post wait and clear respectively. Access to a global variable can be serialized by enclosing it in a CRITICAL SECTION.

4.1 Simplifying assumptions

The analysis technique is presented by starting with very restricted parallel programs, gradually relaxing the restrictions to extend the technique to more general cases. It is assumed that no nested parallelism is specified in the program. This assumption is made primarily to simplify the treatment of the problem; the technique can be extended to include nested parallelism. The purpose of this chapter is to focus on the problem of synchronization across tasks. Problems involving synchronization within the same task, such as between iterations of a loop, can be treated as a special case, and are not dealt with here. Finally, it is assumed that event variables are never reused, restricting the analysis to programs that do not contain clear statements. This is justified because the complications that arise in the analysis due to reuse can be avoided by having different event variables for every synchronization. Although this costs some space, it makes the analysis much easier.

The following sections progressively examine synchronization for the following restricted programs:

- Initially, each task is restricted to be a sequence of simple assignment statements and print statements. No branching constructs such as if-then-else or loops are allowed. In addition, all event variables must be scalars.

- Allow if-then-else branch statements in tasks.

- Allow loops with event arrays (i.e., an array where each element is an event variable). The loops may not contain branching constructs or references to scalar event variables.
PARALLEL SECTIONS

SECTION T₁

\[
\begin{align*}
    x &= 1 \\
    \text{post eval1} \\
    y &= x + 1 \\
    \text{wait eval4} \\
    \text{print } x, y, z
\end{align*}
\]

SECTION T₂

\[
\begin{align*}
    w &= 2 \\
    \text{wait eval1} \\
    z &= x + w \\
    \text{post eval2} \\
    \text{wait eval3} \\
    \text{print } x, z
\end{align*}
\]

SECTION T₃

\[
\begin{align*}
    v &= 3 \\
    \text{wait eval2} \\
    x &= v + z \\
    \text{post eval3} \\
    z &= z + v \\
    \text{post eval4}
\end{align*}
\]

END PARALLEL SECTIONS

(a) Parallel Program

(b) Program Execution Graph

(c) Co-graph

Figure 4.1 (a) A parallel program segment, (b) its Program Execution Graph and (c) its co-graph.
We finally comment on how the technique can be extended to general parallel programs without any of the above restrictions.

4.2 Representing synchronization and control flow

A structure called the Program Execution Graph is used as an internal representation of synchronization and control flow in a parallel program segment. The Program Execution Graph is a directed graph $G = (B, E)$, where $B$ is a set of extended basic blocks and $E = E_c \cup E_s$ is the set of control flow edges $E_c$ and synchronization edges $E_s$. An extended basic block is a sequence of statements, all of which are executed if the first statement in the block is executed. It may contain only one wait and one post statement, in which case they must be the first and last statement respectively. Henceforth, these are referred to as "basic blocks".

Critical sections in the program (specified by the CRITICAL SECTION statement) only enforce serialization, not synchronization; so statements within critical sections are treated just like any other statement when constructing the Program Execution Graph. Thus, the presence or absence of critical sections in the program do not affect the outcome of the analysis described here. The Program Execution Graph is similar in structure to the Synchronized Control Flow Graph proposed by Callahan and Subhlok for the representation of synchronization and control flow [CS88]. Figure 4.1 shows a parallel program and its Program Execution Graph.

4.3 The co-graph

Given a Program Execution Graph $G = (B, E)$, the co-graph of $G$, denoted by $\overline{G}$, is an undirected graph whose nodes are the same as that of $G$. Two nodes are connected in the co-graph $\overline{G}$ if and only if there is no path between them in $G$. The co-graph of the Program Execution Graph in Figure 4.1(b) is shown in Figure 4.1(c).
(a) Program Execution Graph $G$  
(b) Co-graph $\overline{G}$  

(c) Concurrency Graph $\text{conc}(G)$  

Figure 4.2 Three graph representations of a parallel program.
4.3.1 Determinacy of parallel programs

The goal is to determine whether a given parallel program is free from race conditions. Such a parallel program is said to be *determinate*, because every execution schedule for a given input will produce the same result. A determinate program may not be correct however. For example, every execution schedule for a particular input data may cause a floating point exception to occur. However, once the programmer has removed all sources of non-determinacy from the parallel program, conventional breakpoint debugging techniques can be used, because the execution schedule that led to an error can always be reproduced.

The run-time profile of a process executing in a multiprogrammed multiple processor system can vary significantly from one run to another. Events such as a context switch, page fault, network congestion, memory latency, scheduling and synchronization overhead, can make the observed execution time of a process greater than its statically estimated time. If two basic blocks in the Program Execution Graph are not explicitly ordered by some control or synchronization path, we cannot statically predict the order in which they will execute. Clearly, consistency can be guaranteed if and only if no such pair of basic blocks conflict, i.e., contain statements that are involved in race conditions.

The above discussion leads to the following observation:

**Observation 4.1** Let \( P \) be a parallel program segment and \( G \) be its Program Execution Graph. Then \( P \) is consistent if and only if there is no conflict between any pair of adjacent blocks in the co-graph \( \overline{G} \).

4.3.2 Constructing the co-graph

The algorithm for constructing the co-graph takes as input a Program Execution Graph \( G = (B, E) \), represented by adjacency lists, and produces an adjacency matrix for its corresponding co-graph, \( \overline{G} = (B, \overline{E}) \). Algorithm BUILD_COGRAPH (Figure 4.3) uses a transitive closure algorithm due to Eve and Kurki-Suonio [EKS78],
an adaptation of Tarjan’s strongly-connected region finding algorithm [Tar82] that runs in $O(ne + n^2)$, where $n = |B|$ and $e = |E|$. All other operations in the algorithm can be done in $O(n^2)$ time, so that the transitive closure dominates the expense.

If the Program Execution Graph is sparse, this algorithm will run much faster than ordinary bit matrix transitive closure. In particular, if $e = O(n)$, the entire computation will run in time $O(n^2)$. The adjacency matrix representing $\overline{E}$ will be symmetric about the principal diagonal, because $\overline{G}$ is an undirected graph.

4.3.3 Detecting race conditions using the co-graph

For each PARALLEL SECTIONS statement $P$ in the input parallel program, a Program Execution Graph $G$ and co-graph $\overline{G}$ are constructed. For each block $x$ in $G$, the Data Window $\Delta(x)$, that describes the portion of the program’s data structures that are accessed within the block $x$, is determined using the techniques discussed in Chapter 2. Two blocks $x$ and $y$ conflict if $\Delta(x) \cap \Delta(y) \neq \emptyset$. If the two blocks access different portions of the same array variable, the intersection of their Data Windows will be empty, indicating that the blocks do not conflict.

If, instead of the above method, traditional dependence tests are used to check for conflict, the dependences must be computed with respect to some chosen sequential ordering of the statements of the parallel program. The particular sequential order

---

Algorithm BUILD_COGRAPH

Input: A Program Execution Graph $G = (B, E)$, with $E$ specified by adjacency lists.
Output: The corresponding co-graph $\overline{G} = (\overline{B}, \overline{E})$, with $\overline{E}$ specified as an adjacency matrix.

begin
    Temp $\leftarrow$ TRANSITIVE_CLOSURE($E$)
    $\overline{E} \leftarrow \neg (\text{Temp} \lor \text{Temp}^T)$
end

Figure 4.3 Algorithm to construct the co-graph.
chosen should be one that preserves the partial ordering imposed by the control and synchronization in the program. By defining conflict in terms of intersections of Data Windows, the complication of choosing the sequential order is avoided, because the analysis is not based on traditional data dependence information.

**Example 4.1:**
Consider the parallel program segment of Figure 4.1(a). When all adjacent blocks in its co-graph (Figure 4.1(c)) are analyzed for conflict, all but two pairs of blocks will be conflict-free. Blocks 2 and 8 conflict in their access to variable x. Block 2 reads the variable x, and block 8 writes into x. Since these two blocks are connected in the co-graph, there exists some parallel execution of the program where they can coexist simultaneously on different processors. This could yield a nondeterministic value for y, depending on the order in which blocks 2 and 8 access the variable x (see Figure 4.1(b)). Blocks 6 and 9 conflict in their access to variable z. The value of z printed in block 6 depends on the order in which the two blocks access z. The parallel program in Figure 4.1(a) is therefore not determinate. <

Any static technique must be conservative in its analysis, and assume a conflict unless its absence can be proved with certainty. All co-graph edges along which absence of conflict can be proved are removed, thus trimming down the number of edges in the co-graph. The remaining edges may indicate actual race conditions or may be the result of conservative analysis. These remaining potential conflicts can either be verified by the programmer or detected at run-time using techniques such as those proposed by Emrath and Padua [EP88], Schonberg [Sch89], Miller and Choi [MC88] or Allen and Padua [AP87].

Although run-time analysis is more precise than static analysis, it has some drawbacks. Run-time traces can be very large, and the analysis is restricted to a particular execution instance of the program, on a particular input data set. It cannot guarantee that the program is determinate for every possible parallel execution and every choice
of input data. Several methods have been proposed to keep the size of run-time traces small and manageable. Many of the proposed run-time trace analysis techniques rely on keeping track of the concurrent states that the program goes through during its execution. As the next section shows, the co-graph can be used as a representation of the concurrent states of a parallel program.

4.4 The co-graph as a graph of concurrent states of a program

Let us define a concurrent state of a parallel program to be an n-tuple \((B_1, B_2, \ldots, B_n)\), where \(n\) is the number of tasks, and \(B_i\) is the basic block that is active (being executed) in task \(T_i\). A block that is waiting for an event variable to be posted is denoted by placing a "\(^\wedge\)" above the block number. A "\(^\cdot\)" indicates that the task has completed execution. Given a Program Execution Graph \(G\), we can derive the set of all possible concurrent states for the program. The transition between the concurrent states can be represented by a directed graph \(conc(G)\), called the concurrency graph of \(G\). The
nodes of the concurrency graph are the concurrent states of the program, and the edges indicate the transition from one concurrent state to another.

The concurrency graph for the program execution graph in Figure 4.2(a) is given in Figure 4.2(c). The node \( (2, 4, \hat{8}) \) represents the concurrent state of the program where task \( T_1 \) is executing block 2, task \( T_2 \) is executing block 4 and task \( T_3 \) is waiting to start block 8. This concurrency graph is a compressed form of the actual concurrency graph; several nodes have been omitted because they represent redundant or invalid concurrent states. The circled nodes represent concurrent states where race conditions exist. To detect such states, it is necessary to examine every node in the graph.

The concurrency graph is a variant of the concurrency history graph (CHG), originally proposed by Taylor [Tay83b] to represent all possible concurrent states of an ADA program. Appelbe and MacDowell [AM85] extended this idea to Fortran programs and suggested several techniques to reduce the size of the CHG. The problem with any representation that explicitly maintains concurrent states is that the number of nodes and edges are exponential in the size of the program. Thus, although the CHG provides detailed information about the possible concurrent states during a program's execution, it requires exponential storage. Moreover, algorithms that traverse such a graph may be prohibitively expensive to implement in a real system.

The co-graph avoids these problems by not representing each concurrent state explicitly. Instead, the number of nodes is equal to the number of blocks, and every pair of blocks that can run concurrently are connected by an edge. Since the edges do not represent transitions between different concurrent states, they do not grow exponentially with the number of blocks, but rather as the square of the number of blocks.

It can be shown that the Program Execution Graph and co-graph can also be used as a representation of the concurrent states of a parallel program:

**Lemma 4.1** The Program Execution Graph and co-graph pair is equivalent to the concurrency graph, in the sense that each can be derived from the other.
Proof:

Part 1: Given the program execution graph $G$ and its co-graph $G^c$, derive the equivalent concurrency graph $conc(G)$.

Given a concurrent state $S$, the following procedure will generate the set of all valid successor concurrent states:

- Let $S$ be denoted by the $n$-tuple $(b_1, b_2, \ldots, b_n)$. From the control flow subgraph of the Program Execution Graph $G$, determine the immediate successors for each $b_i \in S$. Each $b_i$ can have at most two possible successor blocks, say $b_i'$ and $b_i''$ (this happens if $b_i$ is a conditional branch) \(^1\).

- Generate all $3^n$ possible $n$-tuples of the set of all combinations of the $n$ positions, formed by choosing one of $b_i, b_i'$ and $b_i''$ for the $i$th position.

- Not all of these represent valid successor concurrent states, because we have ignored the ordering constraints imposed by the synchronization edges between tasks. A concurrent state $(x_1, x_2, \ldots, x_n)$ is valid if each $x_i$ is connected to all other $x_k, 1 \leq k \leq n, k \neq i$ in the co-graph $G^c$. Using the co-graph in this manner, all invalid states are discarded.

- The remaining $n$-tuples represent the set of possible successor concurrent states of $(b_1, b_2, \ldots, b_n)$.

Starting with the initial state of the parallel program, this procedure can be applied recursively to generate all possible concurrent states of the program. At each step, when the successor states for a concurrent state are determined, a portion of the concurrency graph can be incrementally constructed by adding the successor nodes and the appropriate edges from the parent state to its successors. This proves the

\(^1\)For simplicity, it is assumed that CASE statements are disallowed within a task (i.e., within a SECTION clause of a PARALLEL SECTIONS statement). The procedure can easily be extended to include CASE statements.
first part of the lemma by giving a procedure to construct a concurrency graph from
the co-graph.

\textit{Part 2:} Given the concurrency graph $\text{conc}(G)$, derive the corresponding co-graph $\overline{G}$.

The following is an outline of the procedure:

- The nodes of the co-graph are exactly the same as the program execution graph $G$.

- Traverse all nodes of the concurrency graph, and for each concurrent state $\langle b_1, b_2, \ldots, b_n \rangle$, add a co-graph edge from each $b_i$ to every $b_k that isnotawatingstate, 1 \leq k \leq n, k \neq i$. The edge is not added if one already exists.

It is easy to see that the above procedure constructs the co-graph $\overline{G}$. This proves the second part of the lemma. \qed

The above lemma implies that the Program Execution Graph and co-graph pair can serve as a substitute for the concurrency graph, and used as a representation of the concurrent states of a parallel program. This representation needs storage that is polynomial in the number of basic blocks, compared to the concurrency graph that requires exponential storage. However, the concurrency graph permits the easy reconstruction of events that lead to a conflict state, while the co-graph does not.

\subsection{Multiple posts and conditional branches}

When more than one task does a post on the same event variable, all but the first are redundant, because subsequent posts cannot change the state of the event variable (recall that we do not allow clear statements in our model). In general, it is not possible to predict which post will actually change the state of the event variable in any given execution of the parallel program. The analysis must be conservative and not make any assumptions about which post will occur first. Either all synchronization
edges involving such event variables can be ignored when constructing the co-graph, or the multiple posts can be replaced by a single post in their least common ancestor node [CS88, EP88].

The presence of conditional branches in the tasks further complicates the analysis. If there is a possibility of some task executing a wait on an event variable, then unless another task posts it, the program may enter an infinite wait state, where no progress is made in computation. If all control flow paths through a given task contain a post on an event variable, the task must post that event variable (Figure 4.4(a)). If there is some control flow path through the task that does not contain a post on the event variable, the task may post that event variable (Figure 4.4(b)).

4.5.1 Deadlock and infinite wait

To avoid the possibility of infinite wait, any system of parallel tasks, must have at least one task that must post each event variable on which a wait is executed. In the absence of this restriction, it may be impossible to decide whether a given program will enter an infinite wait state.

Example 4.2:
Consider the example in Figure 4.4(c). Task $T_2$ will block on the wait at $W$ if condition $c_2$ evaluates to true. Both tasks $T_1$ and $T_3$ have may posts at the points labeled P. Assuming there are no other posts on that event variable, it is easy to see that if the boolean expression $c_2 \land \neg c_1 \land (c_3 \land \neg c_4)$ evaluates to true, the system may wait indefinitely in $T_2$. In Figure 4.5(a), the system of tasks will always enter an infinite wait state. $<$1

Deadlock is traditionally defined as a state of circular wait (i.e., the set of deadlocked tasks are each waiting for the other to post an event variable). The possibility of deadlock is illustrated in Figure 4.5(b). If c evaluates to false, the system will
not deadlock. The post labeled P3 in task $T_3$ will simply have no effect, because its corresponding wait $W_3$ will never be executed.

Infinite wait cannot occur in a system that has a must post for every event variable. However, this restriction is ineffective in preventing circular wait. The possibility of a circular wait can be detected by looking for cycles in the program execution graph, that involve at least two synchronization edges. These cycles can be identified by making a simple modification to algorithm BUILD_COGRAPH, so that it marks nodes involved in such cycles. The existence of a cycle does not mean that every execution instance will deadlock, since deadlock may depend on taking a specific control flow path at run time, as it does in Figure 4.5(b).

### 4.6 Loops and event arrays

Consider a parallel program in which some tasks have loops, in particular, loops that do not contain any branching constructs or nested loops. The loops are assumed to
be normalized, and for simplicity all the loops are assumed to have the same upper bound and only array variables may be referenced within them.

It is also assumed that if control can reach the top of the first block in the loop, then the loop must execute at least one iteration. If there is a post of a scalar event variable in the loop, then by definition of the post statement, all subsequent instances of the statement beyond the first iteration are redundant. Similarly, if the loop contains a wait on a scalar event variable, only the first iteration of the loop will block until the variable is posted. Subsequent iterations will not block at that wait statement. Thus, the first iteration of the loop is peeled, and all scalar posts and waits occurring within the loop are removed, the semantics of the program will remain unchanged. Scalar event variables within loops are handled in this way.

Let us now focus on the problem of event arrays that are used to synchronize accesses to shared array variables in loops. For simplicity, event arrays are restricted to have subscripts of the form I + \( \alpha \), where I is an induction variable and \( \alpha \) is a constant (\( \alpha \) cannot be negative, because the loop is assumed to be normalized, and no conditional branches are allowed in the loop).

If a block \( b \) is contained in a loop, its instantiation on the \( i \)th iteration is denoted as \( b[i] \).

**Definition 4.1 (Conflict distance):** Given two blocks \( b_1 \) and \( b_2 \), and an array variable \( A \) accessed in both, the conflict distance for \( A \) from \( b_1 \) to \( b_2 \), is an integer \( k \), such that \( b_1[i] \) conflicts with \( b_2[i+k] \) for any \( i \) in the iteration space of the loop containing \( b_1 \).

**Example 4.3:**
Consider two blocks \( b_1 \) and \( b_2 \), contained in loops in separate tasks. Let \( b_1 \) store into the location \( A(I+2) \) and \( b_2 \) read the location \( A(J) \), where \( I \) is the induction variable of the loop containing \( b_1 \) and \( J \) is the induction variable of the loop containing \( b_2 \). When the \( i \)th instance of \( b_1 \) executes simultaneously with the \( (i+2) \)th instance of \( b_2 \), the blocks will conflict on their access to \( A \). The execution instance of \( b_2 \), relative to the \( i \)th execution instance of \( b_1 \) that causes a conflict is the \( (i+2) \)th instance. The
conflict distance for $A$ from $b_1$ to $b_2$ is thus equal to 2 (or equivalently, the conflict distance for $A$ from $b_2$ to $b_1$ is -2). \(<\)

The notion of conflict distance is analogous to the concept of dependence distance for data dependences crossing the iterations of a single loop. When there are several array variables that are accessed in both blocks, a conflict distance set, \(\overline{CD}(b_1, b_2)\), is computed from block $b_1$ to block $b_2$. Each array variable occurring in both blocks will have a corresponding component in the conflict distance set, that gives the conflict distance for that variable from $b_1$ to $b_2$.

**Example 4.4:**
Consider the Program Execution Graph of Figure 4.6(a). Let $I$, $J$ and $K$ be the induction variables of the loops in tasks $T_1$, $T_2$ and $T_3$ respectively. The loops are assumed to be normalized (i.e., the lower bound and step size of each loop is 1). Suppose block $a$ stores into $A(I+1)$, block $c$ stores into $A(J+2)$, and block $f$ stores into $A(K)$. The conflict distance sets between the blocks are as follows: \(\overline{CD}(a, c) = \langle -1 \rangle\), \(\overline{CD}(c, f) = \langle 2 \rangle\) and \(\overline{CD}(a, f) = \langle 1 \rangle\). \(<\)

Instance $b_1[i]$ always precedes instance $b_2[i + k]$, denoted $b_1[i] \prec b_2[i + k]$ if the program synchronization ensures that $b_2[i + k]$ cannot begin execution before $b_1[i]$ for all $k$.  

**Definition 4.2 (Safe distance):** Let $b_1$ and $b_2$ be blocks contained within separate loops in the Program Execution Graph. The safe distance from $b_1$ to $b_2$, denoted $SD(b_1, b_2)$, is the minimum $k$ for which $b_1[i] \prec b_2[i + k]$ for all $i$ in the iteration space of the loop containing $b_1$. If there is no synchronization relationship between the two blocks, $SD(b_1, b_2) = \infty$.

The safe distance is used to determine if the given synchronization using event arrays resolves all potential conflicts. The safe distance between adjacent blocks in the Program Execution Graph can be determined in a straightforward manner. However, determining the safe distance $SD(a, f)$ is more difficult, because the synchronization
between the blocks a and f is of a transitive nature. A general solution is now presented, to solve the problem of determining whether any two blocks in separate tasks can conflict.

Assign a label to each edge in the Program Execution Graph of Figure 4.6(a) in the following manner:

- All back edges are assigned a label of 1.

- A synchronization edge from a statement of the form post evar(I + α) to a statement of the form wait evar(J + β) is assigned a label of α − β, where I, J are normalized loop induction variables and α, β are constants.

- All remaining edges within loops are assigned labels of 0.

If blocks x and y are connected by an edge, the label denoted l(x, y) is the earliest execution instance of block y, relative to a given instance of block x that must wait until the given instance of block x has completed execution. In Figure 4.6(a) l(c, f) =
2 indicates that the \((i + 2)\)th instance of \(f\) must wait until the \(i\)th instance of \(c\) has completed, for any \(i\). The definition of conflict distance reveals the purpose of such a labeling. The label on an edge tells us what conflict distances are "safe" (i.e., what conflicts are resolved by the ordering imposed by that edge).

The following lemma gives a method for computing safe distances from the labeled Program Execution Graph.

**Lemma 4.2 (Safe distance lemma)** Let \(b_1\) and \(b_2\) be blocks contained within separate loops in the Program Execution Graph. The safe distance from \(b_1\) to \(b_2\), \(SD(b_1, b_2)\), is the minimum over all paths from \(b_1\) to \(b_2\) of the sum of the labels of the edges in the path. If there is no path from \(b_1\) to \(b_2\), \(SD(b_1, b_2) = \infty\).

**Proof:** Let \(k\) be the sum of the labels along some path from \(b_1\) to \(b_2\). By definition, we must have \(b_1[i] < b_2[i + k]\). Now suppose there are two paths with sums \(k_1\) and \(k_2\). By definition, \(b_1[i] < b_2[i + k_1]\) and \(b_1[i] < b_2[i + k_2]\). Without loss of generality, assume \(k_1 < k_2\). Then \(b_1[i] < b_2[i + k_1] \Rightarrow b_1[i] < b_2[i + k_2]\). Taking the minimum gives the correct safe distance. \(\Box\)

The importance of SD is shown in the following lemma:

**Lemma 4.3 (Safety lemma)** If blocks \(b_1\) and \(b_2\) are blocks occurring in loops in separate tasks, there will be no conflict between them if and only if for every component \(z \in \overline{CD}(b_1, b_2)\), either \(z \geq SD(b_1, b_2)\) or \(-z \geq SD(b_2, b_1)\).

**Proof:** Let \(SD(b_1, b_2) = \lambda_1\) and \(SD(b_2, b_1) = \lambda_2\). If blocks \(b_1\) and \(b_2\) are connected by a path, \(SD(b_1, b_2)\) gives the earliest execution instance of \(b_2\), relative to a given instance of \(b_1\) that must wait for the given instance of \(b_1\) to complete. That is, \(b_1[i] < b_2[i + \lambda_1]\) for all \(i\). Let the conflict distance for a variable from \(b_1\) to \(b_2\) be \(d\) (the conflict distance from \(b_2\) to \(b_1\) is \(-d\)). If \(d \geq \lambda_1\), no conflict is possible because the \(i\)th instance of \(b_1\) will always complete before the \((i + d)\)th instance of \(b_2\) begins. There is a path consisting of control and synchronization edges that will preserve this separation between the instances of blocks \(b_1\) and \(b_2\) during any execution of the program.
If \( d < \lambda_1 \), there is no path from \( b_1 \) to \( b_2 \) that prevents conflict, but there may be one from \( b_2 \) to \( b_1 \). In that direction, the conflict distance becomes \(-d\) and the conflict is impossible if \(-d \geq \lambda_2\), since \( b_2[i] \prec b_1[i - \lambda_2] \) for all \( i \). If \( d < \lambda_1 \) and \(-d < \lambda_2\), then \( b_1 \) and \( b_2 \) can conflict on their access to that variable, since the \( i \)th instance of \( b_1 \) can coexist with the \((i + d)\)th instance of \( b_2 \) in some parallel execution of the program. Since each component of the conflict distance set \( \overline{CD}(b_1, b_2) \) represents the conflict distance of a variable occurring in both blocks, the statement of the lemma follows.

\( \square \)

Figure 4.6(b) shows the safe distances between pairs of blocks for the Program Execution Graph of Figure 4.6(a) in the form of a matrix. The matrix entry \( SD(i, j) \) gives the safe distance from block \( i \) to block \( j \).

**Example 4.5:**

In Figure 4.6(a), the event array \( e_1 \) ensures that block \( c \) always "lags behind" block \( a \) by at least one iteration, so that \( a[i] \prec c[i - 1] \). Thus, \( SD(a, c) = -1 \).

If block \( a \) stores into \( A(I+1) \) and block \( f \) stores into \( A(K) \), \( \overline{CD}(a, f) = \{1\} \). Since \( SD(a, f) = 1 \), and each component of \( \overline{CD}(a, f) \) is greater than or equal to 1, blocks \( a \) and \( f \) will never conflict on access to \( A \). If, however, the store to \( A(K) \) in block \( f \) were replaced by a store to \( A(K-1) \), the two blocks can conflict because the safe distance \( SD(a, f) \) now exceeds the conflict distance for \( A \) between the two blocks. \( \angle \)

### 4.6.1 Deadlock revisited

Detecting the possibility of deadlock becomes much more difficult when event arrays are involved. Figure 4.7 shows a parallel program segment where both tasks will eventually deadlock. The sequences of posts and waits in tasks \( T_1 \) and \( T_2 \) before they reach deadlock are listed below:
Figure 4.7  Deadlock due to incorrect use of event arrays.

\[
\begin{align*}
T_1 & \\
\text{post } e_1(1) & \quad \text{post } e_2(2) \\
\text{wait } e_2(4) & \quad \text{wait } e_1(1) \\
\text{post } e_2(3) & \\
\text{wait } e_1(2) &
\end{align*}
\]

$T_1$ is blocked waiting for $T_2$ to post $e_2(4)$, while $T_2$ is blocked waiting for $T_1$ to post $e_1(2)$. Notice that if the "wait $e_2(I+3)$" in block b is replaced by "wait $e_2(I+2)$", the tasks will not deadlock. The safe distance matrix is useful in determining when a set of tasks synchronized using event arrays can deadlock. In Figure 4.7, the diagonal entries in the safe distance matrix $SD$ are all 0. This implies that for all blocks $b$, $b[i] < b[i]$, which is impossible. The diagonal entries of the safe distance matrix of Figure 4.7 are thus inconsistent with the definition of safe distance, indicating that deadlock is a possibility.
4.7 Putting it all together

After beginning with a restricted form of parallel program, the restrictions were gradually relaxed to allow branching constructs, multiple posts and waits, and finally loops with event arrays that do not contain any branching constructs.

The construction of the co-graph for a very simple Program Execution Graph without loops and conditional branches was discussed earlier. In subsequent sections, the following complications to this procedure were introduced.

- When conditional branches are introduced, algorithm BUILD_COGRAPH must be modified so that blocks on separate branches of the same conditional statement in a task are not connected by a co-graph edge.

- If there are multiple post statements on the same event variable, all synchronization edges on that event variable must be ignored when constructing the co-graph.

- If an event is posted on only one branch of a conditional statement, the possibility of infinite wait must be checked for, reporting a potential error if it cannot established that at least one task must post the event variable.

- The sufficient condition for deadlock and the list of wait statements that may block forever should be determined and reported.

- When loops without any post or wait statements are encountered the entire loop can be treated as a single block when constructing the co-graph.

- If scalar event variables occur in the loop, the first iteration of the loop is peeled and all occurrences of the scalar event variables are removed from the loop. The co-graph can then be handled as if it contains no scalar event occurrences.

- If event arrays occur in the loop, the edges in the Program Execution Graph are labelled and safe distances are determined.
In the remainder of this section, a method is given to unify the treatment of blocks within loops and blocks not enclosed by a loop.

### 4.7.1 Safe distances for the whole program

Let us assign a label of 0 to all control and synchronization edges that do not involve blocks within loops. Edges associated with a loop are labelled as specified in section 7. For completeness, the definition of safe distance can be extended to blocks that are not contained in loops, to be the minimum over all paths from one block to the other of the sum of the labels of the edges in the path. Based on this extended definition, the safe distance matrix for the entire Program Execution Graph is constructed as follows.

- If two blocks, \( x \) and \( y \), have the property that \( \text{SD}(x, y) = \text{SD}(y, x) = \infty \), there can be no path between \( x \) and \( y \) in the Program Execution Graph. Therefore, \( x \) and \( y \) can coexist in some parallel execution of the program.

- If \( \text{SD}(x, y) \) is a non-zero number (and not \( \infty \)), four cases are possible:

  1. Neither of the two blocks are contained within a loop, but the path from \( x \) to \( y \) goes through some loop. Since \( x \) and \( y \) have only a single execution instance each, a non-zero safe distance from \( x \) to \( y \) is meaningless. However, the existence of a path indicates that the \( x \prec y \), so we set \( \text{SD}(x,y) \) to \( -\infty \) for such blocks.

  2. Block \( x \) is within a loop, and block \( y \) is not. Let \( I \) be the induction variable of the loop containing block \( x \), and let \( \text{SD}(x,y) = k \). Let \( n \) denote the upper bound of \( I \). A safe distance of \( k \) indicates that the the first through \( n - k \) execution instances of \( x \) must occur before the only execution instance of \( y \) can begin (i.e., \( x[1 : n - k] \prec y \)). Therefore \( \text{SD}(x,y) \) is changed to \( n - k \). Since a negative value of \( k \) is meaningless in this context, \( \text{SD}(x,y) \) is
changed to 0 when \( k \) is negative. This indicates that block \( y \) cannot start executing until all instances of block \( x \) have completed.

3. Block \( x \) is not in a loop but block \( y \) is. Here, a safe distance of \( k \) indicates that \( x \prec y[k : n] \), so \( \text{SD}(x,y) \) can remain equal to \( k \). A negative value of \( k \) is meaningless however, so we set \( \text{SD}(x,y) \) to 0 when \( k \) is negative, indicating that block \( x \) and all instances of block \( y \) can never coexist.

4. Both blocks are contained within loops. This case was dealt with in detail in an earlier section. \( \text{SD}(x,y) = k \) indicates that the \( i \)th instance of \( x \) and the \((i + k)\)th instance of \( y \) can never coexist for all \( i \).

4.7.2 Safe distances and the co-graph

Let the minimum safe distance between two blocks \( x \) and \( y \) be the minimum of \( \text{SD}(x,y) \) and \( \text{SD}(y,x) \), and denote it by \( \text{minSD}(x,y) \). The minimum safe distance denotes the minimum conflict distance between \( x \) to \( y \) and \( y \) to \( x \) that is guaranteed by the existing synchronization in the program.

By the safety lemma, there will be no conflict between \( x \) and \( y \) if and only if for every component \( z \in \overline{\text{CD}}(x,y) \), either \( z \) or \( -z \) is \( \geq \text{minSD}(x,y) \). The general co-graph for a program with loops and synchronization using scalar and array event variables can be constructed using the \( \text{minSD} \) values by the following procedure:

- If \( \text{minSD}(x,y) = \infty \), add a co-graph edge from \( x \) to \( y \) and set its minimum safe distance to be \( \infty \). This indicates that any instance of \( x \) and any instance of \( y \) can coexist in some parallel execution.

- If \( \text{minSD}(x,y) = -\infty \), \( x \) and \( y \) are not connected by an edge in the co-graph, since there is some path in the program execution graph that connects \( x \) and \( y \), which prevents both from executing simultaneously in any parallel execution.

- If \( \text{minSD}(x,y) = 0 \) and either \( x \) or \( y \), or both are within a loop, add a co-graph edge between \( x \) and \( y \) and set its minimum safe distance to be 0.
• If \( \minSD(x,y) \) is other than 0, \(-\infty\) or \(\infty\), add a co-graph edge between \( x \) and \( y \) and set its minimum safe distance to be \( \minSD(x,y) \).

To prove that the parallel program is determinate, the safety lemma must hold, i.e., for every pair of connected blocks \( x \) and \( y \) in the co-graph, the following must be established:

\[ \forall z \in \overline{CD}(x,y), \text{ either } z \text{ or } -z \text{ is } \geq \minSD(x,y). \]

The time to construct the co-graph is dominated by the time required to compute the safe distances, which is \( O(|B|^3) \).

4.8 Related work

The technique presented here has many similarities with one proposed by Emrath and Padua [EP88], and was developed independently. They classify nondeterministic programs as \textit{associatively} and \textit{completely} nondeterminate. Associative nondeterminacy arises due to non-associativity of floating point operations. Making such a distinction is useful, because the programmer may choose to ignore associative nondeterminacy on the grounds that changes in round-off errors between different executions are acceptable. Their \textit{ordering graph} and the Program Execution Graph both label the edges in similar ways. However, each node in the ordering graph is a single statement, whereas each node in the Program Execution Graph is an extended basic block. It is unclear how their analysis is applied in the general setting of sequential statements, loops and conditional branches in a uniform manner (i.e., a single procedure to analyze synchronization for the entire program). It is also unclear how their technique can be used to identify deadlock.

Taylor has discussed the complexity of analyzing synchronization in Ada programs [Tay83a]. His results pertain to Ada's "rendezvous" synchronization, but can be extended to programs with event synchronization.

Shasha and Snir have developed a method to determine the minimal set of delays that can enforce sequential consistency in a parallel program [SS88]. Their technique
involves looking for "minimal inconsistent executions" which indicate portions that need to be synchronized in order to enforce sequential consistency.

Callahan and Subhlok [CS88] have suggested a technique based on a data flow framework to check if the existing synchronization in a parallel program satisfies all data dependences. Given a parallel program, they construct a *synchronized control flow graph*, which is very similar to the Program Execution Graph. They compute "Preserved" sets for each basic block in the synchronized control flow graph:

**Definition 4.3 (Preserved):** If two blocks $b_1$ and $b_2$ are both executed, then $b_1$ is in Preserved($b_2$) if and only if $b_2$ is executing $\Rightarrow b_1$ has been completed.

This implies that if $b_1$ is in Preserved($b_2$), then all data dependences from $b_1$ to $b_2$ are satisfied. Thus, a program is correct if and only if this can be established for *every* dependence that has endpoints in different blocks. The Preserved sets are computed by performing an iterative data flow analysis on the synchronized control flow graph. There are several restrictions on the kinds of parallel programs that can be analyzed using their technique. They assume the absence of loops, so that only synchronization using scalar event variables is allowed. Moreover, the input parallel program must be *serializable* (i.e., there must be a well-defined sequential execution order of the statements in the program). Correctness of the parallel program is defined relative to this sequential execution: if the synchronization in the parallel program satisfies all the data dependences in the sequential version, the program is correct.

The technique presented in this chapter that is applicable to a wider class of programs. Loops and synchronization using event arrays are allowed and the parallel program need not be serializable. Since serializability is not required, satisfiability of data dependences (in the traditional sense) cannot be used as a criterion for correctness, because data dependence is always defined with respect to a sequential program. Instead, the notion of conflict is used, which does not assume any ordering between blocks unless they are explicitly ordered by control or synchronization paths.
However, Callahan and Subhlok's technique is more aggressive in dealing with multiple posts in the presence of conditional branches, while the technique presented here is more conservative in these cases.

The work of Appelbe and McDowell on the concurrency history graph [AMS85] is also related to the co-graph. It was shown that the Program Execution Graph and co-graph pair can also be used as a representation of the concurrent states of the parallel program. This representation needs a polynomial amount of storage compared to the exponential storage required by the concurrency graph. However, the concurrency graph allows easy reconstruction of the concurrent states that lead to a conflict state, while the co-graph does not.

4.9 Summary

In this chapter, a method for the detection of race conditions in parallel programs was presented. The analysis uses the Program Execution Graph as an internal representation of the synchronization and control flow in the parallel program. The co-graph, constructed from the Program Execution Graph, provides a compact representation of all possible concurrent execution schedules of the program. Every pair of adjacent blocks in the co-graph can execute simultaneously in some execution schedule of the parallel program, and these pairs are checked for conflict.

Some of the problems caused by multiple posts in the presence of conditional branches and by improper use of event arrays within loops were identified. Finally, it was shown how the concepts of conflict distance and safe distance are useful in establishing if the existing synchronization in the program guarantees conflict-free accesses to all variables.
Chapter 5

A Parallel Programming Tool

This chapter ties together the concepts developed so far into a framework that can be used as a basis for the design of a parallel programming tool. The chapter begins with a discussion of the design of an internal representation for the program and then describes how the techniques developed in the previous chapters can be integrated together to use this internal representation.

5.1 Internal representation of the program

A programming language is an abstraction that is used to specify instructions to be executed on a processor. While this abstraction allows for ease of programming, it is unwieldy for use in a system that must perform analysis and transformations on the program. Program analysis tools usually maintain an internal representation of the program, which is used by the algorithms that perform the analyses and transformations. The internal representation must reflect the program's structure and control flow and be maintained in a form that can be easily manipulated. A popular internal representation is the control flow graph, a directed graph in which the nodes represent straight-line code called basic blocks and the edges represent flow of control between the basic blocks [ASU86].

In an interactive programming environment, there is another dimension that significantly complicates the design of an internal representation. Not only does the source program have to map to the internal representation, but information from the internal representation must be mapped back to the source program. This is neces-
sary in order to keep the internal representation transparent to the programmer and keep all interaction in terms of the source program.

The ParaScope programming environment uses an abstract syntax tree (AST) as an internal representation [ASU86]. When the programmer views the source program from within an editor in this environment, he actually sees a pretty-printed form of the abstract syntax tree. Each statement in the program maps to a subtree in the abstract syntax tree. Any change made to the source is immediately reflected in the abstract syntax tree, which is updated incrementally in response to an editing change [AK84, CCH+88, BKK+89]. When the programmer selects a region of the program in the editor, the selection is mapped to a subtree in the abstract syntax tree.

Figure 5.1 shows a program segment and its corresponding abstract syntax tree. The small circles denote statements, which can be further expanded into expression trees. The larger shapes denote compound statements. The vertical edges connecting the nodes in the abstract syntax tree represent flow of control between the nodes. The horizontal edges connect compound nodes to the root of the abstract syntax subtree containing their component nodes. Synchronization edges between SECTION nodes within PARALLEL SECTIONS are represented as special edges in the abstract syntax tree.

Such an abstract syntax tree can be treated as a hierarchical representation of the program. Figure 5.1 shows the nodes of the abstract syntax tree partitioned into different groups. The portion labeled $G^1$ is the task graph at level 1. The tasks are represented by the nodes 1, 5, 6 and 15. Of these nodes, 1, 6 and 15 are compound task nodes at level 1, with node 15 already having parallelism specified by the programmer. The compound task nodes can be expanded to get $G^2$, the task graph at level 2. We denote the task subgraph at level $k$ that is rooted at node $x$ by $G^k(x)$.

---

1There is no synchronization in the program of Figure 5.1, so synchronization edges do not appear in its AST.
1 DO
2 PARALLEL DO
3 stmt
4 stmt
5 stmt
6 IF cond
7 THEN
8 CALL sub
9 ELSE BEGIN
10 DO
11 stmt
12 DO
13 CALL sub
14 stmt
15 PARALLEL SECTIONS
16 SECTION
17 stmt
18 SECTION
19 stmt
20 END PARALLEL SECTIONS

Compound nodes:
- DO
- IF
- CALL
- CASE
- PARALLEL DO
- PARALLEL SECTIONS
- SECTION

Figure 5.1 A program and its abstract syntax tree representation.
Level 2 contains the task subgraphs $G^2(2), G^2(7), G^2(8), G^2(16)$ and $G^2(17)$. The compound nodes 2, 8, 16 and 18 can be expanded further to get $G^3(3), G^3(9), G^3(17), G^3(19)$, and so on. The sequence of task graphs $G^1, G^2, \ldots, G^n$ can also be generated by reversing the reductions derived using the graph parsing technique discussed in Chapter 3. The abstract syntax tree is thus an ideal internal representation for parallelism detection under the scheme discussed in this thesis. The following sections discuss how to implement the techniques presented in the previous chapters on this internal representation.

5.2 Parallelism detection

The primary function of a parallel programming tool is to aid the programmer in the parallelization process. The program may be in a partially parallelized state (i.e., the programmer may already have specified some parallelism in the program). The parallelism detection procedure given in Chapter 3 identifies opportunities for parallelism that may have been overlooked by the programmer. The process of parallelism detection is performed by algorithm FIND_PARALLELISM. The realization of algorithm FIND_PARALLELISM in the setting of the abstract syntax tree representation is now discussed.

Several abstractions can be built on the abstract syntax tree representation. One example is the hierarchy of task graphs $G^1, G^2, \ldots, G^n$. Another useful abstraction is the Task Dependence Graph. The Task Dependence subgraph at level $k$ rooted at the node $x$ is denoted by $TDG^k(x)$. The Task Dependence Graph $TDG^k(x)$ consists of the same task nodes as $G^k(x)$ and the same root node $x$. Two nodes $u, v \in TDG^k(x)$ are connected by a directed edge if there is a data dependence from $u$ to $v$.

The Data Window information is initially computed for the nodes that are the leaves of the abstract syntax tree. If a leaf node is a CALL (such as node 12 in Figure 5.1, for example), the Data Window must be computed for the subroutine body
and translated to the call site. The abstract syntax tree is parsed in the following manner:

- Starting from the deepest level of the abstract syntax tree (level 4 in the figure), union the Data Windows of the connected nodes in that level. This gives the Data Window for the compound nodes in the next upper level (level 3 in the figure). For example, the task subgraph $G^4(11)$ at level 4 consists of three nodes, 11, 12 and 13. The union of their Data Windows is the Data Window of the compound node 10 at level 3. That is, $\Delta(10) = \Delta(11) \cup \Delta(12) \cup \Delta(13)$.

- Continuing in this manner, the Data Window for the entire program, $G^0$ is determined. $G^0$ is the level 0 task graph and consists of a single compound node $x$.

- Call PAR.SEARCH(1, $x$) to look for parallelism at each level (PAR.SEARCH is described in Chapter 3). PAR.SEARCH will traverse the nodes from the uppermost level (level 1) to the deepest level (level 4 for the example in Figure 5.1), uncovering parallelism at each level.

When the abstract syntax tree is parsed, the Data Window information collected must be saved at each compound node. Procedure PAR.SEARCH uses the saved Data Window information at a given level to construct the Task Dependence Graph for that level. This Task Dependence Graph is used for uncovering parallelism at that level. Once all nodes at a given level have been visited by procedure PAR.SEARCH, the saved Data Window information at the compound nodes in that level may be discarded and their storage reclaimed.

### 5.2.1 Programmer guided parallelism detection

The decisions made during the automatic parallelism detection procedure may not be very suitable for the particular parallel computer for which the program is targeted. Although there has been some progress in hardware specification languages, code
generation based on a hardware specification is still a difficult and unsolved problem. Instead, we assume that the programmer understands the architecture of the target computer, and is capable of making optimization decisions that are architecture dependent. However, some assistance can be provided to aid the programmer in making the appropriate decisions.

Procedure PAR.SEARCH can be modified so that the programmer can select the region of the program to be parallelized, and also tune the granularity of parallelism desired. A typical programmer-guided session proceeds as follows:

- The level 1 task graph $G^1$ is graphically presented to the programmer. Optionally, the programmer may choose the level of the task graph that he wants to view. If the task graph at the selected level is too large, a movable window is provided which allows the programmer to view any desired portion of the task graph.

- Using the window, the programmer selects the task subgraphs where the programming tool should look for parallelism.

- The tool determines, based on $TDG^1$, whether some of the selected nodes in the current level can be executed concurrently. This set of nodes is reported to the programmer. The nodes are grouped into a PARALLEL SECTIONs only if the programmer approves this action. The programmer can request the tool to continue recursively searching for finer granularity parallelism within the current task graph. If parallelism is uncovered at a deeper level, the programmer may choose to parallelize at the deeper level instead of the upper level.

- At each stage, when any parallelism is discovered, such as a potential PARALLEL DO or PARALLEL SECTIONS, the programmer's approval is necessary for the parallelism to be made explicit in the program. The programmer can also request the tool to find nested parallelism, i.e., continue searching for finer grain parallelism within an already parallel region. If parallelism is found at a deeper
level, the programmer may choose to parallelize at the deeper level instead of
the higher level.

5.3 Parallelism enhancement

There are several transformations that can be applied to enhance the parallelism in a
program. Several techniques for enhancement of loop parallelism are described in the
literature, including transformations such as loop interchange and loop distribution.
Chapter 3 discussed a new transformation called pipelining to enhance parallelism in
the program. Most of these transformations are inter-dependent, in the sense that
one transformation may affect the validity or effectiveness of another. In general, the
problem of deciding the "best" sequence of transformations is a hard problem. Instead
of using crude heuristics to automatically decide the sequence of transformations
attempted during parallelism detection, the programmer is given complete control.

The programmer can select a task node at the current level and request the tool
to give a menu of valid transformations. The programmer chooses the transformation
and the chosen transformation is performed automatically by the programming tool.
Currently, the ParaScope Editor in the ParaScope programming environment provides
such a facility for enhancement of loop parallelism [BKK+89]. After selecting a loop,
the programmer can request a list of valid transformations from a pool consisting of
loop interchange, loop distribution, loop fusion, peeling of iterations, node splitting
and scalar expansion [BKK+89]. This facility can easily be extended for pipelining
transformations. The selected task node is pipelined with its immediate predecessor
in the current level task graph using algorithm PIPELINE (discussed in Chapter 3). If
pipelining is not possible because of a mismatch of traversal orders, the programmer
is advised to try other transformations first, in order to make the traversal orders
match. Pipelining can then be attempted again. Algorithm PIPELINE can also
be modified so that when two loops are pipelined, the programmer can control the
separation by which the iterations of the two loops are staggered. The larger the
separation, the fewer the POSTs and WAITs necessary to perform the pipelining. This reduces the synchronization overhead during execution of the parallel program at the cost of some loss of parallelism.

5.4 Identification of race conditions

In Chapter 4, a technique was suggested for analyzing EVENT synchronization in a parallel program. All programmer specified PARALLEL SECTIONS in the program that contain synchronization can be analyzed for determinacy using this technique. For example, in Figure 5.1, the two SECTION task nodes within the PARALLEL SECTIONS statement may be synchronized in some complex way using EVENT variables. The parallel programming tool can internally construct the co-graph that corresponds to these parallel sections. Any race conditions that are discovered during the analysis can be reported to the programmer.

5.5 Debugging support

The static analysis method given in Chapter 4 may not be able to establish with certainty whether or not some pairs of connected blocks in the co-graph actually conflict. It is conservative in such cases, assuming a conflict unless absence of conflict can be proved with certainty. The edges left in the co-graph after the static analysis may represent real conflicts or may be the result of conservative analysis.

Unfortunately, there may be a large number of edges in the co-graph along which absence of conflict is not provable by static analysis. Reporting all these potential race conditions may result in too much redundant information being presented to the programmer. Thus, run-time techniques are necessary to analyze the possibility of conflicts between blocks connected by edges in the co-graph. When the program is run with input data, these edges can be analyzed again with much more precise information. Several run-time techniques have been suggested in the literature (see Chapter 4).
Once the parallel program has been freed of all race conditions, conventional breakpoint debugging techniques can be used to identify and remove errors in the program. The exact execution schedule that caused the error to occur can be reproduced because the parallel program is determinate.

5.6 Information about program behavior

The simple section field of the Data Access Descriptor has a straightforward graphical representation. The list of inequalities that describe the bounding planes of the accessed section can be used to construct a picture of the array section. If the dimension of the array is larger than 3, the shape can be projected on different 2 dimensional planes.

Thus, the programmer can select a region (such as an DO loop, a call site, etc.) and the accessed section for a specified variable within the selected region can be graphically displayed. Such visual information is very intuitive and useful in understanding program behavior.

5.7 A parallel programming tool

A parallel programming tool designed to use the techniques discussed in this thesis must use the abstract syntax tree as the fundamental internal representation of the program. Each utility provided by the parallel programming tool should be designed as an independent entity that implements a particular function for manipulating the abstract syntax tree (see Figure 5.2). The ParaScope Editor is one such utility. The editor provides an interface between the programmer and the internal representation. The functionality provided by the ParaScope Editor is the mapping between program source and the abstract syntax tree. In response to an editing change, the ParaScope Editor incrementally updates the abstract syntax tree to reflect the change. After performing the change all the other utilities are notified.
Figure 5.2 A parallel programming tool.
The other utilities such as the parallelism detection procedure, the parallelism enhancing transformations and the procedure for synchronization analysis can be invoked from the editor. The parallelism detection and enhancement techniques use the Task Graph and the Task Dependence Graph, which are abstractions built on the abstract syntax tree. When the underlying abstract syntax tree is changed (as in an editing change), these abstractions must also be appropriately modified. The incremental updating of the task graph and the Task Dependence Graph abstractions is a topic of future research, and is not discussed in this thesis.

5.8 Summary

This chapter discussed how the concepts developed in the previous chapters can be brought together to design a parallel programming tool. The basic internal representation of the program is the abstract syntax tree. All the techniques are designed as utilities that can be invoked from an editor. The editor provides the interface between the programmer and the internal representation. A utility, when invoked, may construct an abstraction on top of the abstract syntax tree, and use this to perform its function. For example, the parallelism detection procedure constructs a hierarchy of task graphs and Task Dependence Graphs using a portion of the nodes of the abstract syntax tree. Such a design keeps the separation between the individual utilities provided by the programming tool, while allowing them to share a common internal representation.
Chapter 6

Conclusion

The general theory of data dependence testing presented in Chapter 2 has been used as the basis for the design of several utilities that together form the basis for the design of a parallel programming tool. The notion of data dependence as an intersection between Data Windows is a very useful one. In this concluding chapter, we attempt to present some of the intuition behind the ideas presented in this thesis, the implicit assumptions that we have made, and their justification. We also discuss the impact that these ideas may have on the process of parallel program development.

6.1 Justification for programmer guidance

The analysis techniques presented in this thesis give the programmer control to make decisions that may require complicated heuristics if the analysis were done completely automatically. This section attempts to justify the decision to allow programmer guided analysis.

Scientific applications generally involve modeling of physical phenomena, to study their behavior under certain conditions. Programs written for such applications tend to have localized compute-intensive regions. The part of the program that set up the initial data and the part that collects the result demands relatively small CPU time compared to the computation itself. If these localized regions can be optimized to expose maximal parallelism, it will be possible to achieve impressive speedups. An aggressive parallelizing strategy that tries to uncover all possible parallelism in every part of the program will most likely provide only marginal gain over a strategy that selectively parallelizes these localized compute-intensive regions. The programmer
can be very selective in the regions of the program he wants to parallelize, and request the parallelism detection procedure to work only on these regions.

Every computational problem in science cannot be solved efficiently on parallel hardware. However, it is well known that there are a great many problems that can exploit parallel computation in a very natural way. The programmer understands the algorithm and the nature of the target parallel computer much better than the parallel programming tool. However, the tool can do a much more thorough analysis of the program than the programmer can. Programmer-system interaction can therefore be very useful.

6.2 Justification of parallelization strategy

Currently, most parallelizing compilers and parallel programming tools restrict parallelism detection to loops. The justification for this is that loops contain the majority of the computation in a typical numerical scientific program, so that parallelizing the loops can achieve sufficient speedups. In this thesis, the parallelization strategy attempts to uncover parallelism of different granularities, from the task level to the loop level. This section attempts to justify such a strategy.

When the size of a given problem is increased - such as increasing the number of particles in a gas or lattice, increasing the required precision of a result, or increasing the number of grid points on a mesh - the computational speed required to solve it tends to grow as a power of the problem scale factor. Consider for example, the solution of a partial differential equation over a space that is modeled by a 3 dimensional grid. The accuracy of the solution improves as we increase the number of grid points because of improved resolution in the data. One of the reasons for the computational difficulties growing much faster than the number of grid points is that the time taken for the numerical algorithm to generate “interesting structures” (or correlations) typically grows with the system size [BBK+87].
When we follow the behaviour of a model over a longer span of time, rounding errors tend to accumulate to a greater extent. Gradually, the results start becoming more and more sensitive to trivial changes in the initial conditions. In order to keep the round-off errors from accumulating, it will be necessary to compute the numerical solution with greater degrees of precision. This increases the computational demands much faster than one would intuitively expect. The increased computational demand can only be met by increasing the degree of parallelism in the program. This can be accomplished by looking for parallelism within loops as well as between entire loops or subroutines. This is done by the parallelism detection procedure and the pipelining technique discussed in Chapter 3. Such a strategy is especially effective in programs where the parallelism is not iterative in nature.

6.3 Programmer-tool interaction

We have seen examples of the kind of information that can be communicated from the programmer to the parallel programming tool and from the tool back to the programmer. The tool may require some information from the programmer in order to refine some of its internal representations, so that analysis can be done with greater accuracy. The analysis of the LUDCOMP program in Chapter 2 exemplified the need for this interaction. Knowledge about the index array IPVT can be used to improve the precision of the Data Access Descriptor representations. Programmer input of another form was seen in Chapter 3, where the programmer is involved in the parallelism detection process by guiding the procedure and making simple decisions.

The tool provide various kinds of useful information to the programmer. The Data Access Descriptors computed at various points in the program contain much useful information. Given a region, the section accessed by an array reference within it can be visually presented at a selected point in a region of the program. The programmer can then select an enclosing region, and the tool can automatically translate the section outwards and present the section in terms of the new region. Dependence
between two regions can also be visually presented. The section of an array that is accessed by two dependent regions is represented in the intersection of the respective Data Access Descriptors for the array in the two regions. Information about incorrect synchronization can be illustrated by a visual presentation of the co-graph for the parallel region that has race conditions. The co-graph edges that correspond to race conditions can be shown in a different shade than the ones that are present due to conservative analysis.

6.4 The Task Dependence Graph as an abstract representation of parallelism in the program

The Task Dependence Graph of a program is in some sense a functional partitioning of the program. Each task can be viewed as performing some independent function. The edges of the graph represent the data movement between the tasks. It is generally accepted as good programming style to use separate subroutines to encapsulate entire functions. Fortran programmers tend to use COMMON blocks to represent data that is "shared" by several subroutines. Thus, the Task Dependence Graph is a very intuitive decomposition of a program, in the sense that it is closely related to the notion of a program as being a set of functions performed by various subroutines, their execution order governed primarily by the flow of data from one subroutine to another.

The Task Dependence Graph has another important significance. When trying to partition programs for distributed memory machines (like a hypercube for example), the programmer has to map the partitioned pieces of the program onto the machine's topology in a manner that reduces the amount of communication between the processors. The Task Dependence Graph can be thought of as representing a partitioning of the program with the edges representing data communication that must be communicated between the separate pieces. If two tasks in the Task Dependence Graph are not connected by an edge, no data needs to be communicated between them. In
Chapter 2, we suggested that data dependence represented as an intersection between two Data Windows is essentially a measure of interprocess communication. To run the program on a distributed memory machine, we must map the Task Dependence Graph onto the machine's topology in an "efficient" manner. An algorithm that attempts to find such a mapping must take into account the following factors:

- If two tasks are connected in the Task Dependence Graph, they must be preferably mapped onto adjacent processors. A more general heuristic is to attempt to map connected tasks onto the pair of available processors that requires the least number of "hops" to communicate.

- All tasks should have approximately the same amount of computation to communication ratio. This can be done by merging smaller tasks to form larger ones, or alternatively splitting larger tasks into smaller ones.

- The amount of communication between the tasks should be reduced. If two tasks need a large amount of data to be communicated between them, they can either be merged together and mapped onto a single processor or the data overlap between them can be pipelined using the techniques introduced in Chapter 3.

Clearly, several heuristics are needed to automate such a partitioning procedure for distributed memory machines, and present opportunities for future research.

6.5 Expected impact of this work

Several new ideas have been presented in this thesis. In Chapter 2, we presented a technique for summarizing data access in regions. The Data Access Descriptor contains a lot of useful information and has several interesting properties. In particular, intersection and union operations involving Data Access Descriptors take constant time. A general theory of dependence testing was developed that defined dependence
as an intersection of Data Access Descriptors. This allows data dependence between
arbitrary regions of a program to be treated in the same manner as traditional data
dependences between pairs of statements.

This observation provided the basis for the algorithms for parallelism detection
and enhancement presented in Chapter 3. The algorithms treat different granularities
of parallelism in a uniform manner. This allows the same algorithm to be applied
for uncovering coarse-grain parallelism as well as relatively finer-grain parallelism.
The parallelism detection technique uses a novel method of traversing the program,
uncovering coarse-grain parallelism in the beginning and progressively uncovering
finer-grain parallelism. The algorithm has the important property that all control
dependences are implicitly accounted for, so that control dependence information
need not be computed at all. Chapter 3 also presented a new pipelining technique,
that can be used to achieve partial parallelism between two regions in cases where a
dependence prevented complete parallelism between them.

Chapter 4 presented a technique to analyze programmer specified parallel regions
for race conditions. The techniques use a representation called the co-graph, derived
from the Program Execution Graph. The Program Execution Graph and co-graph
pair can be viewed as a representation of all possible concurrent states of a parallel
program. The representation requires storage that is polynomial in the size of the
program, while other comparative representations require exponential storage.

Chapter 5 presented the design of a parallel programming tool that integrates
all the techniques discussed in the earlier chapters into a set of utilities that can be
invoked from within an editor. The parallel programming tool provides conceptual
interaction in terms of Data Windows, and also has the capability to provide low-level
information.

It is hoped that such powerful parallel programming tools will help lessen the time
and the intellectual effort that is devoted to the efficient parallelization of numerical
scientific programs.
Bibliography


