

This item was submitted to [Loughborough's Research Repository](#) by the author.
Items in Figshare are protected by copyright, with all rights reserved, unless otherwise indicated.

Automatic parallelization of programs

PLEASE CITE THE PUBLISHED VERSION

PUBLISHER

© Md Yazid Mohd Saman

PUBLISHER STATEMENT

This work is made available according to the conditions of the Creative Commons Attribution-NonCommercial-NoDerivatives 2.5 Generic (CC BY-NC-ND 2.5) licence. Full details of this licence are available at:
<http://creativecommons.org/licenses/by-nc-nd/2.5/>

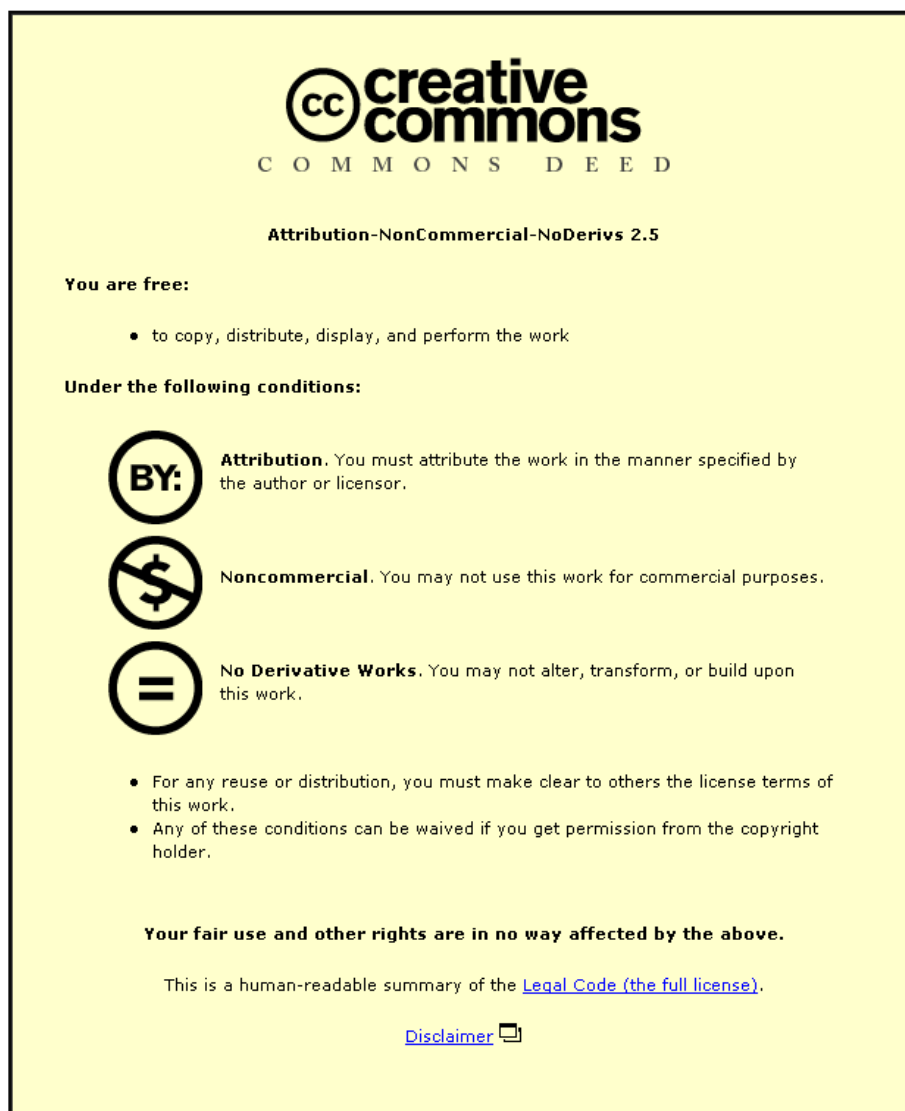
LICENCE

CC BY-NC-ND 2.5

REPOSITORY RECORD

Saman, Mohammad Yazid M.. 2019. "Automatic Parallelization of Programs". figshare.
<https://hdl.handle.net/2134/27235>.

This item was submitted to Loughborough University as a PhD thesis by the author and is made available in the Institutional Repository (<https://dspace.lboro.ac.uk/>) under the following Creative Commons Licence conditions.



For the full text of this licence, please go to:
<http://creativecommons.org/licenses/by-nc-nd/2.5/>

LOUGHBOROUGH
UNIVERSITY OF TECHNOLOGY
LIBRARY

AUTHOR/FILING TITLE

SAMAN, M.Y.

ACCESSION/COPY NO.

040116902

VOL. NO.

CLASS MARK

Loan copy

0401169022



AUTOMATIC PARALLELIZATION OF PROGRAMS

b y

MD YAZID MOHD SAMAN

A Doctoral Thesis

**Submitted in partial fulfilment of the requirements
for the award of
Doctor of Philosophy
of the Loughborough University of Technology**

July 1993

© by Md Yazid Mohd Saman, 1993

Loughborough University of Technology Library	
Date	Nov 95
Class	
Acc. No.	040116902

q 3090535



الحمد لله رب العالمين
والصلاة والسلام على خاتم الأنبياء والمرسلين

*In the Name of Allah, Most Gracious, Most Merciful
Praise to Allah, Lord of the Universe.
May Peace and Prayers Be upon His
Final Prophet and Messenger.*

CERTIFICATE OF ORIGINALITY

This is to certify that I am responsible for the work submitted in this thesis, that the original work is my own except as specified in acknowledgements or in footnotes, and that neither the thesis nor the original work contained therein has been submitted to this or any other institution for a higher degree.

Md Yazid Mohd Saman

ACKNOWLEDGEMENTS

I would like to express my thanks and gratitude to my Director of Research, Professor D. J. Evans for his invaluable help, guidance, supervision and support throughout my research.

I also would like to thank the staff in the Department of Computer Studies, Loughborough University of Technology for the assistance they gave, Associate Professor Dr. Baharom Sanugi for reading a draft of my thesis and all of my friends for their encouragement and advice.

Finally, I would like to thank my father and mother for their love and encouragement, my wife Meriam (who thoroughly read a draft of this thesis) and our children, Nabilah, Aaina and Imran, for their constant love, devotion and patience. May Allah reward them all.

The financial support to do this research was provided by Universiti Pertanian Malaysia.

ABSTRACT

Parallelizing Compilers have emerged to be a useful tool in the development of parallel programs. Most programmers are used to writing sequential programs. With the advent of parallel machines, the task of writing parallel programs has become both time-consuming and difficult. One way to help programmers to write parallel programs is to have a software tool that will parallelize sequential programs. This tool should be able to recognize any parts of a sequential program that can be parallelized. Then, it is transformed automatically by the tool into its parallel version.

The main focus of this thesis is to investigate the methods for automatic parallelization of sequential programs. It includes a study on the data dependence analysis that can be performed on sequential programs. This is one of the most important parts in a parallelizing compiler. The dependence analysis described here is based on the Bernstein Sets (BSs) [Bernstein (1966)] and the dependence tests, called the Bernstein Tests (BTs), developed by Williams (1978). A software tool is developed to determine the BSs and to implement the BTs. The tool also determines program granularity sizes for parallel executions by scheduling the parallel parts of programs for shared-memory parallel computers. This thesis also studies the parallelization of loops, another major topic in a parallelizing compiler. The BTs developed by Williams are extended to handle these loops. These tests are called the Bernstein Loop Tests (BLTs). Apart from these tests, the thesis also discusses the loop transformation techniques that can be carried out, based on information provided by the BSs and the BLTs. The thesis also studies the Inter-procedural Analysis (IPA) which determines information on variables that can be propagated back when a procedure is called. Finally, a technique to verify the correctness of parallel programs is presented. The whole discussion presented in this thesis is based on the Bernstein Sets.

TABLE OF CONTENTS

CHAPTER 1: INTRODUCTION TO PARALLELIZATION OF PROGRAMS

1.1. INTRODUCTION.....	2
1.2 THE ROLE OF A PARALLELIZING COMPILER.....	3
1.3 ORGANIZATION OF THE THESIS.....	7

CHAPTER 2: FUNDAMENTAL CONCEPTS OF PARALLEL PROGRAMMING

2.1 INTRODUCTION.....	10
2.2 PARALLEL COMPUTER ARCHITECTURES.....	11
2.2.1 The Shared-Memory Multi-processor Systems.....	12
2.2.2 The Message-Passing Multi-processor Systems.....	13
2.2.3 The Array Processors.....	14
2.2.4 The Pipelined Processors.....	15
2.2.5 Data Flow Computers (Data Driven).....	17
2.3 ACHIEVING PARALLELISM.....	18
2.4 ELEMENTS OF PARALLEL LANGUAGES.....	20
2.5 PARALLEL PROGRAMMING ON A SEQUENT BALANCE SYSTEM	24
2.5.1 Process Synchronization.....	24
2.6 PROBLEMS WHEN WRITING PARALLEL PROGRAMS.....	26
2.7 SUMMARY	27

CHAPTER 3: DATA DEPENDENCE ANALYSIS

3.1 INTRODUCTION.....	29
3.2 THE BERNSTEIN METHOD.....	32
3.3 THE GRAPH-BASED METHOD	34
3.4 THE DIOPHANTINE ANALYSIS	41
3.5 CONTROL DEPENDENCES	44
3.6 SUMMARY	44

CHAPTER 4: A TOOL FOR AUTOMATIC DETERMINATION OF PARALLELISM

4.1 INTRODUCTION.....	47
4.2 DETECTION OF IMPLICIT PARALLELISM	48
4.3 SCHEDULING OF CONCURRENT STANZAS	49
4.4 DETERMINATION OF STANZA GRANULARITY	53
4.5 TAG: A TOOL FOR AUTOMATIC DETERMINATION OF PROGRAM GRANULARITY	56
4.5.1 The Analyser.....	58
4.5.2 The Detector	59
4.5.3 The Scheduler.....	60
4.5.4 The Merger	68
4.6 EXAMPLE OUTPUT OF TAG	70
4.7 SCHEDULING OF PARALLEL LOOPS.....	95
4.8 SUMMARY	96

CHAPTER 5: DETECTION OF LOOP PARALLELISM AND TRANSFORMATIONS

5.1 INTRODUCTION.....	99
5.2 PARALLELISM IN LOOPS.....	100
5.3 DATA REFERENCE DIRECTIONS	106
5.4 THE BERNSTEIN LOOP TESTS (BLTS)	109
5.4.1 Summary of the BLTs.....	112
5.4.2 Nested Loops.....	114
5.5 EXAMPLES OF THE APPLICATION OF THE BLTS.....	115
5.6 TRANSFORMATION OF LOOPS.....	124
5.6.1 Loop Parallelization and Vectorization.....	124
5.6.2 Definitions of fetch and store directions.....	127
5.7 TRANSFORMATION TECHNIQUES FOR SCALAR VARIABLES	128
5.8 TRANSFORMATION TECHNIQUES FOR ARRAY VARIABLES	133
5.9 RELATED ISSUES ON LOOP DEPENDENCES	146
5.10 SUMMARY	150

CHAPTER 6: INTER-PROCEDURAL ANALYSIS

6.1 INTRODUCTION.....	152
6.2 ALIASING PROBLEM.....	153
6.2.1 Parallelization of procedure calls.....	155
6.2.2 In-line Expansion.....	156
6.2.3 Inter-procedural Constant Propagation.....	158
6.2.4 Collection of Reference Information.....	159
6.3 BERNSTEIN SETS FOR PROCEDURE CALLS.....	162
6.3.1 Simple-Call Algorithm.....	162
6.3.2 Example of handling call-by-value parameters.....	164
6.3.3. Example of handling call-by-reference parameters.....	166
6.3.4 Handling array variables.....	167
6.3.5 Limitations of the Simple-Call Algorithm.....	167
6.4 AN IMPROVED ALGORITHM.....	170
6.5 A GENERAL SOLUTION.....	173
6.6 EXAMPLE OF SOLUTIONS.....	181
6.7 PROCEDURE CALLS IN LOOPS.....	186
6.7.1 Handling array variables.....	186
6.8 SUMMARY.....	190

CHAPTER 7: VERIFICATION OF PARALLEL PROGRAMS

7.1 INTRODUCTION.....	192
7.2 METHODS FOR PROVING PROGRAMS.....	193
7.3 SYMBOLIC EXECUTION.....	194
7.4 OTHER RELATED WORK.....	199
7.4.1 The stanza approach.....	199
7.4.2 The axiomatic approach.....	200
7.4.3 Formal methods.....	201
7.5 VERIFYING PARALLEL STANZAS.....	201
7.5.1 The BT Assertion.....	202
7.5.2 Critical Sections.....	203
7.6 VERIFYING PARALLEL LOOPS.....	207
7.7 SUMMARY.....	212

CHAPTER 8: SUMMARY AND CONCLUSIONS

8.1 INTRODUCTION.....	215
8.2 DEPENDENCE ANALYSIS AND SCHEDULING OF STANZAS.....	216
8.3 LOOP DEPENDENCES AND TRANSFORMATIONS.....	218
8.4 INTER-PROCEDURAL ANALYSIS	218
8.5 VERIFICATION OF PARALLEL PROGRAMS.....	219
8.6 FUTURE RESEARCH.....	220
REFERENCES.....	223
APPENDIX A: TAG MAIN ROUTINE.....	244
APPENDIX B: THE SCHEDULER ROUTINE.....	246
APPENDIX C: THE MERGER ROUTINE.....	256
APPENDIX D: THE BERNSTEIN LOOP TESTS.....	264

CHAPTER 1

INTRODUCTION TO PARALLELIZATION OF PROGRAMS

1.1. INTRODUCTION

Today, computers are a common phenomenon in our daily life. They can be found everywhere such as in offices, in schools and at home. They are widely used as a tool for solving numerical and non-numerical problems. These computers are simple to use and very fast in computation. With the advancement of sophisticated micro chip design, they are getting smaller and cheaper. In the last decade, their architectures have shifted widely from what has been known as 'single-processor computers' to 'multi-processor computers'. These multi-processor computers are also known as 'Parallel Computers' or 'Supercomputers'. They are capable of producing better and faster performance as more than one processor can work in parallel to solve different parts of a single problem [Almasi and Gottlieb (1989), Hwang and Briggs (1984), Zima and Chapman (1990)].

Most of the currently available supercomputers are designed to be used for solving very large scientific and engineering problems [Almasi and Gottlieb (1989), Hwang and Briggs (1984), Zima and Chapman (1990)]. With the introduction of inexpensive but highly efficient commercial multi-processor computers such as the Sequent Balance [Osterhaug (1987), Thakkar et al. (1988)], a wide range of applications ~~is~~ being developed and solved on them. One such application which needs enormous computational resources is the weather forecasting problem where massive data are gathered and have to be processed in time before the weather arrives. Another example of the usefulness of a parallel computer is in its application in the research on structural biology where the structure of DNA can be determined efficiently. The high performance that these machines produce has led to more people from different fields to benefit from them.

Computers have to be programmed in order to fully utilize them. Programs are instructions written either in the low-level languages (such as machine and assembly languages) or the high-level languages (such as FORTRAN, C, Pascal or PL1) [Almasi and Gottlieb (1989)]. Apart from these programming languages, software packages such as the fourth-generation languages are

also available for users to develop application software to solve their problems [Meehan (1990)]. The task of programming has now become more difficult and expensive because of the complexity of the problems and the higher labour cost. For the sequential programs, however, they can still be run on the parallel computers but the time taken to execute them will be the same as running on single-processor computers.

1.2 THE ROLE OF A PARALLELIZING COMPILER

To achieve the high performance of parallel computers, programmers have to write parallel programs. One of the main tasks is to identify parts of the programs that are to be executed in parallel. These parts are then expressed in certain parallel language constructs for execution. This process is an essential part of parallel programming. It is important to execute as many independent operations concurrently as possible on different processors of the parallel computers. This is not as easy as writing the equivalent sequential programs. Identifying and keeping track of these independent operations is very time-consuming and error prone especially in big programs. This is particularly crucial for those parts which are less apparent to programmers. If this is not performed to great extent, it will hinder parallelism and will cause slower computation due to the sequential execution of the program.

One way to develop a parallel program is to write its sequential version in the initial stage. This program is then transformed into its parallel form by a sophisticated software tool such as the **parallelizing compiler**. It should be able to detect, either automatically or with users' help, any form of parallelism that exists and to carry out the transformation process. This transformation tool is also very useful for any existing sequential software. Hence, a good and efficient parallelizing compiler is an important and essential tool to aid the general programmers in writing parallel programs.

The parallelizing compiler, or sometimes referred to as a **supercompiler**, is a software system that compiles programs targeted for execution on a parallel architecture system [Leung (1990), Padua et al. (1980), Padua and Wolfe (1986), Wolfe (1989a, 1989b), Zima and Chapman (1990)]. Figure 1.1 shows an example of the design of a parallelizing software system adapted from Zima and Chapman (1990). It performs either **vectorization** of program to generate vector codes or **parallelization** to produce codes for a multi-processor system. Examples of such systems are the Parafrase [Leasure (1985), Polychronopoulos et al. (1990)], PFC [Allen and Kennedy (1984b)], R^n [(Cooper et al. (1986)], PTOOL [Allen et al. (1986)], Faust [Guarna et al. (1989)], Superb [Zima et al. (1988)], ToolPack [Cowell (1988), Cowell and Thompson (1990)], KAP [Davies et al. (1986), Huson et al. (1986), Macke et al. (1986)], Start/Pat [Appelbe and Smith (1989)] and PTRAN [Cytron et al. (1990)].

As mentioned earlier, one of the principle tasks performed by a supercompiler is to detect any parallelism in a source program. This is carried out in the **Data Dependence Analysis (DDA)** [Allen and Kennedy (1987), Allen et al. (1987), Banerjee (1988), Burke et al. (1988), Callahan et al. (1987), Li (1989), Li et al. (1989), Li and Yew (1990), Williams (1978), Wolfe (1989a, 1989b), Wolfe and Banerjee (1987)]. It involves a detailed examination of the program on how its variables are being referenced. The result of the analysis will appear in the form of **dependence relations** between parts of the program. Most of the work that has been carried out on the DDA represents the dependence relations in the form of a **dependence graph**. In this thesis, however, the information gathered are saved in the **Bernstein Sets (BSs)** [Bernstein (1966), Williams (1978)]. The DDA is performed on the BSs to determine the parallelizable parts of a program.

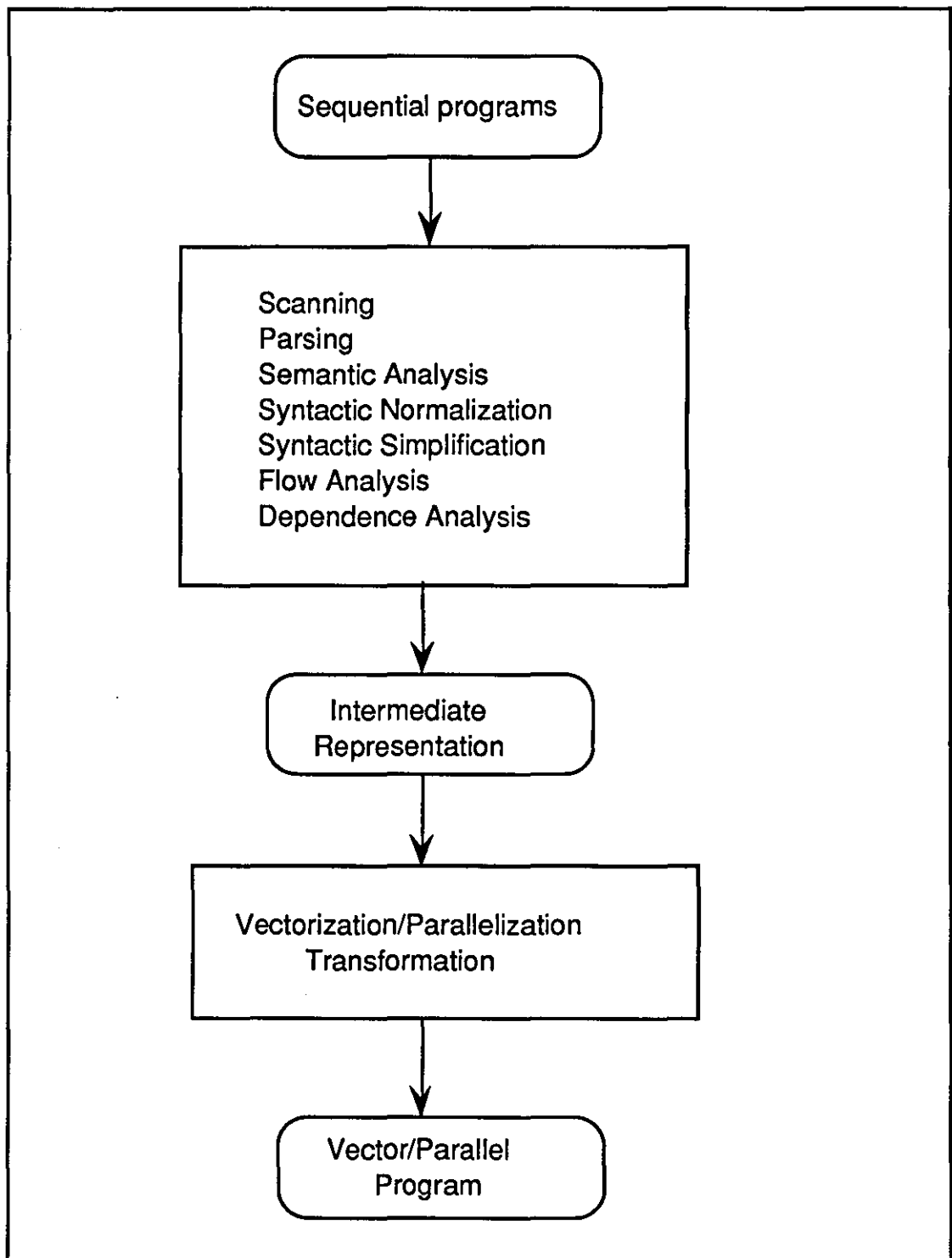


Figure 1.1: The structure of a Parallelizing Compiler

For a sequential program, the parts that offer the best opportunities amenable to parallelism are the loops [Allen and Kennedy (1984a, 1987), D'Hollander (1989), Harrison and Chow (1991), Jackson (1985), Midkiff and Padua (1986, 1987), Mohd-Saman and Evans (1993), Padua and Wolfe (1986), Saltz et al. (1989, 1991), Wolf and Lam (1991), Wolfe (1986, 1988, 1990), Zima and Chapman (1990)]. Loop iterations can be executed on different processors in a concurrent manner if they are independent of each other.

Array references in the loop body are the main cause of the data dependences. One iteration may be modifying an element which is being referenced by another. Hence, the usage of arrays in loops have been a major target to be analysed for program parallelization. Once the dependence relations have been established by the DDA, the loops pass through a transformation process. It will perform two tasks: removal of those data dependences that inhibit parallelism and generation of parallel (or vector) codes. Some of the well-known loop transformation techniques are loop distribution, statement reordering and loop interchanging [Lewis and El-Rewini (1992), Zima and Chapman (1990)].

One of the programming paradigms that has been proposed in program writing is the structured or procedural programming [Kruse (1984), Welsh and McKeag (1980)]. This technique introduces procedures that can be called from either the procedures themselves or other parts of a program. This has become one major problem encountered in the DDA. A call may or may not be modifying a global variable, depending on the way the parameters are passed. This problem needs another detailed analysis called the **Inter-Procedural Analysis (IPA)** [Barth (1978), Burke and Cytron (1986), Callahan et al. (1986), Li (1989), Schouten (1990), Triolet et al. (1986)]. It involves collecting information on the usage of variables in a procedure when a call is found.

1.3 ORGANIZATION OF THE THESIS

The research presented in this thesis covers the various aspects of a supercompiler. Specifically, its main focus is to study the methods in the automatic determination of implicit parallelism that exists in sequential programs, based on the Bernstein Sets (BSs) [Bernstein (1966)] and the sets of dependence tests developed by Williams (1978). It includes a detailed study on the applications of the method in the design of a software tool to perform the extraction of parallelism and the determination of task granularity. A method to detect parallelism in loops, called the Bernstein Loop Tests (BLTs) [Mohd-Saman and Evans (1993)] and the analysis on effects of procedure calls on parallelism, through IPA, are also thoroughly studied. Another topic that is also addressed is the problem of verifying the correctness of parallel programs.

The thesis is organized as follows. Chapter 2 gives a brief discussion on the concepts of parallel computer architectures and parallel programming. In Chapter 3, a survey on the methods for the detection of parallelism is presented. This mainly focuses on the various strategies for the Data Dependence Analysis (DDA). Chapter 4 gives a detailed design on a software tool called TAG that can determine any parallelism in sequential programs automatically. It also includes discussions on the scheduling of concurrent parts of a program and the determination of their granularity. Chapter 5 presents a set of tests (i.e., the BLTs) to detect parallelism in loops and techniques on how the loops can be transformed into parallel forms, based on the results of the BLTs. The effects of procedure calls on the detection of parallelism and how IPA is performed, based on the BSs, are given in Chapter 6. Chapter 7 proposes methods to verify the correctness of parallel programs using the Symbolic Execution method, combined with the dependence tests developed in this thesis. Finally, Chapter 8 summarises and gives conclusions on the topics discussed throughout the thesis.

Before proceeding, it should be noted that the following groups of terms take the same meaning in their own group and may be used interchangeably throughout the thesis.

- a. concurrent, parallel and contemporary
- b. parallel machine (system), supercomputer and multi-processor system
- c. supercompiler, parallelizing compiler and optimizing compiler
- d. tasks, process and stanzas (defined in chapter 3)

CHAPTER 2

FUNDAMENTAL CONCEPTS OF PARALLEL PROCESSING

2.1 INTRODUCTION

The notion of parallelism exists in our every day life. An example is in a bank which has more than one customer counter. It lets the different parts of the system be serviced concurrently in a much faster and efficient way. The desire to achieve results in the same manner for a large problem on a computer, has led to the development and advancement of supercomputers and their related software tools. In Hwang (1989), supercomputers are defined as the fastest computers at any point of time. If compared to today's computer mainframes, they are many times faster in effective speed. This is mainly achieved by the parallel architectures that form the backbone of the systems.

The von Neumann architecture model was the first accepted concept in the development of memory-stored electronic computers. With the introduction of the world's first electronic computer, ENIAC in 1946, computers have passed through several phases of development. ILLIAC IV with 64 processing elements was the first operational supercomputer built [Karplus (1989), Kuck (1968)]. The popular Cray-1 supercomputer which was capable of 130 Mflops (million of floating point operations per second) marked the beginning of a commercial use. The present day outstanding performance of the available parallel machines is still subject to further research for improvement. Now the availability of the supercomputers and their power has made them more accessible to a wider range of users. However, the computer manufacturers have to provide software tools in order to help users to fully utilize the machines. Parallelizing compilers are one such tool [Allen (1988), Appelbe and Smith (1989), Callahan et al. (1987), Cowell and Thompson (1990), Hiranandani et al. (1992), Kuck et al. (1984), Polychronopoulos et al. (1990), Wolfe (1989b)].

In this chapter, fundamental concepts in parallel processing are briefly presented. These include the general parallel computer architectures in Section 2.2 and the concepts of parallel programming in Section 2.3 to Section 2.5.

2.2 PARALLEL COMPUTER ARCHITECTURES

The architectures for parallel computers can be classified as one of the following configurations [Almasi and Gottlieb (1989), Hwang and Briggs (1984), Hwang (1989), Karplus (1989), Perrot (1987), Williams (1990)].

- a. Multi-processor Systems
- b. Array Processor Systems
- c. Pipelined Computers
- d. Data Flow Computers

The first type, the Multi-processor systems are computers with a set of independent and autonomous processors. They can be divided into two categories, the Shared-Memory Multi-processor Computers and the Message-Passing Multi-processor Computers. For the second configuration, the Array Computers, arrays of processing elements receive the same instruction from one main control. In the third configuration, the Pipelined Computers, their operations are processed successively by separate hardware units. The last architecture, the Data Flow Machines are designed for a fully maximum parallel computation.

The Flynn's taxonomy is one way to categorize the structures of computer systems [Almasi and Gottlieb (1989), Flynn (1972), Williams (1990)]. They are as follows.

- (a) SISD - Single instruction stream, single data stream. This is the von Neumann uniprocessor computer model.
- (b) SIMD - Single instruction stream, multiple data stream. The Array and Pipelined computers are examples of this type.
- (c) MISD - Multiple instruction stream, single data stream. No known machine has been built for this type.

- (d) MIMD - Multiple instruction stream, multiple data stream.
This includes the Multi-processor systems.

2.2.1 The Shared-Memory Multi-processor Systems

The Shared-Memory Multi-processor Systems (SMSs), sometimes called the Tightly-Coupled System, has a set of processing elements (PEs) and a pool of memory available to all processors through which they communicate [Almasi and Gottlieb (1989), Hwang and Briggs (1984), Perrot (1987)]. This type of architecture is illustrated in figure 2.1. Examples of SMSs include the Sequent Symmetry and Balance, Alliant FX/8, and the Encore Multimax. Due to the problem imposed by the communication through the shared memory, they usually have a relatively small number of PEs. For example, the Sequent Balance 8000 and the Encore Multimax can only have at most 12 and 20 processors respectively.

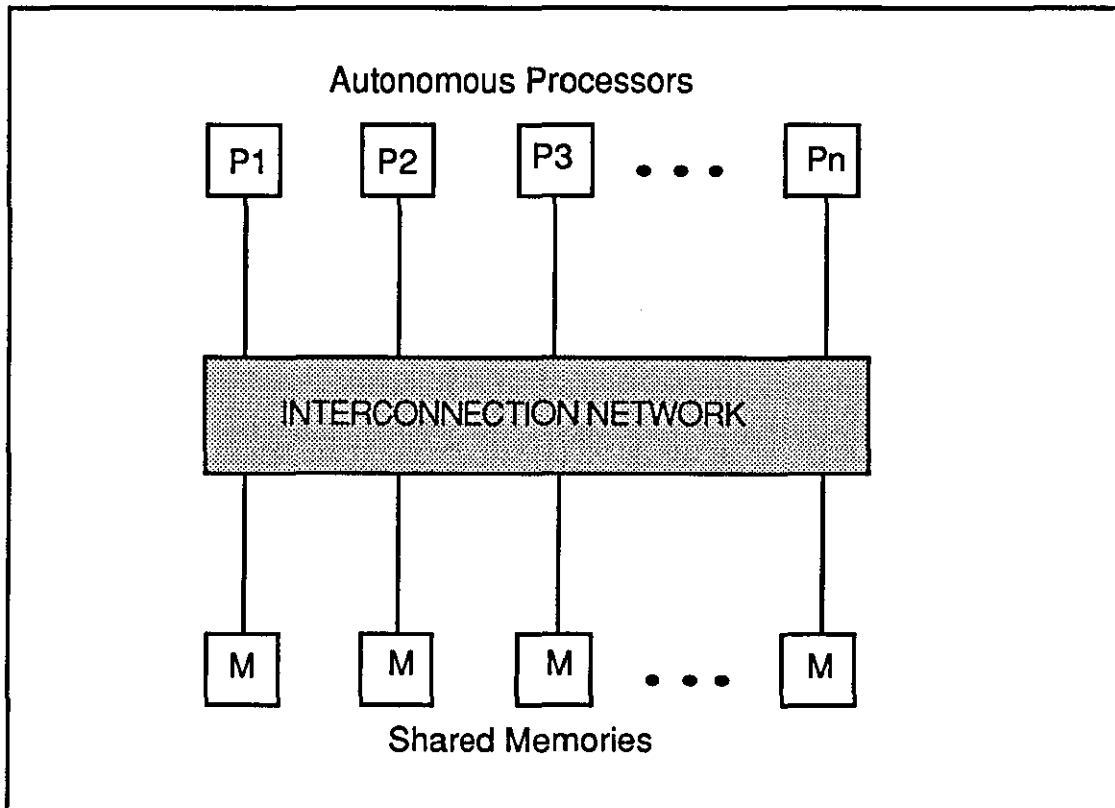


Figure 2.1: Configuration of the Shared-Memory Multi-processor System

2.2.2 The Message-Passing Multi-processor Systems

In the Message-Passing Multi-processor Systems (MPSs), each autonomous processor (PE) in the systems has its own local memory, as shown in figure 2.2. These systems are also known as the Local Memory Systems, Loosely-Coupled Systems or the Distributed-Memory Systems. Communications among the processors are performed through a message transfer system. The Intel Hypercubes (iPSC/1, iPSC/2 and iPSC/860 models with 128 PEs), BBN Butterfly (128 PEs), Ametek System 14 (256 PEs), NCUBE Hypercube (1024 PEs) and the Transputer systems (T414, T212 and T800 models) are examples of the MPSs [Almasi and Gottlieb (1989), Freeman and Phillips (1992), Hwang and Briggs (1984)]. In general, these systems have a much greater number of PEs than that of the SMSs.

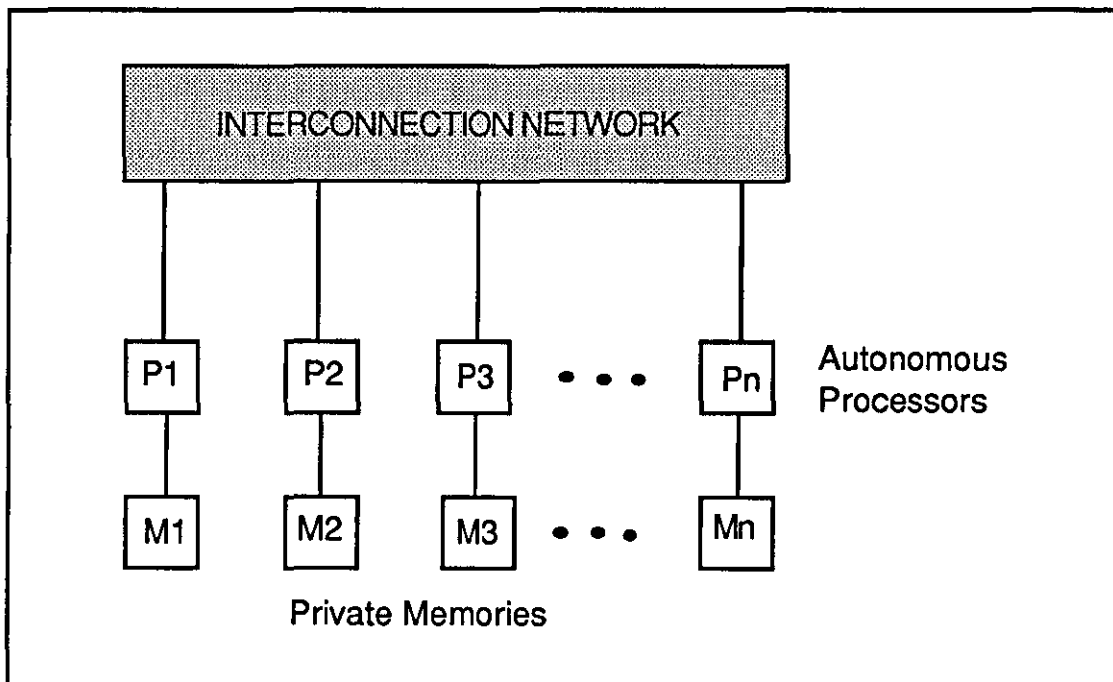


Figure 2.2: Configuration of the Message-Passing Multi-processor System

2.2.3 The Array Processors

The Array Processors are parallel computers which have a number of PEs where each one of them obeys the same instruction issued by a single control unit but they operate on different local data [Almasi and Gottlieb (1989), Hwang and Briggs (1984), Perrot (1987)]. Figure 2.3 illustrates this configuration. Examples of these machines are the Active Memory Technology Distributed Array Processor, the Goodyear Aerospace Corporation Massively Parallel Processor and Connection Machines of the Thinking Machines.

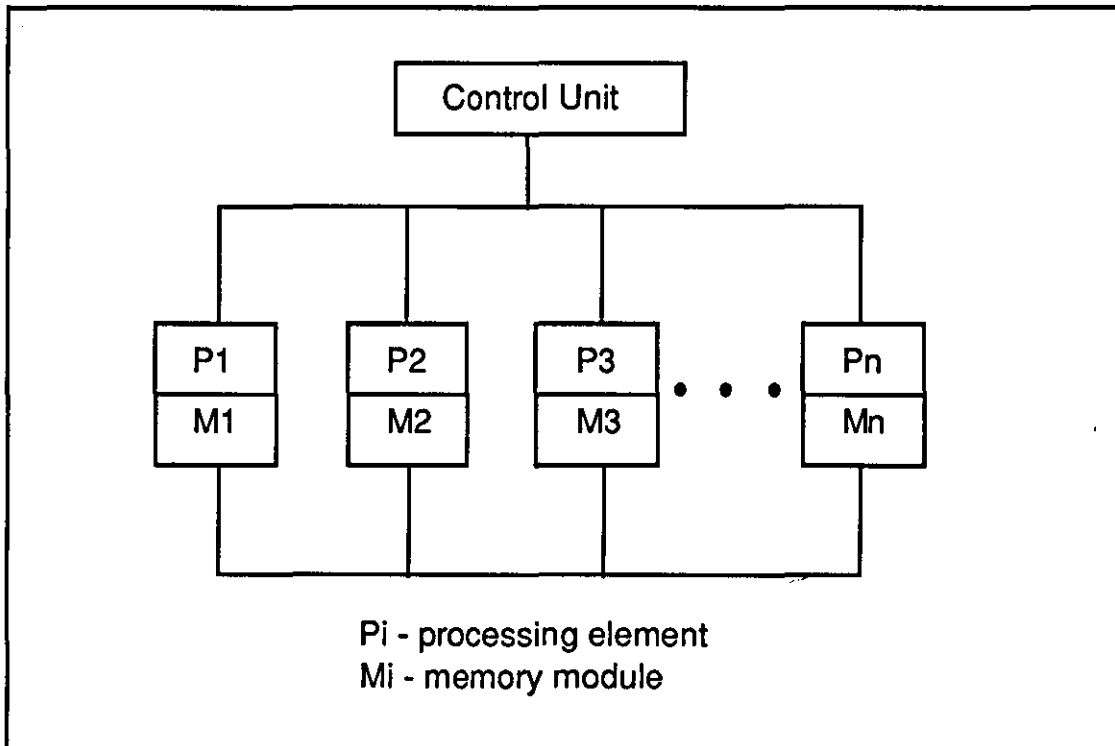


Figure 2.3: An Array Processor Machine

2.2.4 The Pipelined Processors

The Pipelined Processors are SIMD computers that have an architecture in which a series of operations are streamed into their multiple processors at the same time and executed in an overlapped manner [Almasi and Gottlieb (1989), Hwang and Briggs (1984), Perrot (1987), Williams (1990)]. They are also called Pipelined Vector Processors that can handle vector instructions with vector operands. The Cray family supercomputers and the Fujitsu VP2600/10 supercomputer are examples of this type of architecture. Figure 2.4 shows the executions of four series of operations by four processors in an overlapped manner. The Instruction Pipelining is one classification of a Pipelined Processing and it is shown in figure 2.5. Other classifications include the Arithmetic Pipelining and the Macro Pipelining. A systolic array is one type of a pipelined computer with more than one dimension.

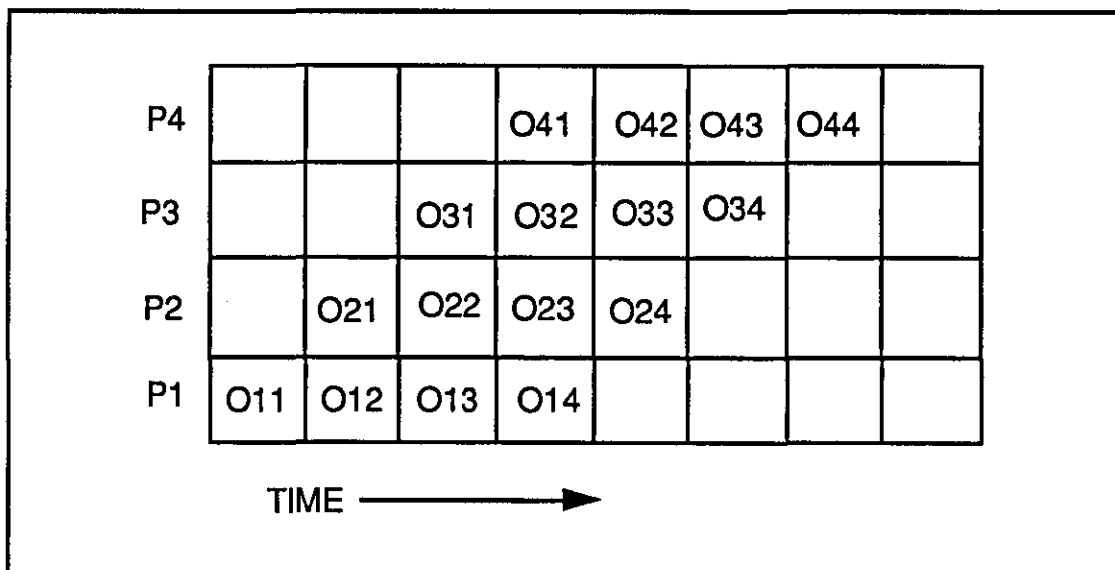


Figure 2.4: Pipelined operations

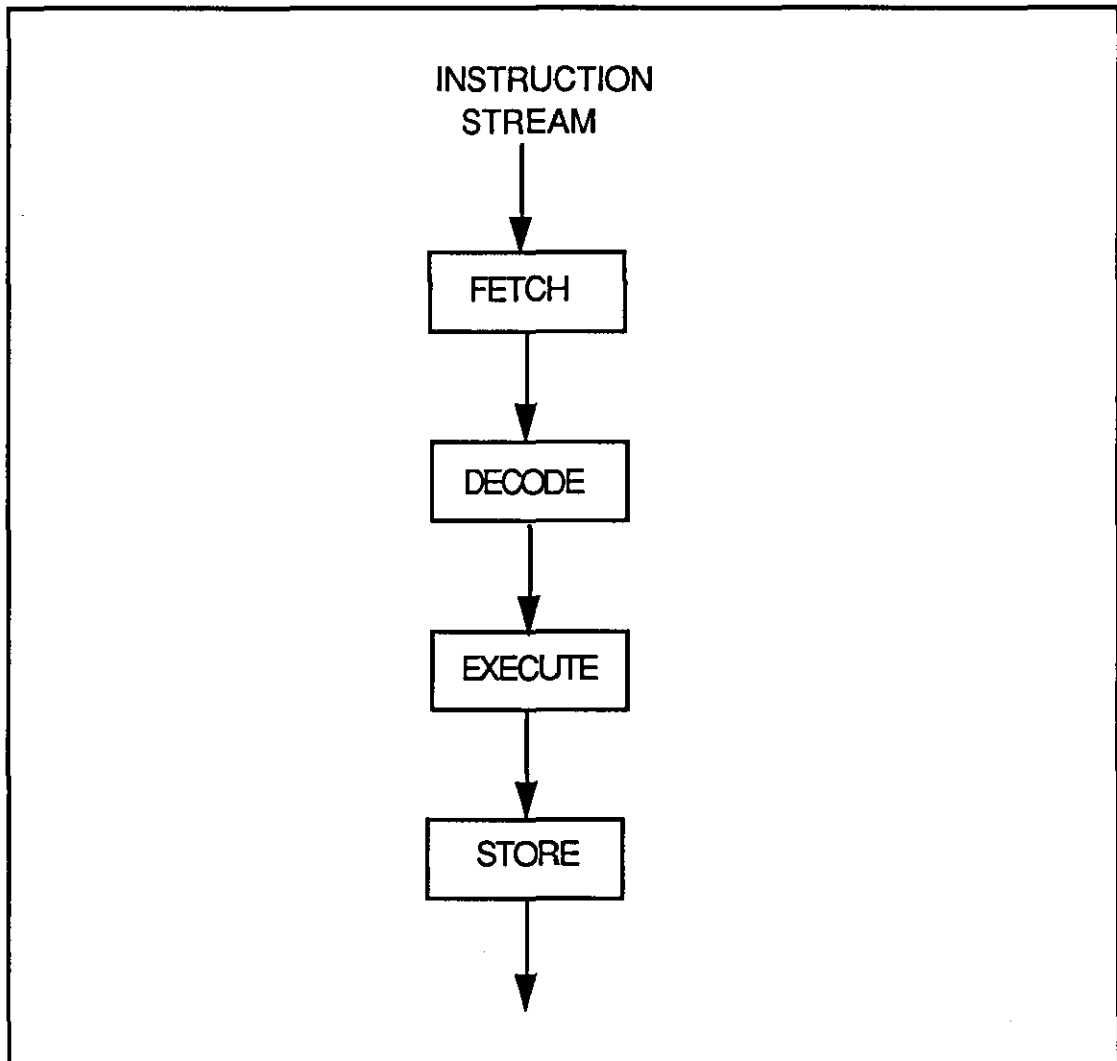


Figure 2.5: Four stages of Instruction Pipelining that are overlapped

2.2.5 Data Flow Computers (Data Driven)

Conventionally, the computers adopt the von Neumann model of architecture which is based on the stored instructions controlled by a program counter. This leads to the sequential execution of a program. In a Data Flow computer, however, a different execution approach is followed to achieve maximum parallelism [Almasi and Gottlieb (1989), Dennis (1980), Hwang and Briggs (1984), Perrot (1987)]. Basically, an execution of an instruction proceeds as soon as its operands are available. Therefore, the flow of computation is not controlled by a program counter but by the availability of data in the program. However, there is a precedence constraint for each operation imposed by the algorithm used. This is to ensure the correctness of the result produced. Figure 2.6 shows an example of a Data Flow execution. An example of this machine is the Manchester Data Flow Machine [Gurd et al. (1985)].

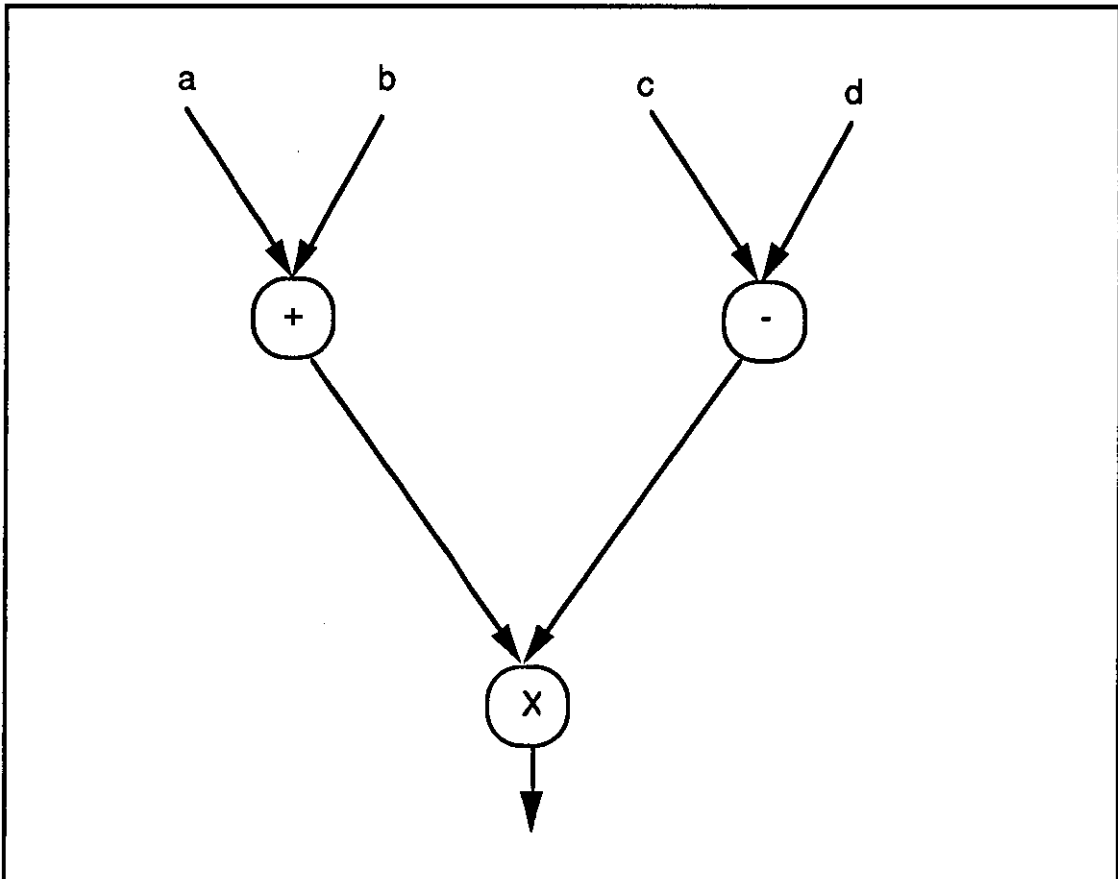


Figure 2.6: Data Flow execution of $(a+b) \times (c-d)$

2.3 ACHIEVING PARALLELISM

Parallelism in a computer system can be achieved through two ways, *MULTIPROGRAMMING (or Timesharing)* and *MULTITASKING* [Osterhaug (1987)]. Multiprogramming allows several jobs (or programs) to be processed at the same time and this will give the maximum throughput of the computer. This is common on most computers nowadays which allow more than one user to log on to the machines, although they have one processor [Evans (1990), Hansen (1973), Silberschatz (1991)]. In the other situation, Multitasking, a program is broken up into several processes (tasks or parts) and each one of these processes will be handled by the different available processors. This will give the shortest processing time to solve a problem.

The Operating System in a computer, such as UNIX, is able to handle Multiprogramming by allocating jobs in a ready queue to the CPU as soon as it is free. The jobs are given time slices to be processed [Bach (1986), Hansen (1973), Silberschatz (1991)]. If the processing of a job exceeds its time slice, it is pre-empted and it will join the ready queue again and this allows other jobs to be executed. Hence, from the user point of view, parallelism through multiprogramming is achieved. Figure 2.7 illustrates this multiprogramming environment.

DYNIX, an operating system for the Sequent Parallel Machine, allows Multitasking as well as Multiprogramming [Osterhaug (1987)]. It has library commands to create processes and to synchronize them such as the FORK, JOIN and LOCK instructions. Thus, it is left to the programmer to write a parallel program specifying which tasks are to be executed in parallel. An illustration of fork and join operations is shown in figure 2.8.

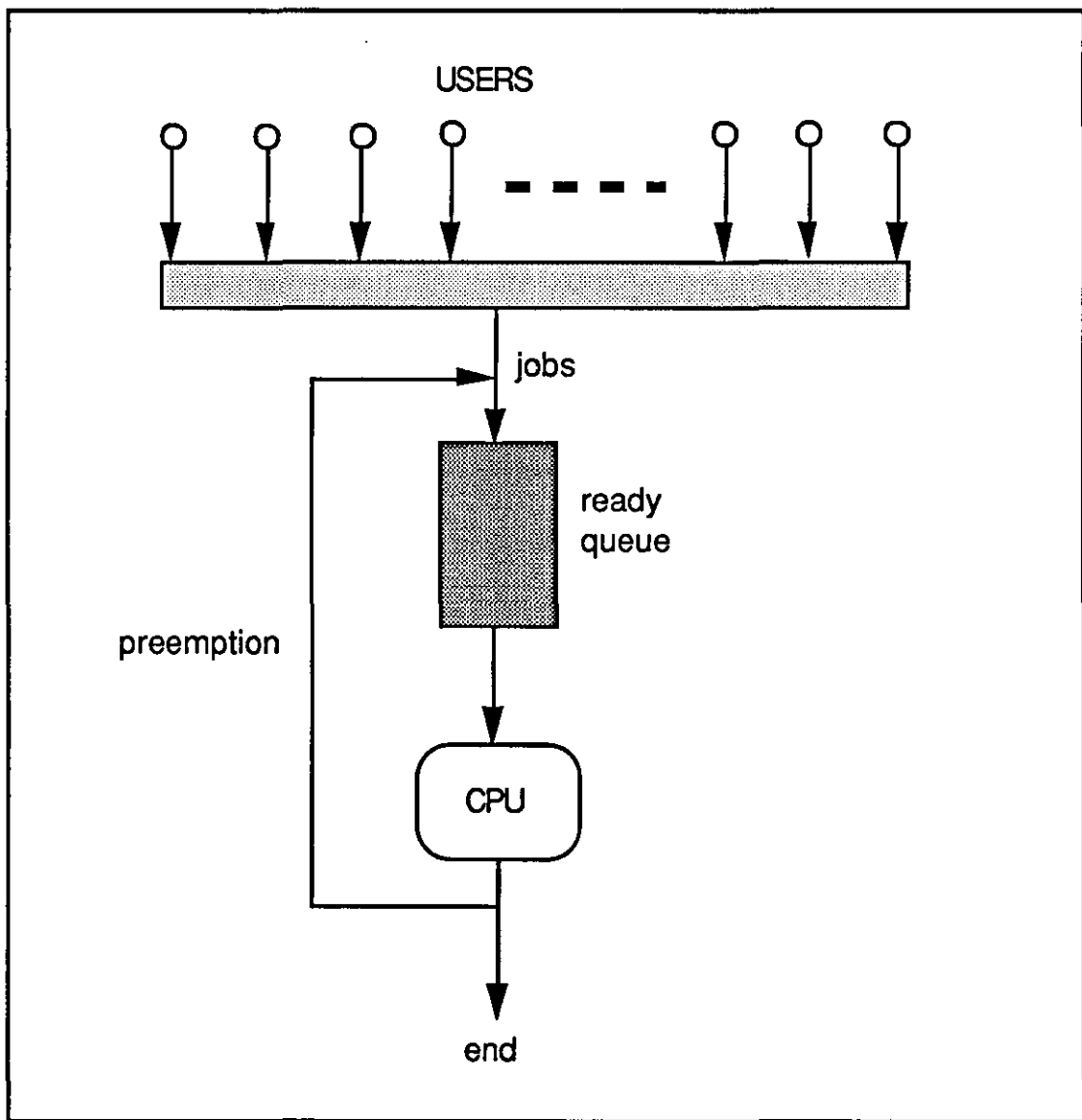


Figure 2.7: Multiprogramming model for a uniprocessor computer

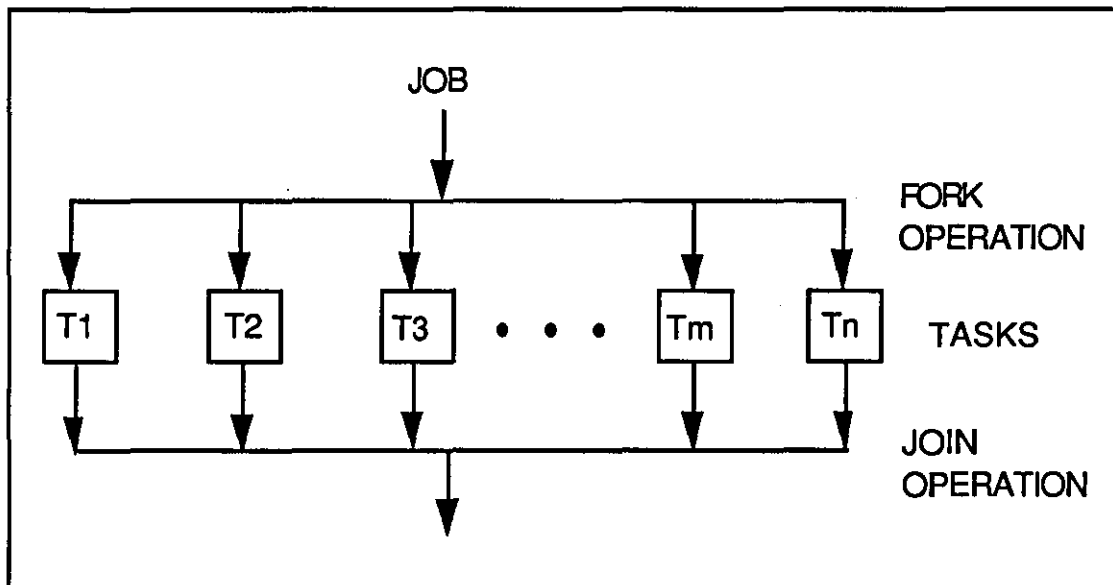


Figure 2.8: Multitasking Environment

2.4 ELEMENTS OF PARALLEL LANGUAGES

To take advantage of the capabilities provided by the parallel machines, programs have to be written and coded in a specific way in a parallel language. To develop a program, the first step is to choose the most appropriate algorithm. Then the data structure is selected followed by coding the algorithm. Examples of the languages available for parallel programming include ADA [Gehani (1984)], OCCAM [Pountain and May (1987)], Concurrent Pascal [Hansen (1975)], CSP [Hoare (1978)], Multilisp [Halstead (1985)], COOL [Chandra et al. (1990)], Concurrent C++ [Gehani and Roome (1988)], Presto [Bershad et al. (1988)], SISAL [Garsden and Wendelborn (1990)] and V-Pascal [Tsuda and Kunieda (1992)].

Writing the codes for a parallel program can be categorized into three manners. First, as coarse grain where a program is organized into procedures (as processes), second, as medium grain where the parallelism is organized at loop level (loop iterations as processes) and third, as fine grain where parallelism is expressed at basic block level, statement level and expressions [Polychronopoulos (1988)].

Languages with parallel constructs allow programmers to manually insert directives where the parallelism can be achieved [Almasi and Gottlieb (1989), Andrew and Schneider (1983), Freeman and Phillips (1992), Hwang and Briggs (1984), Perrot (1987)]. These directives usually define:

- a. a set of subtasks to be executed in parallel
- b. the start and stop of their execution
- c. the coordinating and interaction while the parallel tasks are executing.

Some examples of the parallel constructs that are usually found in languages include the following [Almasi and Gottlieb (1989), Hwang and Briggs (1984)]:

- a. parbegin/parend (or cobegin/coend)
- b. fork/join
- c. doall (or forall, pardo, doaccross)
- d. synchronization primitives:
 - test-and-set
 - semaphores
 - barriers

Figures 2.9 and 2.10 show an example of a parallel program with parbegin-parend constructs and its flow control respectively. These constructs specify the parts of the program that can be executed in parallel. Delimiters begin-end indicate the normal sequential statements.

For a Shared-Memory Multi-processor System, a parallel language must be able to create (spawn) new processes, destroy processes and identify processes. It should also be able to differentiate variables which are globally accessible to all processors as well as those local to a given processor. Apart from these operations,

there should also be a way to synchronize processes through the use of the shared memory.

The requirements for a parallel language intended for a Message-Passing Multi-processor System include similar capabilities of creating, destroying and identifying processes as in the Shared-Memory Systems. However, it should also have the ability to specify explicitly the methods of communication between processes through send and receive commands either on a one-to-one basis or in a broadcast style.

```
statement-A
PARBEGIN
    statement-B
    BEGIN
        statement-C
        PARBEGIN
            statement-D
            statement-E
        PAREND
        statement-F
    END
    statement-G
PAREND
statement-H
```

Figure 2.9: A program with PARBEGIN-PAREND

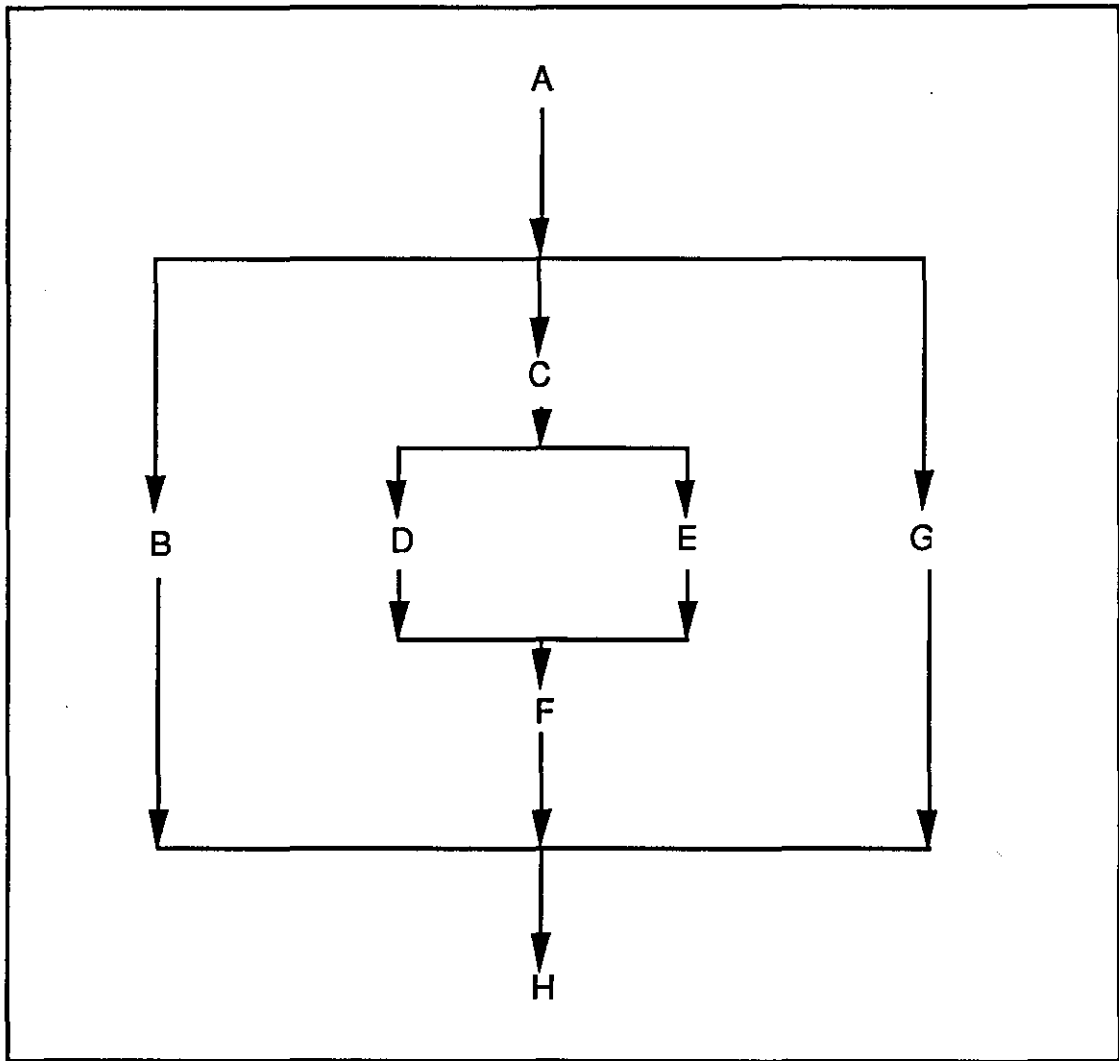


Figure 2.10: The flow control of the program in fig. 2.9

2.5 PARALLEL PROGRAMMING ON A SEQUENT BALANCE SYSTEM

For a Shared-Memory Multi-processor Computer such as the Sequent Balance 8000, two types of multitasking programming methods are available for the users to implement their programs. They are Data Partitioning and Function Partitioning [Osterhaug (1987)]

Data Partitioning allows the users to create multiple identical tasks (or processes). It then assigns a portion of the data to each process. This is also called homogeneous multitasking since it involves identical tasks execution in parallel. This kind of programming is well suited to executing loop iterations in parallel such as the matrix multiplication and the Fourier Transformation. Figure 2.11 shows a segment of a C program performing the Data Partitioning strategy with static scheduling.

The other technique, *Function Partitioning*, allows the users to create multiple unique processes running in parallel, each accessing a shared data set. This method, also called heterogeneous multitasking, is suitable for applications with tasks of different operations but on the same set of data, such as the flight simulation and the program compilation.

2.5.1 Process Synchronization

When many processes which are running in parallel, try to modify a shared variable, they have to be synchronized. This can be controlled by shared data structures called *semaphores*. The simplest of all semaphores is called a *lock* that allows a user to create a critical code region that can be accessed by only one process. Other forms of semaphores include the *ordering semaphore* and the *counting/queueing semaphore*.

```

#include <stdio.h>
#include <parallel/microtask.h> /* microtasking routine header */
#include <parallel/parallel.h> /* standard parallel library header */
#define SIZE 10                /* size of matrices */

/* Global shared memory data */

shared float a[SIZE][SIZE]; /* first array */
shared float b[SIZE][SIZE]; /* second array */
shared float c[SIZE][SIZE]; /* result array */

main()
{
    void init_matrix(), m_fork(), m_kill_procs(), matmul(), print_mats();

    init_matrix(a,b); /* initialise data */
    m_set_procs(nprocs); /* set no. of processes */
    m_fork(matmul,a,b,c); /* execute parallel loop */
    m_kill_procs(); /* kill child processes */
    print_mats(a,b,c); /* print results */
}

/* matrix multiply function */
void matmul(a,b,c)
float a[][SIZE], b[][SIZE], c[][SIZE];
{
    int i,j,k,nprocs;

    nprocs = m_get_numprocs(); /* get no. of processes */
    for (i=m_get_myid(); i<SIZE; i+=nprocs)
    {
        for (j=0; j<SIZE; j++)
        {
            for (k=0; k<SIZE; k++)
                c[i][k] += a[i][j] * b[j][k];
        }
    }
} /* matmul */

```

Figure 2.11: A segment of matrix multiplication program performing the Data Partitioning [Osterhaug (1988)]

Apart from the semaphores, other ways to synchronize the execution of parallel processes include the use of *events* and *barriers*. Events can have two values: posted and cleared. Processes will have to wait for an event until another process posts the event before proceeding. On the other hand, a barrier is a synchronization point where a process waits at a barrier until other processes arrive before it can proceed. Detailed discussions on these synchronization statements are given in Osterhaug (1987).

2.6 PROBLEMS WHEN WRITING PARALLEL PROGRAMS

With the new parallel machines, there are several problems that are mostly encountered by users when writing programs [Evans and Williams (1978), Williams (1978)]. Among these problems are the following.

- a. **T**he programmer has to detect and express manually all possible parallelism in his programs. This includes performing the dependence analysis and inserting the synchronization statements in places where shared data are being modified. This is a very difficult task.
- b. **S**mall changes in the programs will require the programs to be reorganized and reanalysed.
- c. **E**xisting programs on which a lot of time, effort and money have been spent, need to be rewritten in order to take advantage of the capabilities offered by the parallel machines.

Another way to develop a program for a parallel machine is to write it in a sequential way. This program can then be transformed into its parallel version by a sophisticated software tool. This tool should be able to detect automatically any form of parallelism that exists and transform it into its parallel form. Extensive research work has been carried out to alleviate

the above-mentioned problems. They include the data dependence analysis and the program transformation [Allen and Kennedy (1987), Banerjee (1988), Cowell and Thompson (1990), Guarna et al. (1989), Williams (1978), Wolfe (1989)].

2.7 SUMMARY

Nowadays, parallel computers have become a major tool that are widely used. The architectural advancement has greatly improved their performance since they were first introduced. Their prices have decreased and a lot of software tools are now available for the users. This has led to many people from different fields to use them. In the beginning, their computational power is left to the programmers to be manipulated to the fullest extent, but now, there are several commercial software tools, such as KAP [Huson et al. (1986), Macke et al. (1986)], that will aid the parallel programming. Parallel languages are now designed to include more instructions for programmers to use to exploit the parallel capabilities of the machine.

As pointed out in this chapter, there is a great need for an automatic software tool that will aid users in writing parallel programs. This includes performing the dependence analysis and transforming a program already written in a sequential manner (which most programmers nowadays are used to) into a parallel form and to take advantage of the fast concurrent processing available on these new parallel computers.

CHAPTER 3

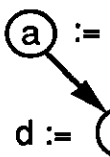
DATA DEPENDENCE ANALYSIS

3.1 INTRODUCTION

As pointed out in Chapter 1, an analysis called the **Data Dependence Analysis (DDA)**, is one of the most important tasks in a parallelizing compiler that automatically detects parallelism in sequential programs. This analysis will give information on the inter-relation of statements (or groups of statements), based on how the data in the program is computed and used. Figure 3.1 illustrates the various ways data dependences occur in programs. They may also apply to loop iterations as figure 3.2 shows. Another type of dependence that is usually found in programs is the **Control Dependence** which occurs when conditional statements are present. Once these dependences within a program unit have been determined, they may be removed by code modification or the required synchronization points and parallel mechanisms can be set to transform its parts to correctly run in parallel.

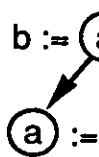
The DDA has been studied thoroughly for many years by researchers in the field of parallelizing compilers [Allen and Cocke (1976), Allen et al. (1983), Allen and Kennedy (1987), Banerjee (1988), Bernstein (1966), Kuck (1978), Kuck et al. (1981), Li (1989), Muchnick and Jones (1981), Polychronopoulos (1988), Williams (1978), Wolfe (1989a, 1989b)]. Different techniques have been employed to perform it. The majority of the work has focused on determining data dependences in loops. This chapter presents a survey on the various techniques performed by the DDA. They are classified into three main groups: the **Bernstein Method**, the **Graph-based Method** and the **Diophantine Analysis**. Section 3.2 discusses the Bernstein Method and in Section 3.3, the Graph-based method is outlined. Detection of data dependences in loops using the Diophantine Analysis is discussed in Section 3.4 while Section 3.5 briefly explains the Control Dependence. Section 3.6 summarises the chapter.

S1: $\textcircled{a} := b + c;$
S2: $d := \textcircled{a} - e;$




(a) Store first/Fetch later dependence

S1: $b := \textcircled{a} + c;$
S2: $\textcircled{a} := d - e;$



(b) Fetch first/Store later dependence

S1: $\textcircled{a} := b + c;$
S2: $\textcircled{a} := d - e;$



(c) Store dependence

Figure 3.1: Sources of Data Dependences between statements

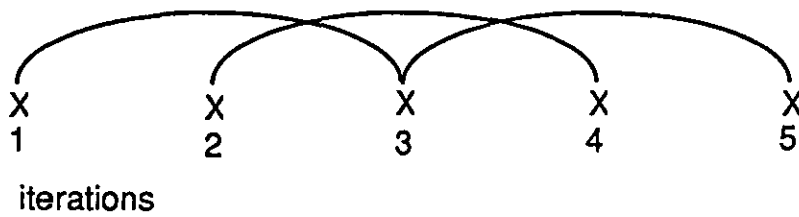
```
FOR i := 1 to n DO  
BEGIN
```

```
  S1:  $a[i] := b[i] + c[i];$ 
```

```
  S2:  $d[i] := a[i+2] - e[i];$ 
```

```
END
```

(a) Store first/Fetch later dependence



(b) Dependencies between iterations

Figure 3.2: Data Dependence in a loop

3.2 THE BERNSTEIN METHOD

The Bernstein Method is a concept originally forwarded by Bernstein (1966) to determine implicit parallelism in sequential programs. It is based on the set formation containing variables in programs. The sets show how the variables are being used, i.e., fetched and stored. These sets are termed as the Bernstein Sets (BSs) in this thesis and consist of W, X, Y and Z sets. They are defined below.

Williams (1978) has shown some approaches to detect parallelism in a sequential program, based on the BSs. She has developed a set of conditions to test parts in the program to determine whether they can be executed in parallel or not. These parts of the program are called stanzas. Each stanza will have its own BSs containing W, X, Y and Z sets.

DEFINITIONS 3.1

- (i) *A stanza is either a single program statement or a group of statements appearing adjacently in a computer program and intended to be executed one after the other.*
- (ii) *The Bernstein Sets (BSs) consist of four sets defined as follows:*
 - a. *W set - set of variables fetched during execution of stanza*
 - b. *X set - set of variables stored during execution of stanza*
 - c. *Y set - set of variables which involves a fetch and one of the succeeding operations is a store*
 - d. *Z set - set of variables which involves a store and one of the succeeding operations is a fetch*

Figures 3.3 and 3.4 show examples of stanzas and their BSs. Williams has defined six classes of relationship that can exist between stanzas in ALGOL-type programs. They are termed as Contemporary, Prerequisite, Conservative, Commutative, Consecutive and Synchronous. These relationships indicate the dependences between stanzas for data during execution. Detailed discussions are given in Williams (1978).

Contemporary stanzas, the relationships that are dealt with in this thesis, are the parallel tasks which can be executed on different processors at the same time if there exist no inter-stanza dependences. These inter-stanza dependences exist if one stanza refers to a variable whose value is altered by another stanza. To test for this contemporary property between two stanzas, they must satisfy three conditions which are termed as the Bernstein Tests (BTs).

DEFINITION 3.2

The Bernstein Tests (BTs) between two stanzas i and j , are tests to determine whether they can be run concurrently or not, i.e., if they satisfy all of the following three conditions:

$$BT_1: (X_i \cup Y_i \cup Z_i) \cap (W_j \cup Y_j \cup Z_j) = \emptyset$$

$$BT_2: (W_i \cup Y_i \cup Z_i) \cap (X_j \cup Y_j \cup Z_j) = \emptyset$$

$$BT_3: (X_i \cup Y_i \cup Z_i) \cap (X_j \cup Y_j \cup Z_j) = \emptyset$$

The above conditions are for shared-memory computers only. The operators " \cup " and " \cap " are set operators for 'union' and 'intersection' respectively. The symbol " \emptyset " denotes an empty or null set. Note that BT₁, BT₂ and BT₃ in definition 3.2 correspond to testing the data dependences in figure 3.1(a), (b) and (c) respectively. Figures 3.3 and 3.4 show examples of contemporary and non-contemporary stanzas respectively. By applying the BTs above, it can be concluded that both stanzas in figure 3.3(a) are contemporary (see figure 3.3(c)) and thus can be executed in

parallel. However, the stanzas in figure 3.4(a) are not contemporary because of the presence of data dependences on variables *a* and *f*, as the tests in figure 3.4(c) show.

An implementation of the software modules called the Analyser and the Detector to perform the above testing and how they can be embedded into an existing compiler has also been described in Williams (1979). Apart from the stanzas derived from groups of statements, Williams has also studied how the Bernstein method can be used to detect parallelism in programs containing conditional statements, loops and procedure calls. Chapter 5 of this thesis will extend the work on loops to explore the uses of the BTs and the BSs in making decisions for the detection of data dependences and loop transformations. Data dependences caused by procedure calls will be treated in Chapter 6.

3.3 THE GRAPH-BASED METHOD

This method initially is based on the Bernstein method to derive information for data dependences. However, graphs are used to represent the information captured showing the dependences between the statements (or groups of statements) in a program unit [Ferrante et al. (1987), Kuck (1978), Kuck et al. (1981, 1984), Lewis and El-Rewini (1992), Padua and Wolfe (1986), Wolfe (1989b), Zima and Chapman (1990)]. Two types of graphs have been popularly employed: the **Data Dependence Graph** and the **Iteration Space Graph**. In gathering information on data dependences, each statements in the program is analysed and the **IN** and **OUT** sets are determined. Three types of data dependences are defined and they are termed as **Flow dependences**, **Antidependences** and **Output dependences**.

<u>stanzas 1</u>	<u>stanza 2</u>
a := b + c;	d := e + f;
x := v1;	y := v2;
c := a - i;	j := j - d;
(a) Two stanzas	
<u>W, X, Y and Z sets</u>	
W ₁ - { b,v1,i }	W ₂ - { e,f,v2 }
X ₁ - { x }	X ₂ - { y }
Y ₁ - { c }	Y ₂ - { j }
Z ₁ - { a }	Z ₂ - { d }
(b) The Bernstein Sets for stanzas in (a)	
$(X_1 \cup Y_1 \cup Z_1) \cap (W_2 \cup Y_2 \cup Z_2) = \emptyset$ $(W_1 \cup Y_1 \cup Z_1) \cap (X_2 \cup Y_2 \cup Z_2) = \emptyset$ $(X_1 \cup Y_1 \cup Z_1) \cap (X_2 \cup Y_2 \cup Z_2) = \emptyset$	
(c) The Bernstein Tests	

Figure 3.3: Example of contemporary stanzas

<u>stanzas 1</u>	<u>stanza 2</u>
a := b + c;	j := a / l;
d := e * f;	m := n + o;
g := h - i;	f := q - r;
(a) Two Stanzas	
<u>W, X, Y and Z sets</u>	
W ₁ - { b,c,e,f,h,i }	W ₂ - { a,l,n,o,q,r }
X ₁ - { a,d,g }	X ₂ - { j,m,f }
Y ₁ - \emptyset	Y ₂ - \emptyset
Z ₁ - \emptyset	Z ₂ - \emptyset
(b) The Bernstein Sets for stanzas in (a)	
$(X_1 \cup Y_1 \cup Z_1) \cap (W_2 \cup Y_2 \cup Z_2) = \{ a \}$	
$(W_1 \cup Y_1 \cup Z_1) \cap (X_2 \cup Y_2 \cup Z_2) = \{ f \}$	
$(X_1 \cup Y_1 \cup Z_1) \cap (X_2 \cup Y_2 \cup Z_2) = \emptyset$	
(c) The Bernstein Tests	

Figure 3.4: Example of non-contemporary stanzas

DEFINITIONS 3.3

- (i) *The IN set contains all variables in the right-hand-side of an assignment statement.*
- (ii) *The OUT set contains variables in the left-hand-side of an assignment statement.*
- (iii) *To determine the existence of data dependences between two statements, $s1$ and $s2$, all of the following conditions must be true.*

- a. *flow dependences (fd)*

$$OUT(s1) \cap IN(s2) \neq \emptyset$$

- b. *Antidependences (ad)*

$$IN(s1) \cap OUT(s2) \neq \emptyset$$

- c. *Output dependences (od)*

$$OUT(s1) \cap OUT(s2) \neq \emptyset$$

- (iv) *A Data Dependence Graph (DDG) of a program is a graph containing nodes representing the statements and edges representing the dependences (either fd, ad or od) between the statements.*
- (v) *An Iteration Space Graph (ISG) of a d -nested loop is a graph representing a d -dimensional discrete cartesian space where each axis of the space corresponds to a loop counter.*

Similar to the Bernstein Tests, definitions 3.3iii(a), iii(b) and iii(c) correspond to detecting data dependences in figure 3.1(a), (b) and (c) respectively. After determining the data dependences and building the DDG, the compiler can begin the optimization process and restructure the program into a parallel form.

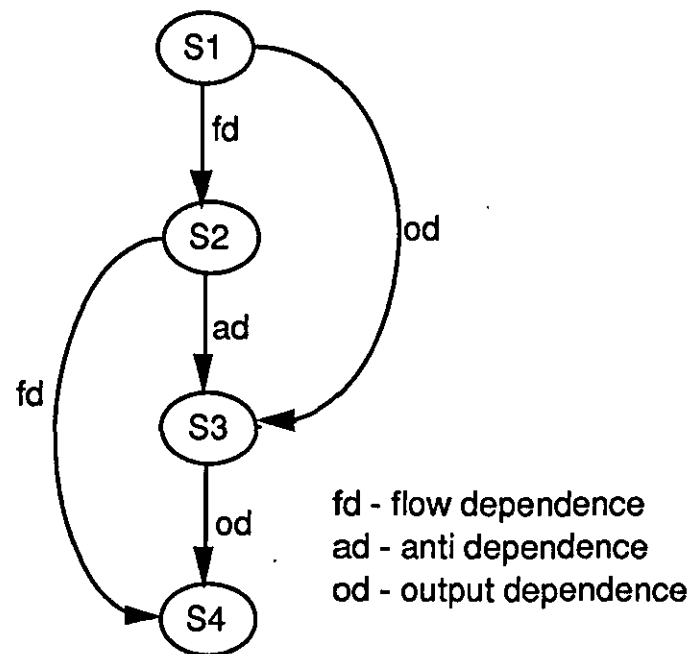
Figure 3.5 shows an example of a DDG. Loop distribution and node splitting are some of the transformation techniques that are based on a DDG [Lewis and El-Rewini (1992), Zima and Chapman (1990)].

In the other type of graph, the ISG, data dependences are represented by arrows from the point corresponding to one iteration to another whenever there exists statements S_i and S_j in the loop such that S_j is dependent on S_i , for each pair of iterations where $i \neq j$. An example of an ISG is illustrated in figure 3.6 for a two-dimensional nested loop. This representation is suitable for detecting Wavefront parallelism for the loop skewing transformation technique [Aiken and Nicolau (1990), Lewis and El-Rewini (1992)].

The Graph-based Method has been the major technique employed by most researchers in the field of parallelization of programs. A group of researchers at the University of Illinois, USA, led by David Kuck has been working on a project called Parafrase [Kuck et al. (1984), Padua and Wolfe (1986), Polychronopoulos (1990)]. Their programming languages are Fortran and C. A similar project called the Parascope project, has been carried out at Rice University [Callahan and Kennedy (1987)]. This project concentrates on developing an integrated system environment suitable for parallel programming mainly for the Fortran language. It consists of several modules which will assist users to program applications with parallel capabilities. The group has developed a system called the Parallel FORTRAN Converter (or PFC) that automatically vectorizes Fortran programs by performing a sophisticated analysis of dependences [Allen and Kennedy (1984b)]. Another group of researchers at the IBM T.J. Watson Research Center is also working on a project that will parallelize FORTRAN programs [Burke et al. (1988), Cytron et al. (1990)]. Its main module is called the Parallel TRANslator (or PTRANS) which can perform program transformation of Fortran programs from a sequential version to a parallel form. The DDG and the ISG are some of the main data structures maintained in the development of their systems.

S1: $a := 1.0;$
S2: $b := a + 3.142;$
S3: $a := 1/3 * (c - d);$
S4: $a := (b * 3.8) / 2.7183;$

(a) Group of four statements



(b) Data dependences of segment in (a)

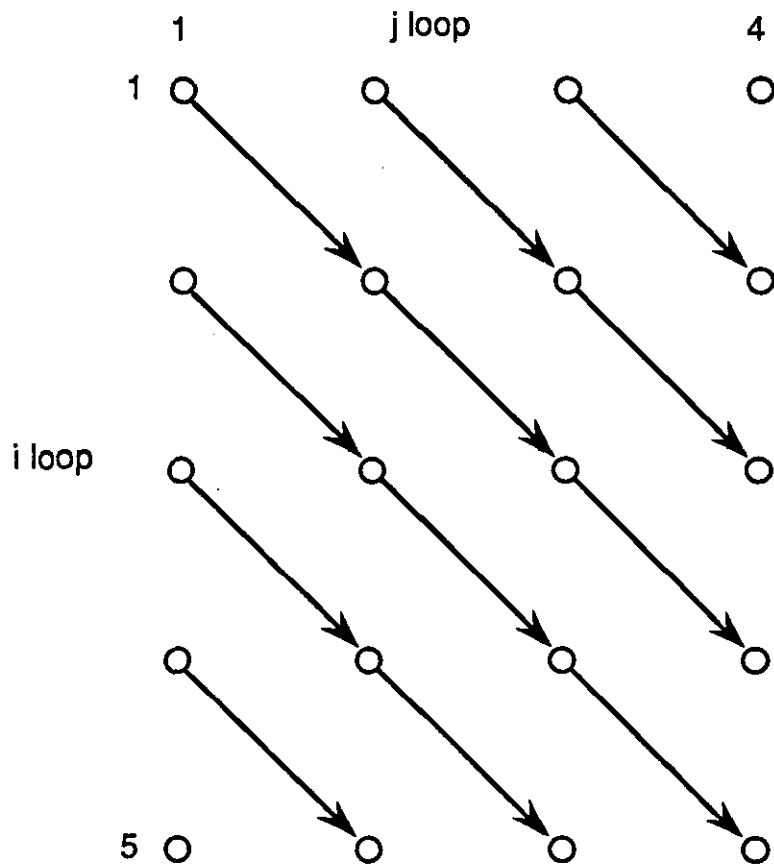
Figure 3.5: Example of a Data Dependence Graph [Zima and Chapman (1990)]

```

For i := 1 to 5 do
  For j := 1 to 4 do
    x[i+1,j+1] := x[i,j] + y [i,j];

```

(a) Two-dimensional nested loop



(b) The Iteration Space Graph with dependences

Figure 3.6: An Iteration Space Graph [Lewis and El-Rewini (1992)]

3.4 THE DIOPHANTINE ANALYSIS

The **Diophantine Analysis** involves numerical algorithms in finding data dependences in sequential loops. Loops are potentially suitable for parallelization where their iterations can be executed independently of each other on different processors. Many algorithms have been developed for this scheme [Banerjee (1988), Goff et al. (1991), Kong et al. (1991), Leung (1990), Li (1989), Maydan et al. (1991), Pugh (1992), Wolfe (1989b)].

The techniques in this analysis can be classified as Exact Tests (or definite) if they either determine the data dependences or independences. If they neither ascertain data dependences nor independences then they are called Inexact Tests (or indefinite) [Lewis and El-Rewini (1992), Zima and Chapman (1990)]. In the algorithms, Linear Diophantine Equations and greatest-common-divisor (GCD) algorithm are used to analyse the dependences. The equations represent the dependences caused by the array subscripts in the loops.

DEFINITIONS 3.4

- (i) *$GCD(i_1, \dots, i_n)$ of n numbers is the maximum $\{ b \text{ such that } i_j \mid b \text{ for all } 1 \leq j \leq n \}$ where $a \mid b$ means b divides a iff there is an integer x such that $a = bx$.*
- (ii) *A Linear Diophantine Equation has the following form:*

$$\sum_{i=1}^n a_i x_i = c$$

where $n \geq 1$, c and a_i are integers for all i , not all $a_i = 0$ and x_i are integer variables. A Diophantine equation has a solution iff $GCD(a_1, \dots, a_n) \mid c$.

Solving a system of Diophantine Equations will give the result of the dependence test for a loop. If there is a solution, data

dependence is assumed. If no solution exists, then there is no data dependence and the loop iterations are parallelizable. Figure 3.7 illustrates the use of this technique in the Exact Test for a single dimensional loop. Note that the loop index variables must be linear and do not contain symbolic terms.

The Diophantine Analysis becomes computationally expensive and inefficient when applied to loops with arbitrarily many variables, i.e., with many nested loops. No efficient method has yet been developed [Li (1989)]. Thus weaker algorithms than the Exact Test have been proposed such as the GCD Test, the Bounds Test and the Banerjee Test. Detailed discussions of these methods can be found in Allen and Kennedy (1987), Banerjee (1988), Lewis and El-Rewini (1992), Li (1989), Wolfe (1989b), Zima and Chapman (1990).

Several other algorithms have been proposed to determine exact solutions in the Diophantine Analysis. The Lambda-test is one such algorithm that efficiently tests for data dependence to give a precise result [Li (1989), Li et al. (1989), Li and Yew (1990)]. This test is particularly effective in handling coupled subscripts, i.e., subscripts with some loop index appearing in more than one dimension. Another technique, called the I-test, has been proposed to produce a definite positive answer [Kong et al. (1991)]. It is a refinement of the GCD and Banerjee tests which checks for the existence of integer solutions. Pugh (1992) has developed the Omega-test, based on integer programming. Li and Yew (1990) have argued that an integer programming method is extremely slow. However, Pugh has showed that his technique can be used to determine an integer solution for an arbitrary set of linear equations and inequalities. In Goff et al. (1991), another test, called the Delta-test, which is based on classifying pairs of subscripted variable references, is said to produce a solution. Another method using the Diophantine Analysis has also been described by Maydan et al. (1991).


```

FOR i := 1 to 101 do
BEGIN
    a[2*i] := _____;
    _____ := a[3*i+198];
END;

```

The Diophantine equation: $2x = 3y + 198$
 where: $1 \leq x, y \leq 101$

$\text{GCD}(2, -3) = 1$

General solutions: $x = 3t + 396$
 $y = 2t + 198$

where t is an arbitrary integer.

The constraints on t :

$$\begin{aligned}
 1 \leq 3t + 396 \leq 101 & \Rightarrow -131 \leq t \leq -99 \\
 1 \leq 2t + 198 \leq 101 & \Rightarrow -98 \leq t \leq -49
 \end{aligned}$$

Conclusion:

Since $t \leq -99$ contradicts $-99 \leq t$, then the Diophantine Equation does not have a solution that satisfies the given constraints. Hence, the loop iterations are independent.

Figure 3.7: Diophantine Analysis for the Exact Test [Lewis and El-Rewini (1992)]

3.5 CONTROL DEPENDENCES

Most of the discussions on dependences in programs have concentrated on the Data Dependence. Another type of dependence, called the Control Dependence, is one that appears in programs containing conditional statements [Allen et al. (1983), Cytron et al. (1990, 1991), Ferrante et al. (1987), Padua and Wolfe (1986), Zima and Chapman (1990)]. As an example, in the following conditional statement:

```
IF a > b THEN
    max := a
ELSE
    max := b;
```

the two statements 'max := a' and 'max := b' are control dependent on the condition 'a > b'. To represent these dependences, **Control Flow Graphs** are used [Zima and Chapman (1990)].

There have been several proposals suggested on how to handle the control dependences. One technique is to transform them into data dependences which are then treated as discussed in the previous sections [Allen et al. (1983), Padua et al. (1980), Padua and Wolfe (1986)]. This scheme is briefly discussed in Chapter 5 of this thesis. Some authors have suggested a method to combine both the data dependence and the control dependence in one representation such as the program dependence graph (PDG). Details can be found in Ferrante et al. (1987).

3.6 SUMMARY

This chapter has presented a survey on the methods performed in the Data dependence analysis (DDA). The DDA is an important part of a parallelizing compiler in discovering any implicit parallelism that may exist in sequential programs. The three main categories of techniques mainly used to perform the DDA surveyed in this chapter are the Bernstein Method, the Graph-based Method and the Diophantine Analysis.

Most of the work on the DDA carried out by researchers are based on the Graph-based method. The Bernstein Method, however, has not been pursued in great detail, except by Williams (1978). In this thesis, the Bernstein Method becomes the basis of the research study. The third category of techniques, the Diophantine Analysis, is particularly suited in detecting dependences in loops. They give quite accurate results for array references with complicated subscript expressions. However, their computation may be slow especially when the level of nesting in the loops increases.

CHAPTER 4

A TOOL FOR AUTOMATIC DETERMINATION OF PARALLELISM

4.1 INTRODUCTION

With the availability of the multi-processor computer, in which each processor can execute different parts of a program in parallel, the task of programming in parallel has increased. Programs targeted for parallel execution have to be coded in a special way in order to take advantage of the parallel capabilities provided by the machines. The programmer must be able to carry out analysis to identify any parallelizable parts and to ensure that they are free from any data dependences. These tasks can be greatly reduced if there exists a sophisticated software tool to perform the analysis automatically.

Implementations of such a software tool, commonly known as the parallelizing compiler, are already available. With their aid, a programmer can write a program without having to think in parallel terms. The program can then be analysed and transformed into its parallel version. Ideally, the whole process of compiling and restructuring a program should be done by the compiler itself [Allen (1988), Allen and Kennedy (1984b), Appelbe and Smith (1989), Cowell and Thompson (1990), Guarna et al. (1989), Kuck et al. (1984), Polychronopoulos (1988)].

The next step after the development of a parallel program, is to map or schedule the concurrent tasks in the program onto a target parallel machine [Kruatrachue and Lewis (1988), Polychronopoulos (1988)]. The scheduling process, carried by the operating systems, has to be performed with an objective of attaining an optimal overall execution time for the program. This consideration involves many factors such as the size of the tasks, their communication times, the number of processors and the strategy of task assignment to processors.

The main problem addressed in this chapter is the determination of implicit parallelism in a sequential program and how to maximize it during scheduling. It includes an automatic identification of the size of task granularity that gives the best execution performance of the program. Any implicit parallelism

that exists will be determined by partitioning the program into concurrent tasks called stanzas [Williams (1978)]. These stanzas are then scheduled on a shared-memory parallel computer to find the optimal execution time and the optimal stanza granularity. Heuristics are developed to find these solutions. A software tool has been developed to carry out the above functions.

The organization of this chapter is as follows. Section 4.2 briefly explains the concepts used in the determination of implicit parallelism in sequential programs. Sections 4.3 and 4.4 discuss the issues in the stanza scheduling and in the determination of optimal stanza granularity respectively. In Section 4.5, a description of the software tool called TAG is given with some example outputs in Section 4.6. Section 4.7 gives a brief comment on scheduling of loops and in Section 4.8 a summary of the chapter is given.

4.2 DETECTION OF IMPLICIT PARALLELISM

As discussed in Chapter 3, Williams (1978) has developed approaches to detect implicit parallelism in a sequential program. The program is partitioned into groups of statements called stanzas. She develops a set of conditions (termed as the Bernstein Tests (BTs) in this thesis) to test the stanzas to determine whether they can be executed in parallel or not. The technique is based on the Bernstein Sets (BSs) [Bernstein (1966)]. Based on this concept, this chapter studies how the inter-relations of stanzas due to data dependences affect their scheduling for the shortest execution time. Figure 4.1 shows the whole process of detecting any implicit parallelism in sequential programs. The information about the usage of variables in the stanzas will be in the form of the BSs.

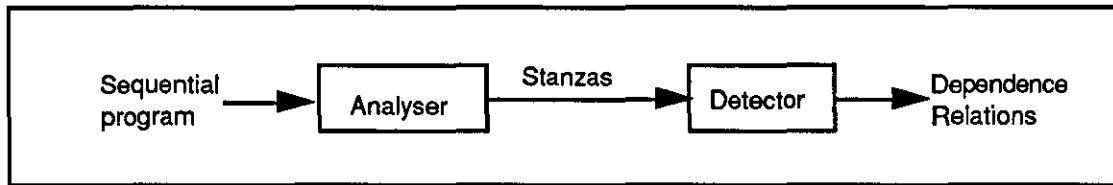


Figure 4.1: Determination of implicit parallelism in sequential programs

4.3 SCHEDULING OF CONCURRENT STANZAS

Programs containing concurrent stanzas (or tasks) need to be scheduled or mapped carefully onto a multi-processor system during execution [Bokhari (1988), Duda (1988), Girkar and Polychronopoulos (1988), Kruatrachue and Lewis (1988), Polychronopoulos (1988), Sahni (1984), Sarkar (1989)]. The main goal is to determine the best schedule which will give an optimal execution time, that is, the shortest possible execution time for the program on a certain parallel processor system. As an example, if a program has n independent concurrent stanzas, each of size e , then these stanzas can be run on n processors in the unit time of e as compared to $(e \cdot n)$ units of time for a sequential execution. However, if some of the stanzas are dependent on others (called the predecessor stanzas), then they have to be assigned to processors with great care in order to get an optimal execution time and to have maximum parallelism.

Program scheduling can be divided into two categories, static scheduling and dynamic scheduling. The static scheduling is performed at compile time, before the program is actually executed. It is based on the global program information gathered during compilation and it is an approximation. The second scheme, the dynamic scheduling is done at run-time and this incurs an overhead in assigning the stanzas to processors. However, it has the capability to schedule the stanzas with more information readily available. The goals of dynamic scheduling are to have a well balanced load for all processors by keeping

them as busy as possible and to keep the run-time overhead minimum [Polychronopoulos (1988)].

The scheduling problem has been studied theoretically and has been shown to be NP-complete [Coffman (1976), Garey and Johnson (1979), Sahni (1984)]. This means that, to obtain an optimal solution in its general form will require a considerably large (that is, exponential) computation time. However, heuristics can be developed that will produce near optimal solutions. These heuristics are simple, easy to implement and usually have low complexity [Butt (1993), Kruatrachue and Lewis (1988)].

A common scheme in stanza scheduling is the list scheduling [Adam et al. (1974), Lewis and El-Rewini (1992)]. In this scheme, a stanza is assigned as soon as a processor is available. However, before the processor can execute the stanza, it is kept idle while waiting for all of its predecessor stanzas to complete their executions. In the list scheduling, load balancing is a strategy where it tries to ensure that all processors are kept as busy as possible at all time and that they finish at the same time [Butt (1993), Kruatrachue and Lewis (1988)].

In general, the input to a scheduling process (called a scheduler) are a set of stanzas, their sizes, the communication costs and the dependence relations between the stanzas. The output is a schedule for a multi-processor system. Before scheduling begins, the relevant information have to be determined first. The stanzas can be formed and their sizes estimated during the compilation of the program. The dependence relations are derived by performing the dependence tests on the stanzas.

The communication overhead is due to inter-stanza data dependences and synchronization of stanzas [Axelrod (1986), Duda (1988), Kruatrachue and Lewis (1988), Li and Abu-Sufah (1985), Su (1992)]. The inter-stanza overhead is the extra time needed for data transfer from one processor to another. Synchronization on the other hand, creates a situation where stanzas have to wait until others have completed their jobs in

order to be able to proceed in execution. An example of this is the barrier instruction [Osterhaug (1987)]. In this chapter, only overheads due to data transfer are considered and it is assumed to be constant. Figure 4.2 illustrates the effects of the communication overhead on the processor allocation. $C1$ and $C2$ are the communication times needed for data transfers. It shows in figure 4.2(e) that the presence of large overhead for the data transfer from $P1$ to $P2$ (as compared to the size of stanza 2) prevents stanzas 2 and 3 from being executed in parallel.

In conclusion, having more than one processor to solve a problem does not automatically guarantee that the execution time will be shorter. Sometimes, to run the same program on a single-processor machine can be a lot faster because there is no communication overhead. If a program is scheduled by ignoring the overhead, then a perfectly balanced load with all processors finishing at time $T_{balance}$ can be achieved. However, if the program is scheduled with the aim of minimizing overhead, then it will give an unbalanced load on the processors with all finishing at different times. The last processor finishes at time $T_{overhead}$ with $T_{overhead} < T_{balance}$. This is the min-max problem that a scheduler has to solve, i.e., to produce a schedule with maximum parallelism and with minimum overhead [Kruatrachue and Lewis (1988)].

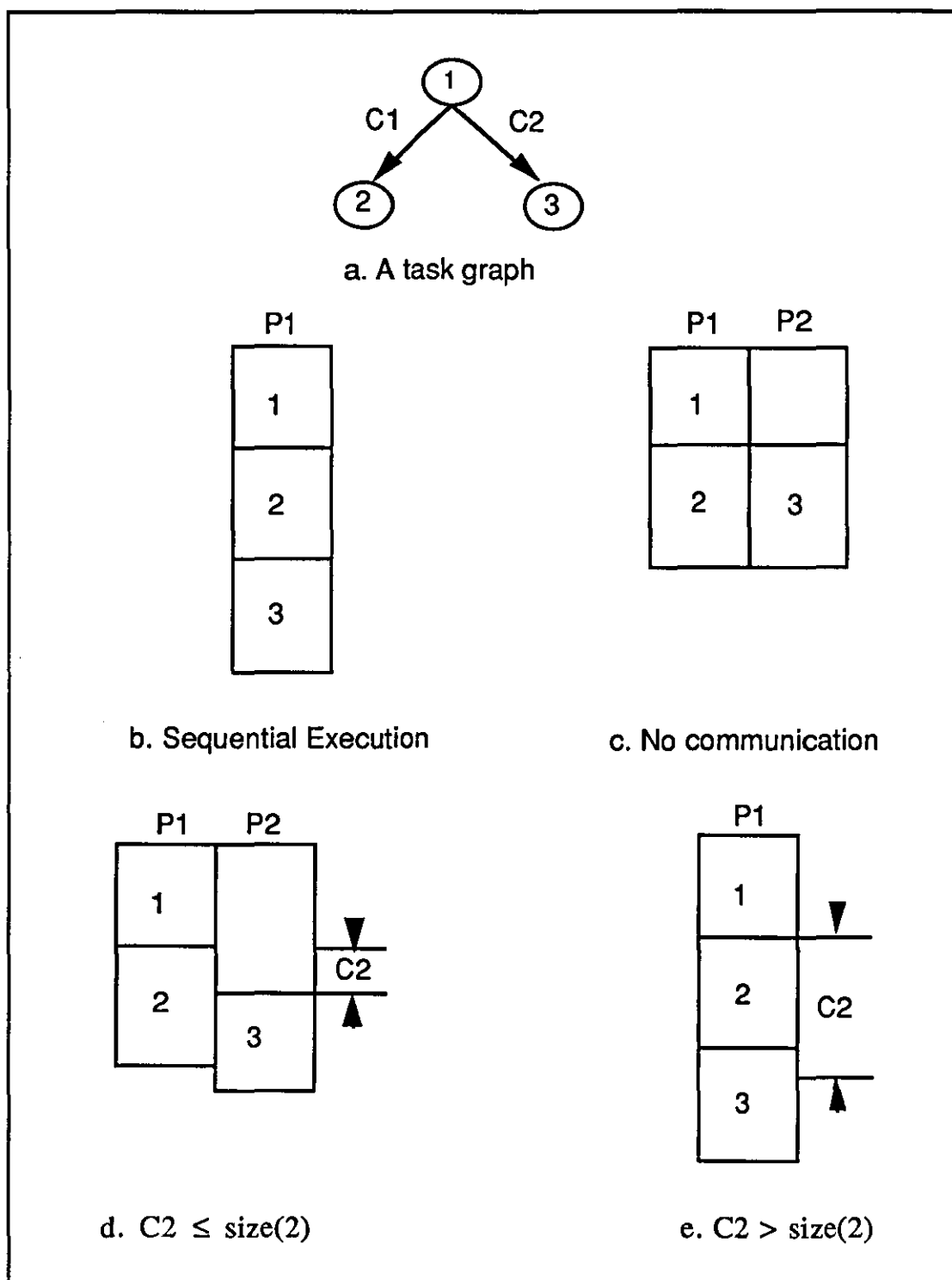


Figure 4.2: Effects of communication overhead

4.4 DETERMINATION OF STANZA GRANULARITY

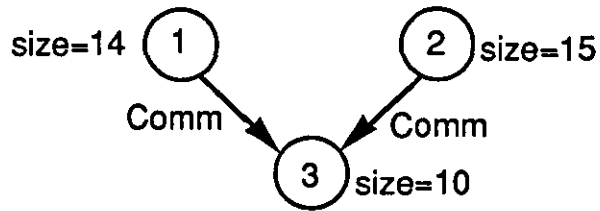
The determination of stanza granularity is a partitioning problem. It is a process of breaking down a program into a set of stanzas (tasks) suitable for parallel execution [Girkar (1991), Kruatrachue and Lewis (1988), Kwan et al. (1990), McCreary and Gill (1989), Polychronopoulos (1988), Sarkar (1989)]. A grain is defined as a module containing one or more stanzas that has to be executed in a sequential manner by a single processor. Polychronopoulos (1988) defines the size of a stanza derived entirely from the syntax of the underlying language. For languages such as Pascal, C and Fortran, the stanzas are the loop body, procedure calls and basic assignment blocks (BAS). Williams (1978) has limited the maximum size of a stanza to be a group of statements with a maximum number of variables of 15 and it is not necessarily a BAS. This however, does not ensure an optimal stanza size. In this chapter, a stanza can be a statement, a block of statements delimited by begin-end block as in Pascal, a loop or a procedure call.

The problem is to determine the best stanza size that will give the shortest execution time. A large grain size will limit potential parallelism. Small grain, however, will result in greater communication overhead and may cause execution time degradation. This needs a good automatic merging (or packing) strategy to decide which stanzas are best executed on the same processor. Together with the scheduling process, they will have to balance between the possible parallelism and the communication overhead to achieve the best grain size. It has been shown that the general solution to this granularity problem is NP-complete but a near-optimal solution to a subproblem can be determined [Garey and Johnson (1979), Kruatrachue and Lewis (1988), Sarkar (1989)]. In this chapter, a heuristic is developed to determine this near-optimal stanza size.

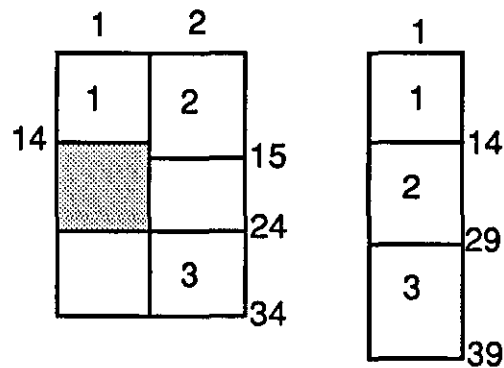
The way stanzas are merged is crucial. Sometimes, merging produces groups of stanzas which give degrading execution time.

This is illustrated in figure 4.3. The main factors that govern this merging operation are the stanza size, the communication times and the dependence relations. It is essential to determine before merging, the effects of these factors. If it proves to degrade the schedule time, then the stanzas should be left unmerged. However, since the study described in this chapter involves heuristics, then improved solutions cannot always be guaranteed. This is because the merging operation restructures the dependence relations of the new sets of stanzas and this may create less parallelism.

Most of the work done by researchers assumes an input to a scheduler is a parallel program in the form of task graph. Kwan et al. (1990) uses the Critical Path Analysis to improve the performance of parallel programs. Aggregating tasks by forming clans as grains have been proposed by McCreary and Gill (1989). These clans can then be assigned to parallel processors to achieve an optimal execution time. Sarkar (1989) proposes two models for partitioning and scheduling task graphs, called the macro-dataflow model (compile-time partitioning and run-time scheduling) and the compile-time scheduling model (partitioning and scheduling at compile-time). Polychronopoulos (1988) also uses the task graphs to model the program to be scheduled. A Critical Process Size (CPS) is estimated for each task and the size of processes are determined, based on this CPS. The CPS is the minimum size of a process whose execution time is equal to the overhead that it incurs during scheduling. Kruatrachue and Lewis (1988) have developed a method to optimize parallel programs called grain packing which will reduce total parallel execution time by balancing the sequential execution time and communication time. Their Duplicating Scheduling Heuristic duplicates tasks where necessary to reduce overall communication delays and maximizing parallelism at the same time.

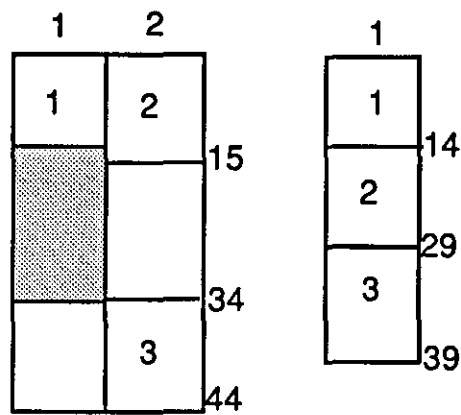


(a) A stanza with two predecessors



(i) unmerged (ii) merged

(b) Merging degrades the performance if Comm=10



(i) unmerged (ii) merged

(b) Merging improves the performance if Comm=20

Figure 4.3: The effects of stanza and communication sizes on merging operation.

Bieler (1990) has studied the partitioning of parallel programs written in UNITY by developing the d-graphs of the programs. These d-graphs are then mapped onto a parallel processor. d-graphs are graphs with two edges, weak edges and solid edges. Statements connected by weak edges are suitable for allocation in different processors.

4.5 TAG: A TOOL FOR AUTOMATIC DETERMINATION OF PROGRAM GRANULARITY

In this section, a software tool called TAG (Tool for Automatic determination of program Granularity) for detecting potential parallelism in a sequential program in the form of stanzas is presented [Evans and Mohd-Saman (1993)]. These stanzas are then scheduled by TAG for a shared-memory multi-processor system. It is extended to find the best stanza size (or the grain size) for near-optimal parallel execution time by the process of scheduling and merging. Its overall structure is shown in figure 4.4. It contains four main modules:

- a. the ANALYSER module which scans a sequential program and forms the basic stanzas
- b. the DETECTOR module which performs the dependence test (BTs) to determine which stanzas are concurrent
- c. the SCHEDULER module which schedules the stanzas onto a multi-processor system to give the fastest parallel execution time
- d. the MERGER module which merges stanzas to determine program granularity

Appendix A shows the main program for TAG while Appendix B and Appendix C contain the codes for the Scheduler and Merger modules respectively. The Analyser and Detector modules are similar to those described in Williams (1978).

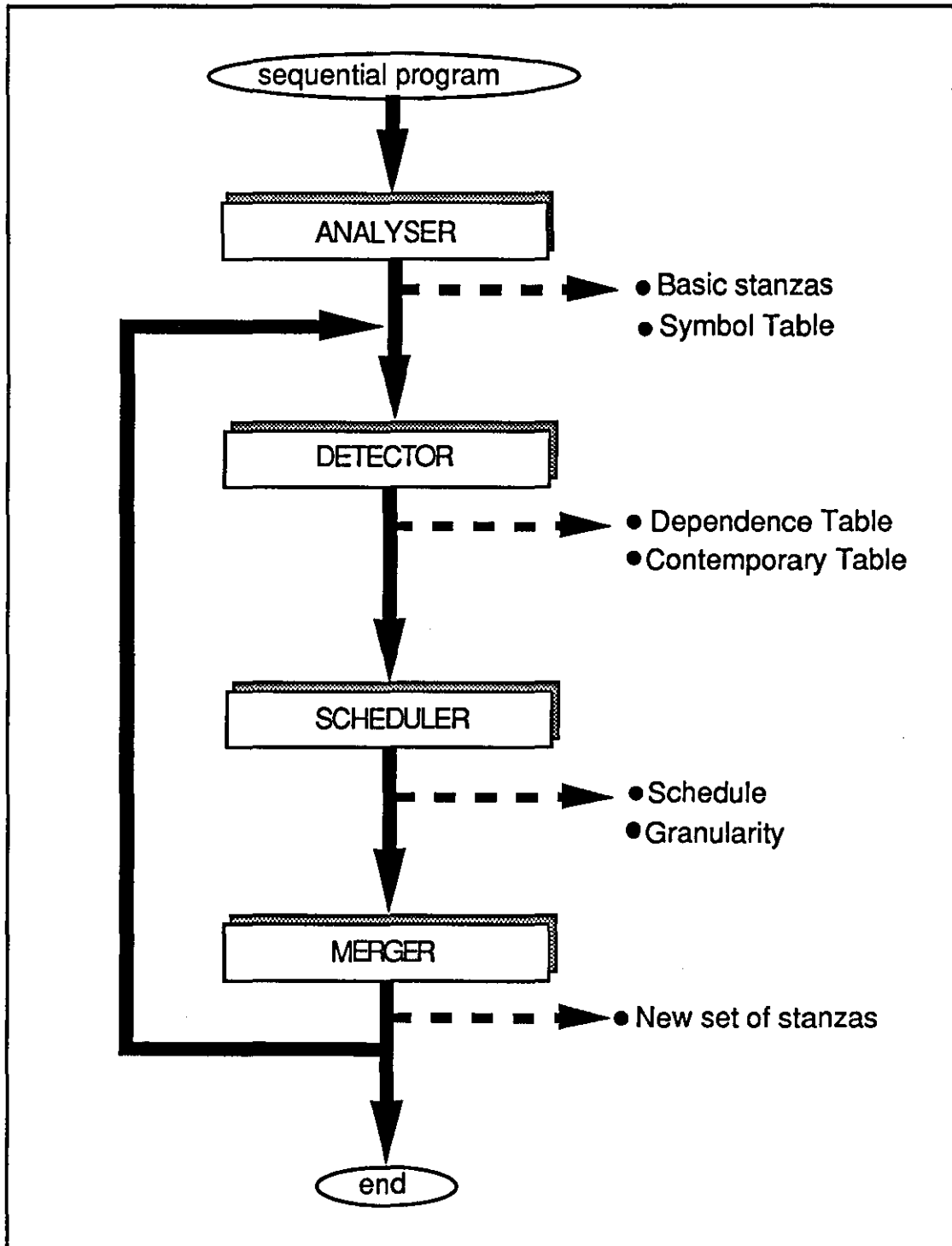


Figure 4.4: The Overall Structure of TAG Software Tool

4.5.1 The Analyser

The **Analyser** module accepts a subset of Pascal program as input and performs a lexical analysis to check its syntactic correctness [Aho et al. (1986)]. It then forms the basic stanzas based on each statement (or block of statements). Each statement is analysed to determine the contents of the BSs, i.e, the W, X, Y and Z sets. It also estimates the sequential execution time and the communication time for each stanza for the purpose of scheduling. Figure 4.5 shows the declaration of a stanza. In scanning the source program, all variable names found in it are stored in a symbol table. The BSs structure only keeps the index values of the variables in the symbol table. This Analyser can be modified and adapted to accept other programming languages such as MODULA-2 [Walmsley and Williams (1990)] or C [Kernighan and Ritchie (1988)]. Figure 4.6 shows an example of a program and its corresponding information on single-statement stanzas in terms of the BSs.

For the timing estimates, an arbitrary unit value is assigned to the execution time and the communication time of each stanza. This suffices since this study is concerned with the relative behaviour of the scheduling process. The times may assume any other values. Thus, for conformity, the assignment statement and the addition operation take 1 unit of time each and the multiplication operation takes 10 units of time. The time for a stanza to communicate to other stanzas is assumed to be a constant for all stanzas and is read at the beginning of the execution of TAG. It usually takes a value between 10 to 20 units of time.


```

struct bernstein
{
    int stanzatype;      /* type of stanza */
    int etime;           /* execution time */
    int ctime;           /* communication time */
    int bcnt[4];         /* W,X,Y,Z set counters */
    int bset[4][maxident]; /* W,X,Y,Z set contents */
} stanza

```

Figure 4.5: Definition of a stanza

4.5.2 The Detector

The list of stanza information in the form of BSs derived by the Analyser module is then passed to the **Detector** module. This module performs the BTs on the BSs to determine the data dependences. To do this, tables of XYZ and WYZ (i.e., $(X \cup Y \cup Z)$ and $(W \cup Y \cup Z)$), are first formed, based on the contents of the BSs. Then the \cap operation is applied to the contents of the XYZ_i and WYZ_j , where $1 \leq i, j \leq \text{number of stanzas}$ and $i \neq j$. For the \cup and \cap operations, they are performed in bit-wise method. For this purpose, two working spaces are used and they are set to 0 or 1 to indicate the presence of each variable, based on its position in the symbol table.

The Detector will create two tables as outputs, the Dependence Table and the Contemporary Table. The Dependence Table gives information about the predecessor stanzas on which another stanza depends for data during execution. The Contemporary Table contains groups of stanzas which are concurrent and can be executed in parallel. Figure 4.7 shows examples of a Dependence Table and a Contemporary Table of the simple program from figure 4.6(a). Figure 4.7(b) indicates that stanzas 1, 2 and 3 are independent and stanzas 4, 5 and 6 have predecessor

stanzas. There are four concurrent groups in the Contemporary Table as shown by figure 4.7(c).

4.5.3 The Scheduler

By using the information available in the basic stanzas, the Dependence Table and the Contemporary Table, the Scheduler determines the best schedule by assigning the stanzas onto a parallel machine with two or more processors. This is done by balancing the communication time and the maximum parallelism that can be achieved. A 'quick and dirty' algorithm is used to generate the schedule as fast as possible [Spyropoulos (1979)]. At the end of the process, it estimates the parallel execution time and the sequential time. From these two results, the speed-up value is derived as follows.

$$\text{speed-up} = \frac{\text{parallel execution time}}{\text{sequential time}}$$

Figure 4.8 shows a series of schedules for the program in figure 4.6(a).

In the stanza assignment to processor, the main strategy used by TAG is called the **largest-contemporary-stanza-first allocation**. In this scheme, for each group of stanzas in the Contemporary Table, their contents are sorted in descending stanza size order. Then for each stanza S in the sorted group, its predecessor stanzas are determined from the Dependence Table. If S is independent (i.e., it has no predecessors), then it is allocated to a processor with the lowest current finishing time. Figure 4.9 shows an example of this process.

```

program pr46;
begin
  n1 := a1*b1;
  n2 := a2*b2;
  n3 := a3*b3;
  n4 := n1;
  n5 := n2-n3;
  n6 := n4*n5;
end.

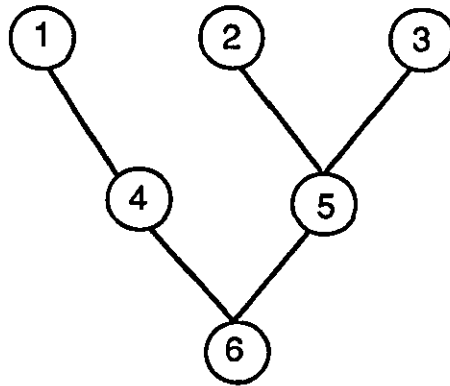
```

(a) A simple program

STANZA	W sets	X sets	Y sets	Z sets	EXEC TIME	COMM TIME
1	a 1 b 1	n 1	-	-	12	20
2	a 2 b 2	n 2	-	-	12	20
3	a 3 b 3	n 3	-	-	12	20
4	n 1	n 4	-	-	2	20
5	n 2 n 3	n 5	-	-	3	20
6	n 4 n 5	n 6	-	-	12	20
Total sequential time =					53	

(b) The Bernstein Sets

Figure 4.6: An example program with its BSs



(a) A task graph for the program in figure 4.6(a)

stanza - { predecessor stanza set }

1	- \emptyset
2	- \emptyset
3	- \emptyset
4	- { 1 }
5	- { 2, 3 }
6	- { 4, 5 }

b. Dependence Table

stanza { concurrent set }

1 -	{ 2 3 6 }
2 -	{ 3 4 }
3 -	{ 4 }
4 -	{ 5 }
5 -	\emptyset
6 -	\emptyset

c. Contemporary Table

Figure 4.7: Dependence and Contemporary Tables

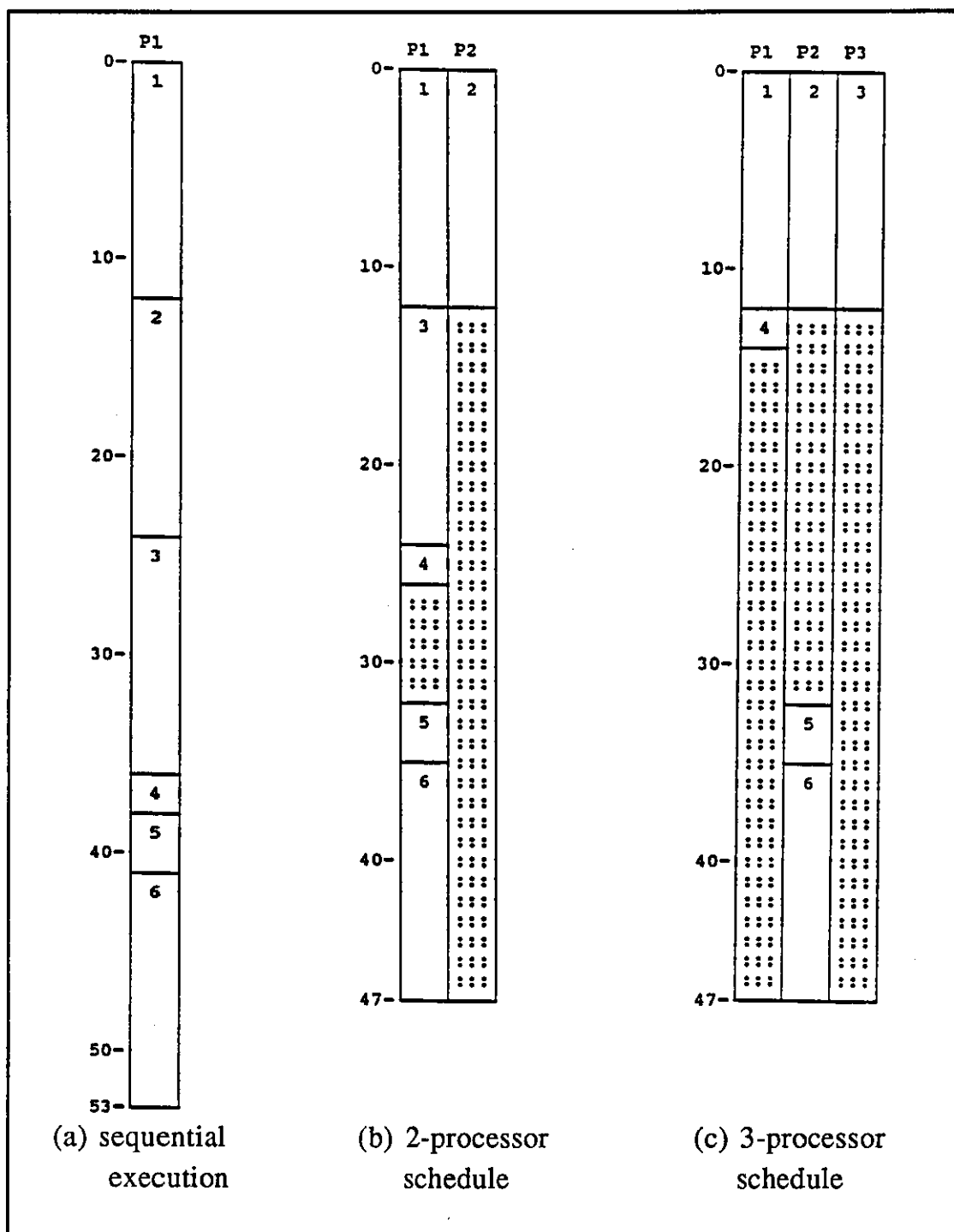


Figure 4.8: The schedules generated for the program in figure 4.6(a).

Let $\text{finish_time}(S)$ be the finish time for stanza S on a processor and $\text{comm}(S)$ be the communication time for the processor executing S to transfer its value to other processors. Therefore, if it is assumed that:

$$\text{finish_time}(S3) < \text{finish_time}(S2) < \text{finish_time}(S4) < \text{finish_time}(S1)$$

then stanza $S5$ is allocated on processor 3 since $\text{finish_time}(S3)$ is the lowest of all.

However, if S is not independent but has all of its predecessor stanzas allocated already, then it will be assigned to a processor with the earliest execution time after considering the effects of communication overhead. In general, to assign the stanza, the finishing times of all of its predecessors are checked first. It is then assigned to a processor with the highest ($\text{finish_time} + \text{communication time}$). An illustration of the scheduling for a non-independent stanza is given in figure 4.10. If it is assumed that the communication times are the same and constant, then stanza $S5$ (which depends for data on $S1$, $S3$ and $S4$) is assigned to processor 1 since $(\text{finish_time}(S1) + \text{comm}(S1))$ is the highest. This will minimize the processor idle time and hence the total execution time. Its starting time will be at the next highest time, that is, at $(\text{finish_time}(S4) + \text{comm}(S4))$ because:

$$(\text{finish_time}(S4) + \text{comm}(S4)) > (\text{finish_time}(S3) + \text{comm}(S3))$$

The shaded area in the chart for figure 4.10 is the communication delay time for P1. For those stanzas which have one or more of their predecessor that has not been assigned yet, their allocations are delayed until later. The general scheduling algorithm is shown in figure 4.11.

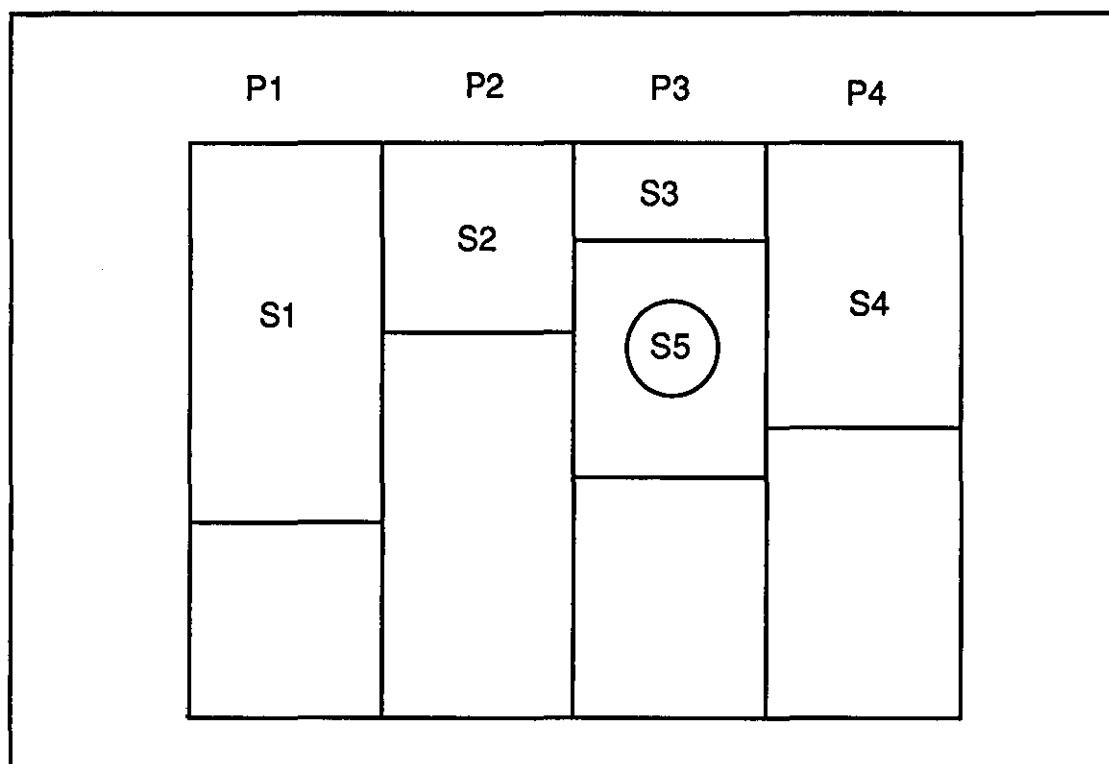


Figure 4.9: Allocation of an independent stanza S5

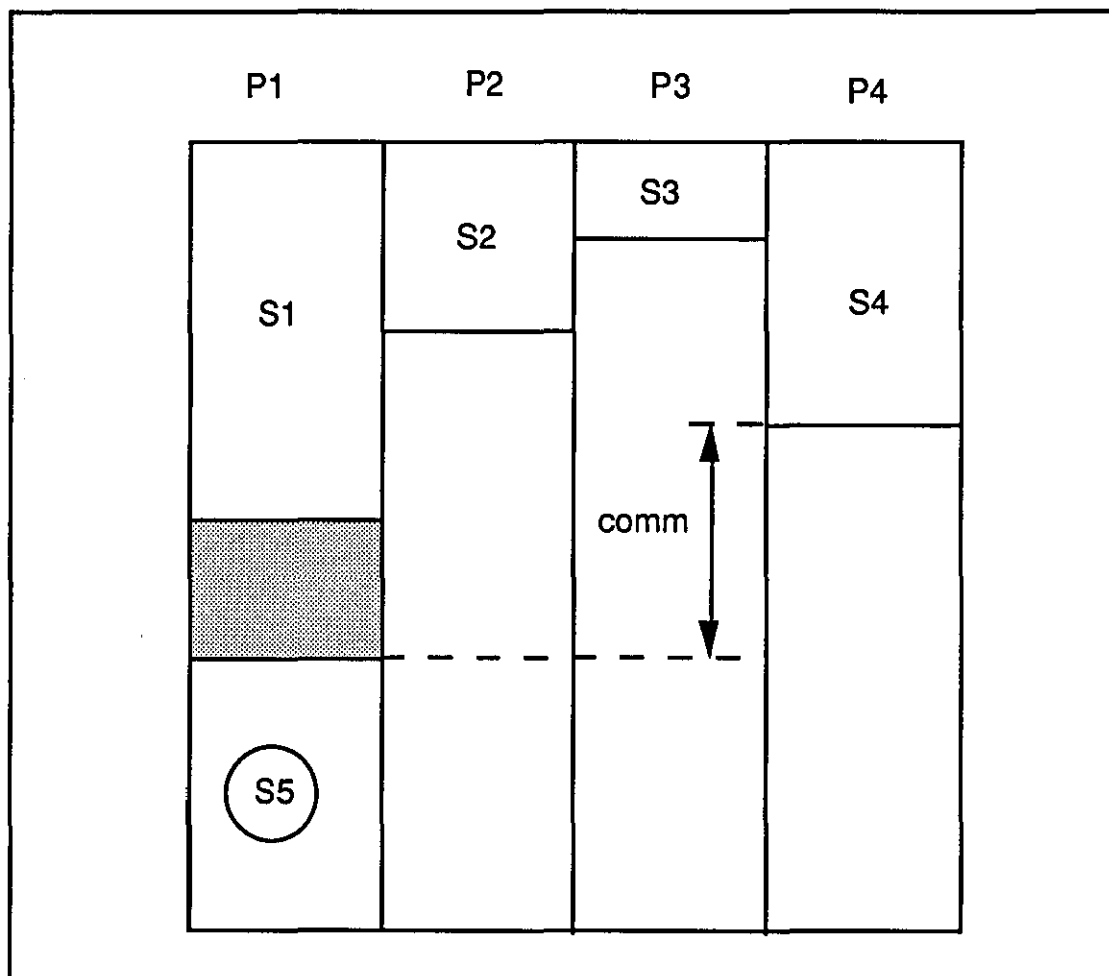


Figure 4.10: Allocation of a non-independent stanza, S5


```

INPUT:    List of stanzas, Contemporary Table, Dependence
          Table
OUTPUT:   A Schedule Table for a multi-processor with n
          processors
BEGIN
  FOR each group i of the concurrent stanza in the
    Contemporary Table,
  BEGIN
    SORT the stanzas into a descending order of stanza
      size and call it the Sorted Group
    FOR each stanza S in the Sorted Group
    BEGIN
      DETERMINE the Predecessor Stanzas of S from
        the Dependence Table
      IF S is independent
      THEN allocate it to an available processor
        having the lowest finishing execution
        time
      IF S is dependent on other stanzas AND all of
        the predecessor stanzas have been
        allocated,
      THEN assign S on a processor of a predecessor
        stanza with the highest (finishing
        execution time + communication time)
      ELSE delay its allocation
    END
  END
  CALCULATE the estimated parallel execution time
END

```

Figure 4.11: Algorithm for scheduling stanzas

4.5.4 The Merger

In order to reduce the communication overhead, some stanzas have to be merged so that they will be executed on the same processor. This is performed by the **Merger** module which forms bigger stanzas from the basic ones. It is assumed that this process will add the execution times of the stanzas but still maintain the same communication time [Kruatrachue and Lewis (1988)]. The merging of stanzas is based on the following principles.

Given a stanza S and its predecessors PSs :

- a. they will be merged if none of PSs has been merged with others.
- b. let PS_i be the largest of all PSs and $comm(PSs)$ be the communication times for PSs to transfer data to any other stanzas. Then for all PS_j (where $i \neq j$), they are merged with PS_i iff:

$$(comm(PS_j)) > (size(PS_i)).$$

Otherwise they are left unmerged. This is to ensure that the merge operation will not give a new stanza whose execution time is higher than that of the unmerged stanzas. Figure 4.3 shown earlier illustrates this point.

When the BSs of two stanzas are merged, it produces new BSs whose contents depends on which sets the variables are members of before merging. For example, if a variable v is a member of the X set in BS_1 and it is also a member in the W set in BS_2 , then it will be included in the Z set of the resulting BSs. Figure 4.12 shows the algorithm to merge any two BSs of stanzas i and j resulting in stanza k . It should be noted that $MERGE(S_1, S_2)$ will not necessarily give the same result as $MERGE(S_2, S_1)$.

```

INPUT:   Stanzas  $S_i$  (with  $W_i, X_i, Y_i, Z_i$  sets) and
          $S_j$  (with  $W_j, X_j, Y_j, Z_j$  sets)
OUTPUT:  Stanza  $S_k$  (with  $W_k, X_k, Y_k, Z_k$  sets)
BEGIN
  FOR each variable  $v$  in the  $W_i, X_i, Y_i$  and  $Z_i$  sets of  $S_i$ ,
    search for it in stanza  $S_j$ .
  BEGIN
    IF  $v$  is an element of  $W_i$ 
      IF  $v$  an element of  $W_j$  or NOT found,
        THEN  $v$  is an element of  $W_k$ 
      ELSE  $v$  is an element of  $Y_k$ 
    IF  $v$  is an element of  $X_i$ 
      IF  $v$  an element of  $X_j$  or NOT found,
        THEN  $v$  is an element of  $X_k$ 
      ELSE  $v$  is an element of  $Z_k$ 
    IF  $v$  is an element of  $Y_i$ 
      THEN  $v$  is an element of  $Y_k$ 
    IF  $v$  is an element of  $Z_i$ 
      THEN  $v$  is an element of  $Z_k$ 
  END
  FOR all variable  $v$  in stanza  $S_j$  NOT found in stanza  $S_i$ 
    then  $v$  is an element of its original set of stanza  $S_k$ 
END

```

Figure 4.12: The Merge Algorithm

Once the merging process has been completed, an average granularity size of the new stanzas and new dependence relations are determined. Then, another process of scheduling is carried out to determine a new schedule with another estimated parallel execution time and speed-up factor. This process of merging and scheduling is repeated until no more merging operation is carried out. A schedule with the best execution time will be taken as the near-optimal execution time and its average stanza size is the near-optimal average grain size. Figure 4.13 illustrates the effects of merging for stanzas in figure 4.6(b).

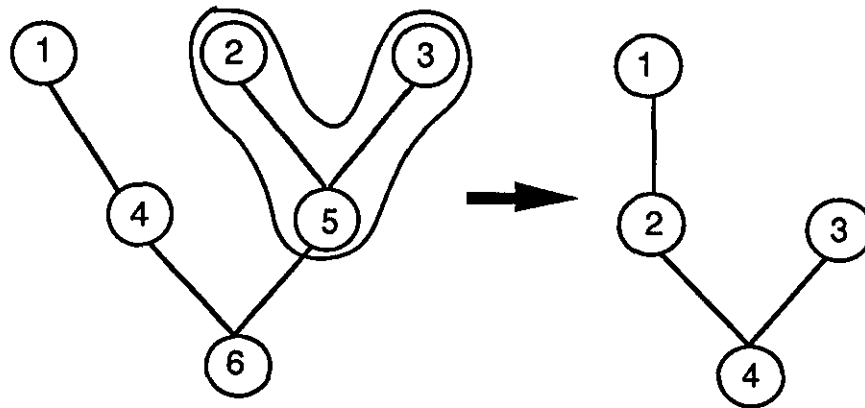
4.6 EXAMPLE OUTPUT OF TAG

In this section, more examples of the output of TAG are presented.

EXAMPLE 4.6.1: Figure 4.14 shows a simple program that performs a summation of $(a_i * a_j)$ and its graphical representation in the form of a task graph depicting the inter-dependence of the statements.

Figure 4.15 shows an output produced by the Analyser module. It gives information for the BSs of each stanza for the program. Note that the communication time for each stanza has been fixed as 10 units during execution. This can be changed to other values. The Dependence Table and the Contemporary Table produced by the Detector module are shown in figure 4.16.

The diagrams in figure 4.17 show the first merging operation of stanzas 9, 10 and 13 to form a new stanza 9 and stanzas 11, 12 and 14 to form a new stanza 10. They are merged after considering the effects of their sizes and the communication overhead. Stanzas 1, 2 and 9 or stanzas 3, 4 and 10 or stanzas 5, 6 and 11 or stanzas 7, 8 and 12 have not been merged because merging will degrade their execution times.



(a) Stanzas 2,3 and 5 are merged

STANZA	W sets	X sets	Y sets	Z sets	EXEC TIME	COMM TIME
1	a 1 b 1	n 1	-	-	12	20
2	n 1	n 4	-	-	2	20
3	a 2 b 2 a 3 b 3	n 5	-	n 2 n 3	27	20
4	n 4 n 5	n 6	-	-	12	20

Total sequential time = 53

(b) The new Bernstein Sets after merging

Figure 4.13: Merging of stanzas of the program in figure 4.6

Figure 4.18 shows a new set of stanzas produced after the first merging operation. Note that stanzas 1 to 8 are the same as previous stanzas and stanza 11 is the same as stanza 15. Stanza 9 and 10 are the new merged stanzas. Stanza 9 in this table comes from stanzas 9, 10 and 13 and stanza 10 from 11, 12 and 14. The new Dependence Table and the Contemporary Table after the first merging process are shown in figure 4.19.

In the second merging operation, stanzas 9, 10 and 11 are merged. Figure 4.20 gives an illustration of their merging. The new set of stanzas after the second merging is shown in figure 4.21. In figure 4.22 are the revised Dependence Table and Contemporary Table after the second merging.

As a conclusion, TAG arrives at the final result after performing two merging operations. The third merging does not produce any new stanzas and hence it stops. Table 4.1 shows the performances produced by the schedules generated before and after the two merging operations. It indicates that the best execution time of 2.91 speed-up value comes with a schedule for a 8-processor parallel machine after the second merging operation (column 4 row 7). The average granularity size is 11.33 which has nearly doubled from the original size of 6.80. Figure 4.23 shows the grains of stanzas that give the best schedule.

The performances shown in Table 4.1 are for the program that has a communication time of 10 units. The diagrams in figure 4.24 show the effects of a higher communication time on the merging operation. In this case, 20 units is fixed for the communication time and it results in different groups of stanzas being merged.

In the first merging operation, four groups of stanzas are merged. If this is compared with the diagrams in figures 4.17 and 4.20, different groups are being merged. In the second merging operation, only one group is merged. Table 4.2 shows its performances where the best schedule has a speed-up value of 2.04 on a 4-processor machine. Figure 4.25 illustrates the grains

of stanzas for this particular schedule. The speed-up is lower than that with 10 unit of communication time (see figure 4.23(b)) and it is achieved after the second merging. However, the average stanza size is about three times bigger than the original size (that is 20.40).

number of processors	before merging	first merging	second merging
2	1.65	1.65	1.50
3	2.00	2.00	1.79
4	2.12	2.55	2.22
5	2.17	2.55	2.22
6	2.55	2.55	2.22
7	2.17	2.62	2.83
8	2.17	2.62	2.91
average granularity	6.80	9.27	11.33

Table 4.1: The performances of example 4.6.1 with 10 units of communication time

number of processors	before merging	first merging	second merging
2	1.42	1.42	1.38
3	1.50	1.50	1.89
4	1.50	1.50	2.04
5	1.32		
6	1.73		
7	1.32		
8	1.32		
average granularity	6.80	14.57	20.40

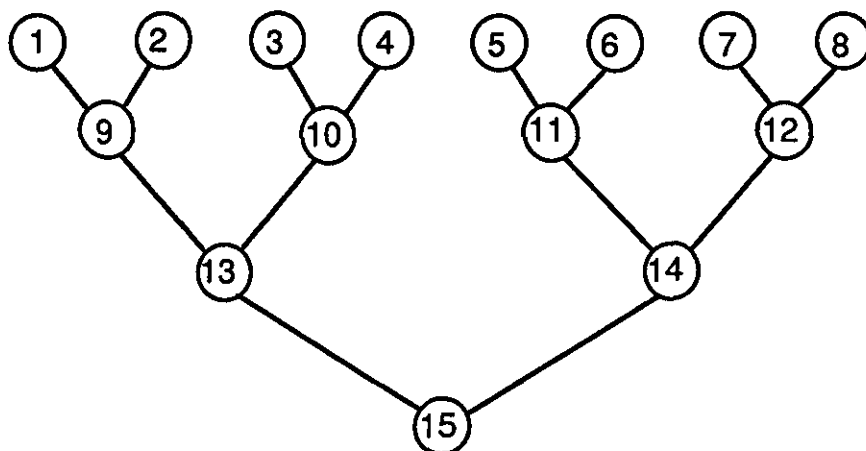
Table 4.2: Performances of example 4.6.1 with 20 units of communication time

```

program example_4_6_1;
begin
  n1 := a1*b1;
  n2 := a2*b2;
  n3 := a3*b3;
  n4 := a2*b4;
  n5 := a5*b5;
  n6 := a6*b6;
  n7 := a7*b7;
  n8 := a8*b8;
  n9 := n1+n2;
  n10 := n3+n4;
  n11 := n5+n6;
  n12 := n7+n8;
  n13 := n9+n10;
  n14 := n11+n12;
  n15 := n13+n14;
end.

```

(a) A simple program



(b) A task graph

Figure 4.14: Example 4.6.1 and its task graph

STANZA	W sets	X sets	Y sets	Z sets	EXEC TIME	COMM TIME
1	a 1 b 1	n 1	-	-	1 1	1 0
2	a 2 b 2	n 2	-	-	1 1	1 0
3	a 3 b 3	n 3	-	-	1 1	1 0
4	a 2 b 4	n 4	-	-	1 1	1 0
5	a 5 b 5	n 5	-	-	1 1	1 0
6	a 6 b 6	n 6	-	-	1 1	1 0
7	a 7 b 7	n 7	-	-	1 1	1 0
8	a 8 b 8	n 8	-	-	1 1	1 0
9	n 1 n 2	n 9	-	-	2	1 0
10	n 3 n 4	n 10	-	-	2	1 0
11	n 5 n 6	n 11	-	-	2	1 0
12	n 7 n 8	n 12	-	-	2	1 0
13	n 9 n 10	n 13	-	-	2	1 0
14	n 11 n 12	n 14	-	-	2	1 0
15	n 13 n 14	n 15	-	-	2	1 0
Total sequential time =					102	

Figure 4.15: A set of stanzas produced by the Analyser

Stanza no. - { Predecessor stanzas }

1	- Ø
2	- Ø
3	- Ø
4	- Ø
5	- Ø
6	- Ø
7	- Ø
8	- Ø
9	- { 1 2 }
10	- { 3 4 }
11	- { 5 6 }
12	- { 7 8 }
13	- { 9 10 }
14	- { 11 12 }
15	- { 13 14 }

(a) Dependence Table

stanza - { concurrent stanzas }

1	- { 2 3 4 5 6 7 8 13 14 }
2	- { 3 4 5 6 7 8 13 14 }
3	- { 4 5 6 7 8 9 14 }
4	- { 5 6 7 8 9 14 }
5	- { 6 7 8 9 10 14 }
6	- { 7 8 9 10 14 }
7	- { 8 9 10 11 15 }
8	- { 9 10 11 15 }
9	- { 10 11 12 15 }
10	- { 11 12 15 }
11	- { 12 13 }
12	- { 13 }
13	- { 14 }
14	- Ø
15	- Ø

(b) Contemporary Table

Figure 4.16: The Dependence Table and the Contemporary Table generated by the Detector module

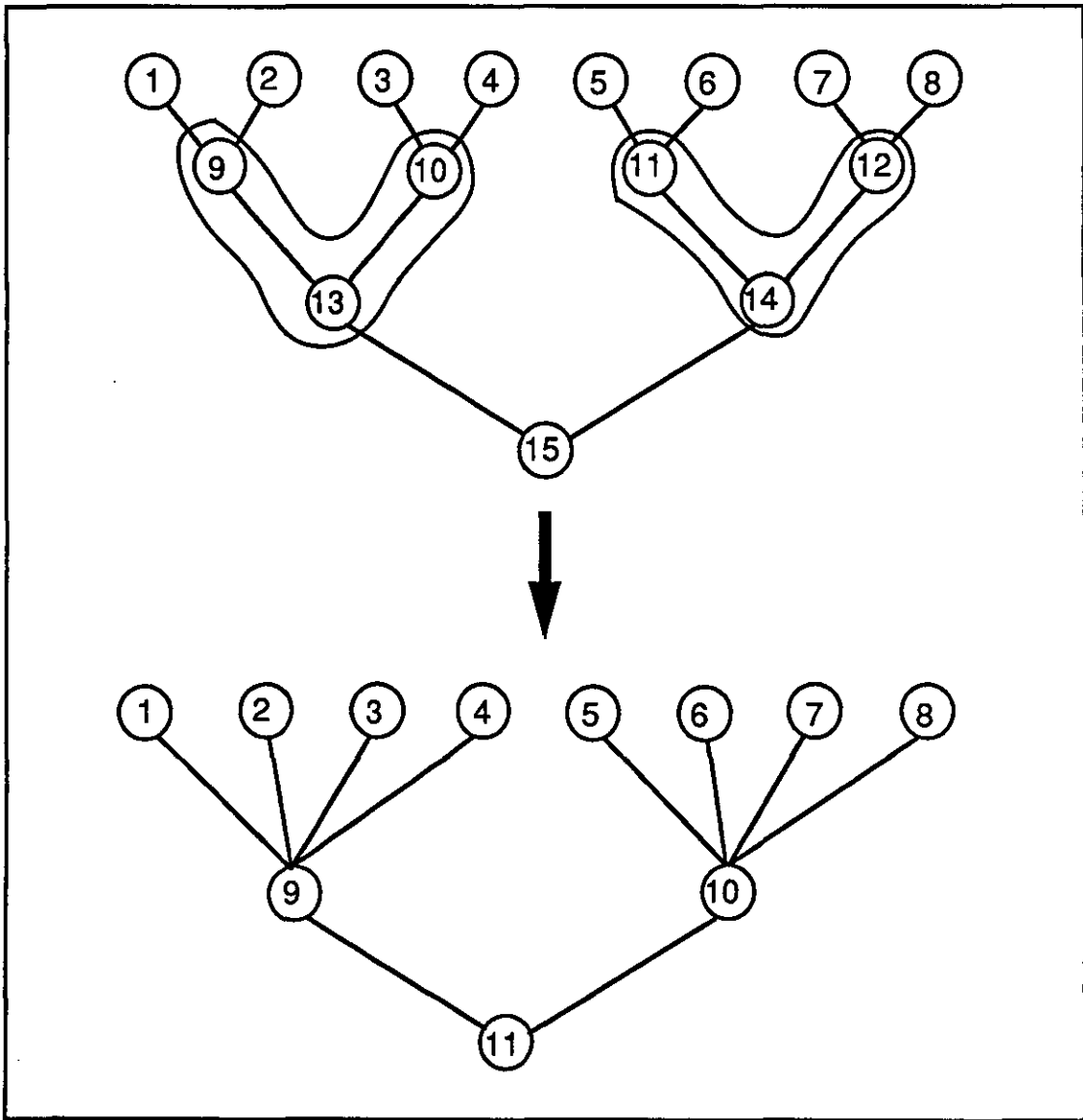


Figure 4.17: The first merging process of example 4.6.1 (communication time = 10 units)

STANZA	W sets	X sets	Y sets	Z sets	EXEC TIME	COMM TIME
1	a 1 b 1	n 1	-	-	1 1	1 0
2	a 2 b 2	n 2	-	-	1 1	1 0
3	a 3 b 3	n 3	-	-	1 1	1 0
4	a 2 b 4	n 4	-	-	1 1	1 0
5	a 5 b 5	n 5	-	-	1 1	1 0
6	a 6 b 6	n 6	-	-	1 1	1 0
7	a 7 b 7	n 7	-	-	1 1	1 0
8	a 8 b 8	n 8	-	-	1 1	1 0
9	n 1 n 2 n 3 n 4	n 1 3	-	n 9 n 1 0	6	1 0
10	n 5 n 6 n 7 n 8	n 1 4	-	n 1 1 n 1 2	6	1 0
11	n 1 3 n 1 4	n 1 5	-	-	2	1 0
Total sequential time =					102	

Figure 4.18: A new set of stanzas after the first merging

Stanza no. - { Predecessor stanzas }

1	- Ø
2	- Ø
3	- Ø
4	- Ø
5	- Ø
6	- Ø
7	- Ø
8	- Ø
9	- { 1 2 3 4 }
10	- { 5 6 7 8 }
11	- { 9 10 }

(a) Dependence Table

stanza - { concurrent stanzas }

1	- { 2 3 4 5 6 7 8 11 }
2	- { 3 4 5 6 7 8 11 }
3	- { 4 5 6 7 8 11 }
4	- { 5 6 7 8 11 }
5	- { 6 7 8 9 }
6	- { 7 8 9 }
7	- { 8 9 }
8	- { 9 }
9	- { 10 }
10	- Ø
11	- Ø

(b) Contemporary Table

Figure 4.19: The new Dependence Table and the Contemporary Table after the first merging

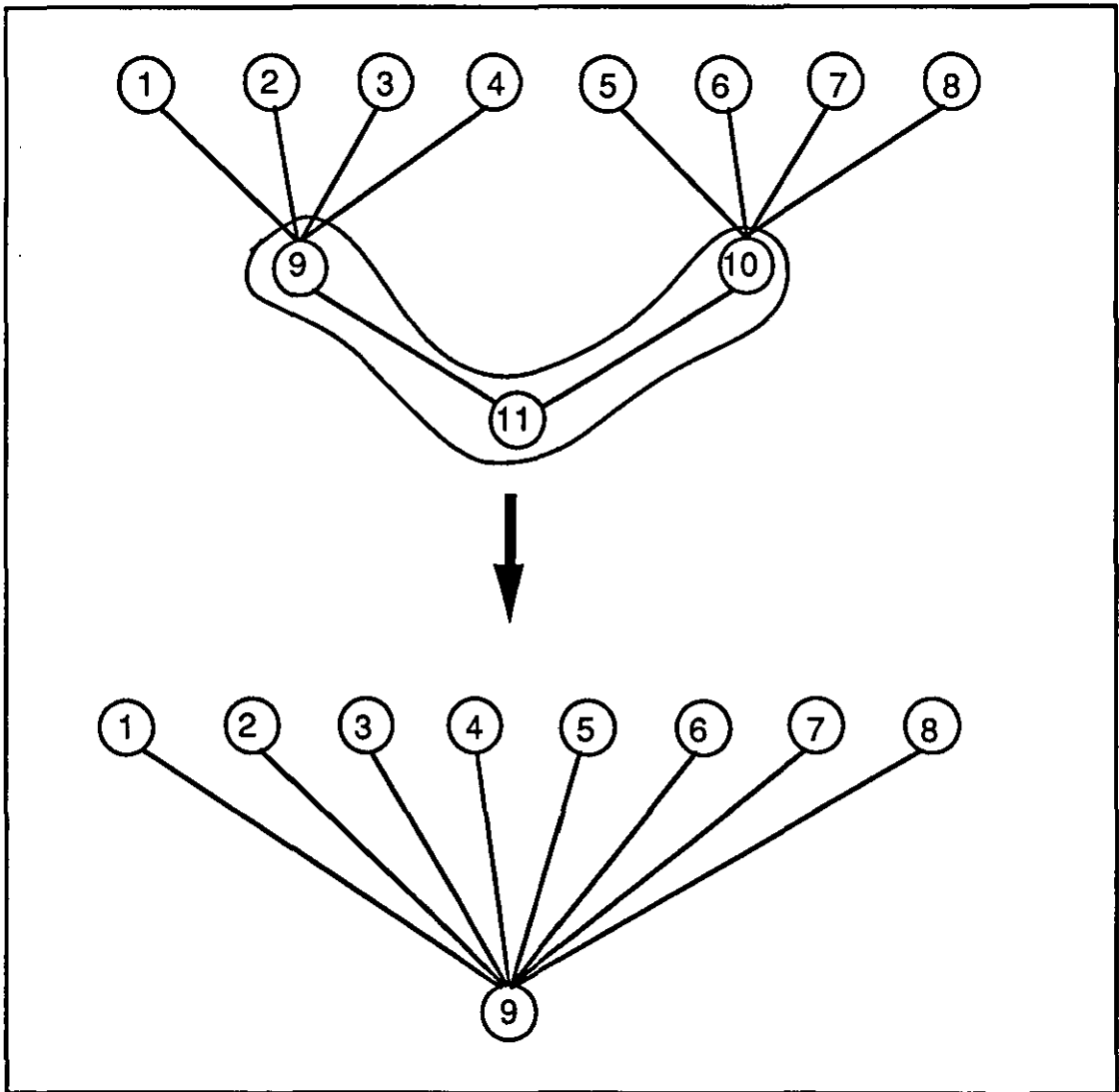


Figure 4.20: Second merging operation of example 4.6.1 (communication time = 10 units)

STANZA	W sets	X sets	Y sets	Z sets	EXEC TIME	COMM TIME
1	a 1 b 1	n 1	-	-	1 1	1 0
2	a 2 b 2	n 2	-	-	1 1	1 0
3	a 3 b 3	n 3	-	-	1 1	1 0
4	a 2 b 4	n 4	-	-	1 1	1 0
5	a 5 b 5	n 5	-	-	1 1	1 0
6	a 6 b 6	n 6	-	-	1 1	1 0
7	a 7 b 7	n 7	-	-	1 1	1 0
8	a 8 b 8	n 8	-	-	1 1	1 0
9	n 1 n 2 n 3 n 4 n 5 n 6 n 7 n 8	n 15	-	n 13 n 14 n 9 n 10 n 11 n 12	14	1 0
Total sequential time =					102	

Figure 4.21: The new set of stanzas after second merging

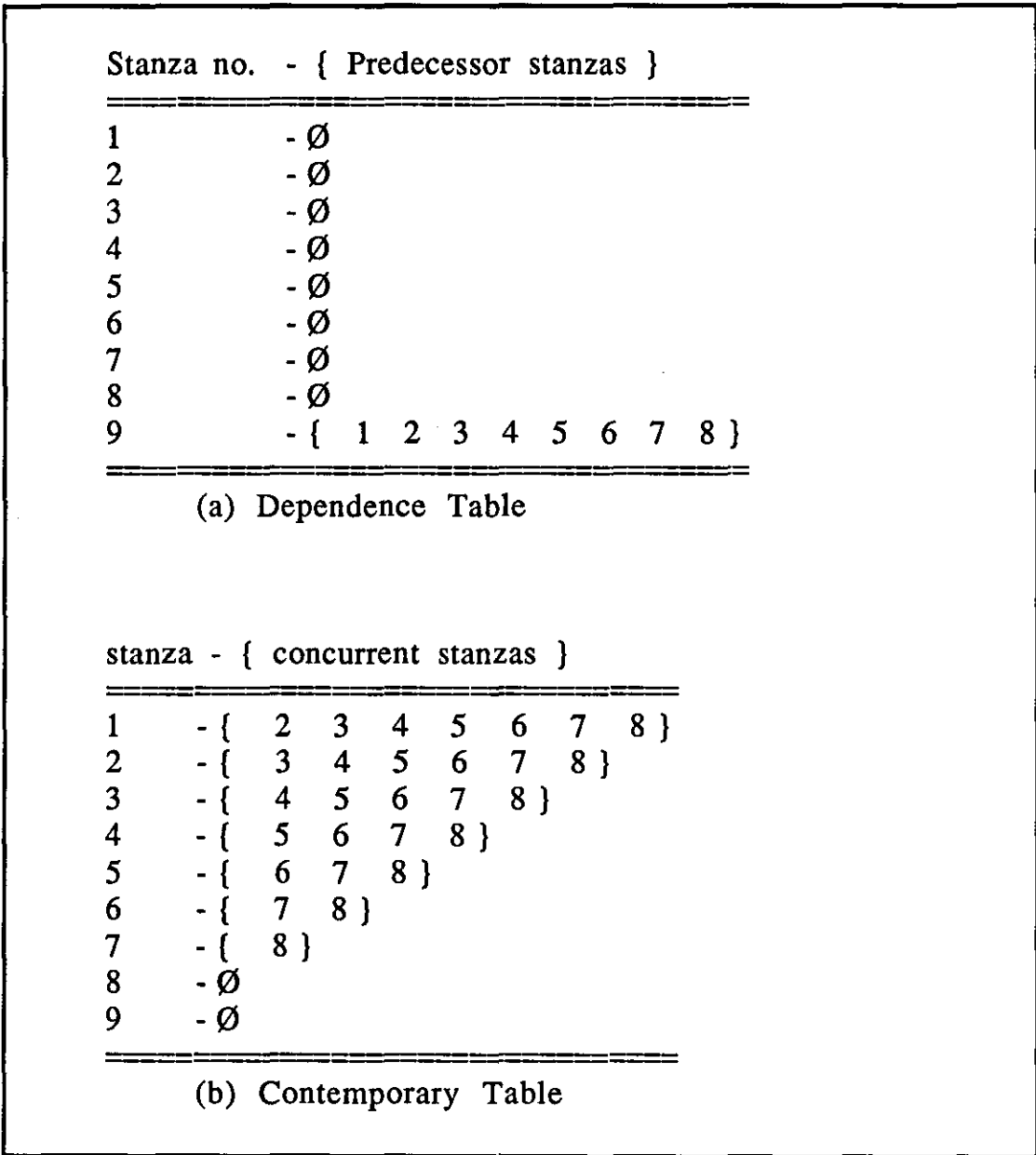
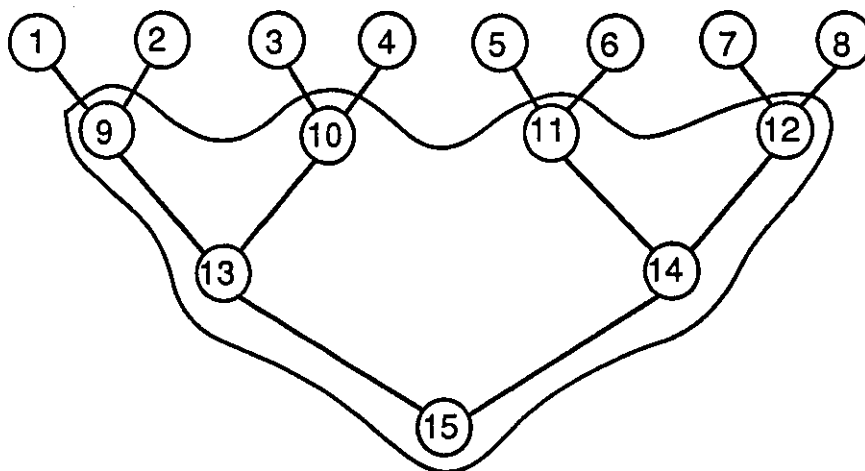
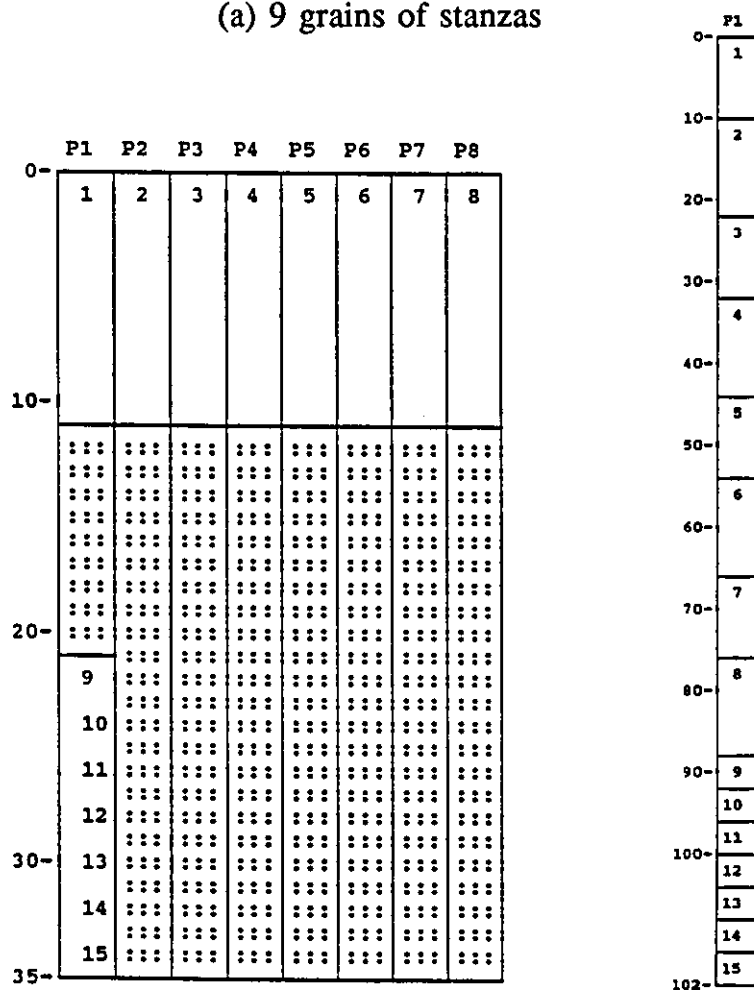


Figure 4.22: The new Dependence Table and Contemporary Table after the second merging



(a) 9 grains of stanzas



(b) The schedule for the grains (speed-up = 2.91)

Figure 4.23: The schedules for example 4.6.1

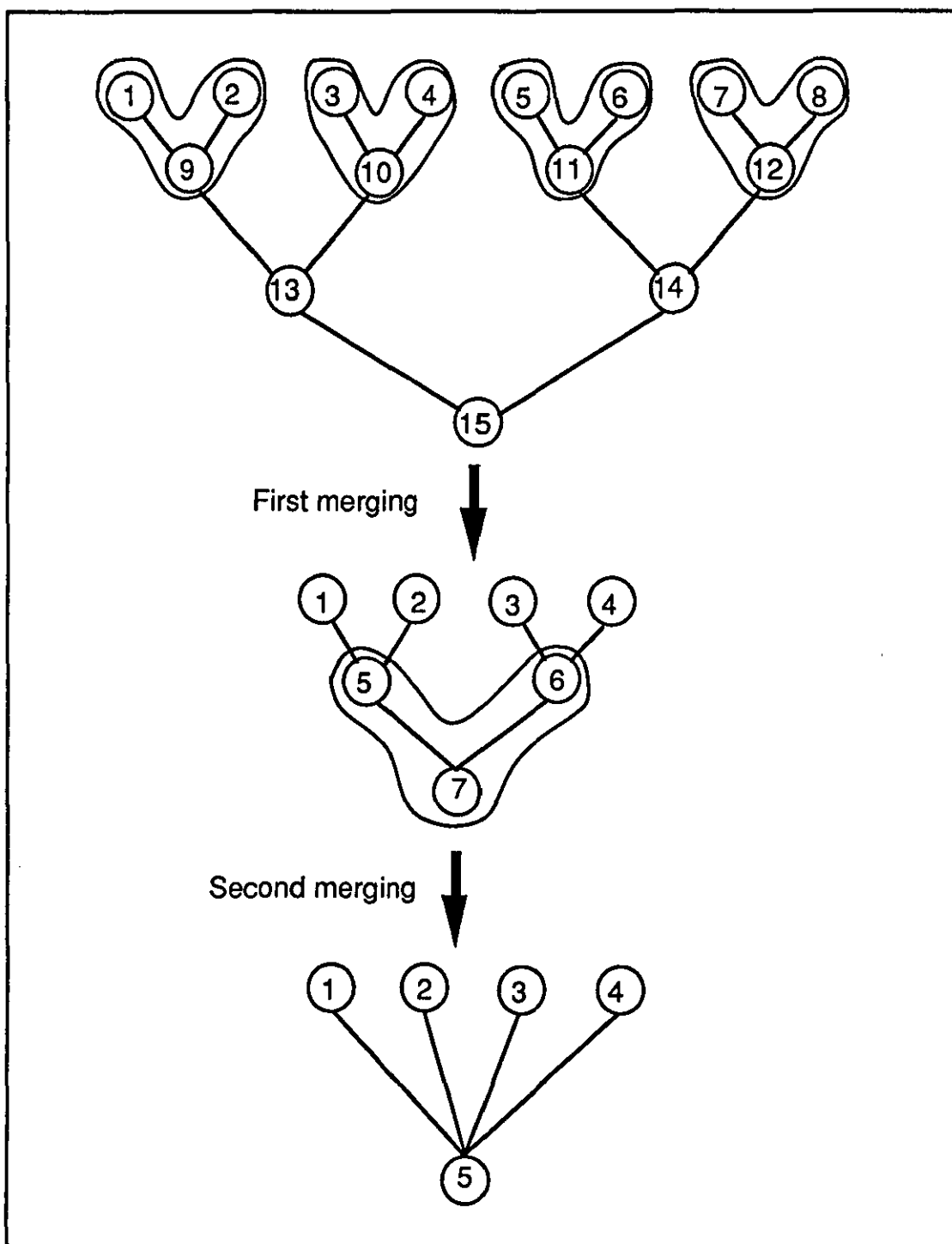
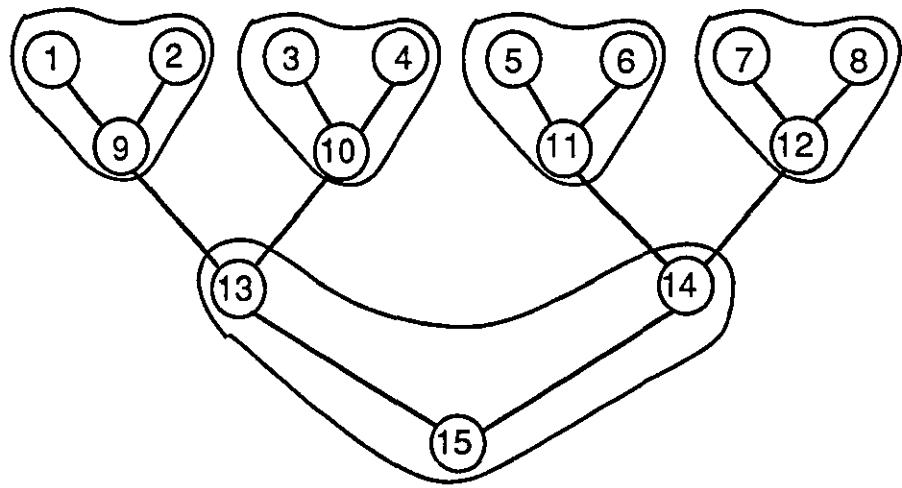
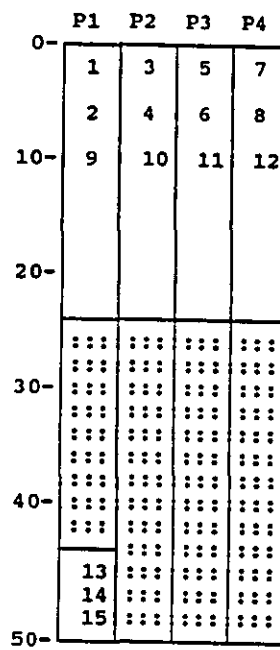


Figure 4.24: The merging of stanzas from example 4.6.1 with 20 units of communication time



(a) grains of stanzas



(b) the best schedule (speed-up = 2.04)

Figure 4.25: The grains of stanzas for example 4.6.1 with the communication time = 20 units

EXAMPLE 4.6.2:

In the following example, as shown in figure 4.26, the program has a different dependence structure. Figure 4.27 shows a series of merging operations performed on the stanzas of the program. Table 4.3 shows its performances obtained by TAG.

number of procs.	before merging	first merging	second merging	third merging	fourth merging
2	0.94	0.95	1.07	1.29	1.23
3	1.23	1.23	1.23	1.29	
4	1.14	1.16	1.23		
5	1.14	1.21			
6	1.14				
average gran.	5.27	6.44	8.29	11.60	19.33

Table 4.3: Performances of example 4.6.2 with 20 units of communication time

The above table shows that the execution times can be worse than that of the sequential run (see row with 2 processors). These are the effects of the high communication time of 20 units. However, after the merging operation, it begins to improve. The best time is achieved on 2 and 3-processor machines after the third merging (speed-up factor of 1.29). For the 2-processor machine, after the fourth merging, however, the performance degrades to 1.23. This is due to the less parallelism that exists after the merging and the merged stanzas have more predecessors that need to be considered in the scheduling. Figure 4.28 shows the 2-processor schedule generated for the groups of stanzas that gives the near-optimal solution.

```

program par_4_6_2;
begin
    n1 := a1-b1;
    n2 := a2*b2;
    n3 := a3*b3;
    n4 := 10;
    n5 := a5-10;
    n6 := a6+b6;
    n7 := n5+n6;
    n8 := n4*n7;
    n9 := n8+n3-10;
    n10 := n9+n2;
    n11 := n10/n1;
end.

```

STANZA	W sets	X sets	Y sets	Z sets	EXEC TIME	COMM TIME
1	a 1 b 1	n 1	-	-	2	20
2	a 2 b 2	n 2	-	-	11	20
3	a 3 b 3	n 3	-	-	11	20
4	-	n 4	-	-	1	20
5	a 5	n 5	-	-	2	20
6	a 6 b 6	n 6	-	-	2	20
7	n 5 n 6	n 7	-	-	2	20
8	n 4 n 7	n 8	-	-	11	20
9	n 8 n 3	n 9	-	-	3	20
10	n 9 n 2	n 10	-	-	2	20
11	n 10 n 1	n 11	-	-	11	20
Total sequential time =					58	

Figure 4.26: The second example 4.6.2

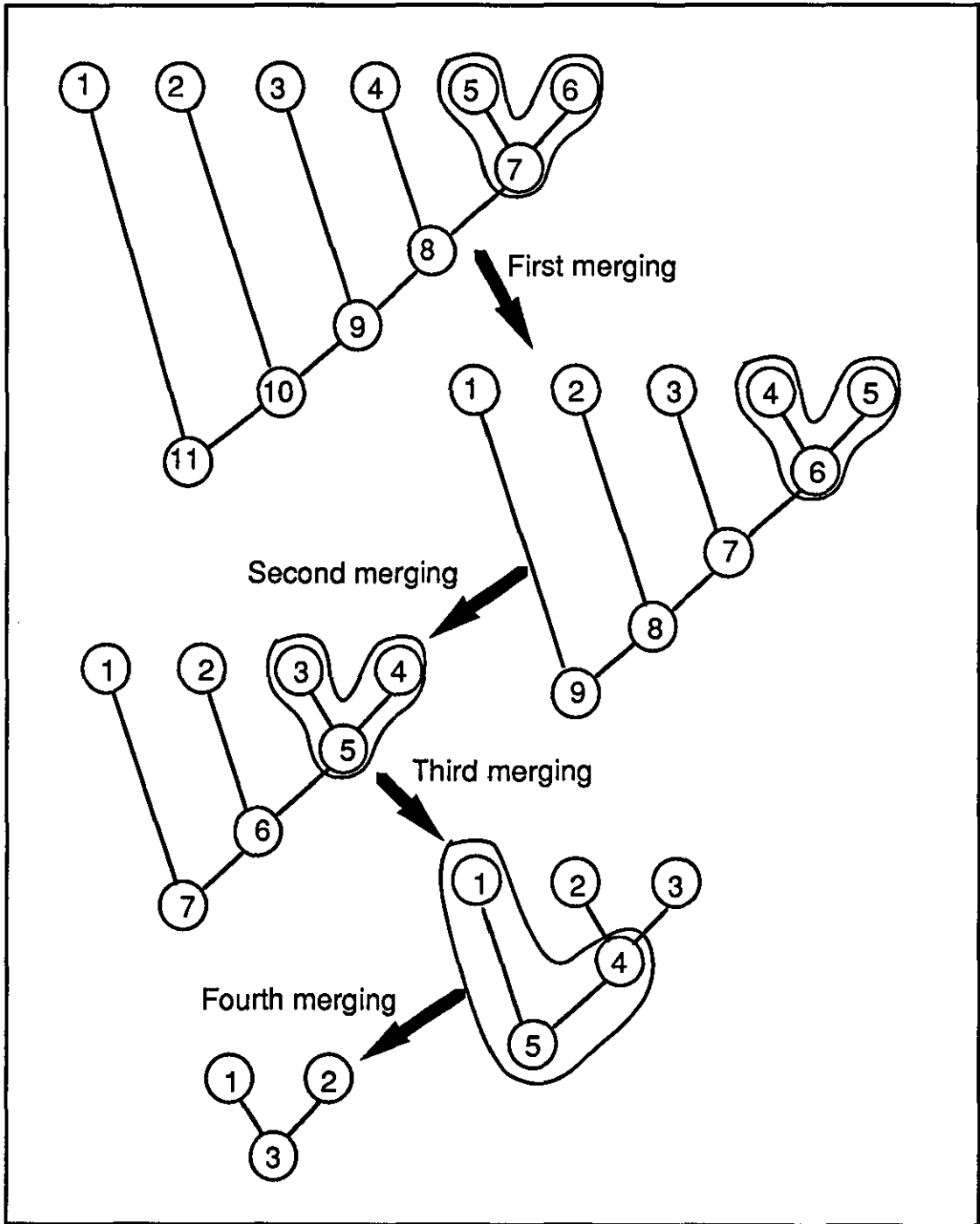


Figure 4.27: The merging sequence of example 4.6.2 (communication time = 20 units)

EXAMPLE 4.6.3:

The third example, adapted from [Kruatrachue and Lewis (1988)] is shown in figure 4.29. Table 4.4 shows the speed-up factors for the program. Figures 4.30, 4.31 and 4.32 show the first and second merging operations and the best schedule generated respectively.

number of processors	before merging	first merging	second merging
2	1.19	1.20	0.98
3	1.25	1.25	1.21
4	1.16	1.17	
5	1.16		
6	1.17		
average granularity	8.12	10.62	12.55

Table 4.4: Performances of example 4.6.3 with 20 units of communication time

This example has a complex dependence structure. Merging of one group of stanzas may alter the dependences of other stanzas. However, TAG manages to determine the speed-up value of 1.25 as its best solution.


```
program par_4_6_3;  
begin  
  a := 1;  
  c := 3;  
  b := 2*5;  
  d := 4;  
  e := 5-3;  
  f := 6;  
  g := a*b;  
  h := c*d;  
  i := d*e;  
  j := e*f;  
  k := d*f;  
  l := j*k;  
  m := 4*l;  
  n := 3*m;  
  o := n*i;  
  p := o*h;  
  q := p*g;  
end.
```

Figure 4.29: The third example 4.6.3

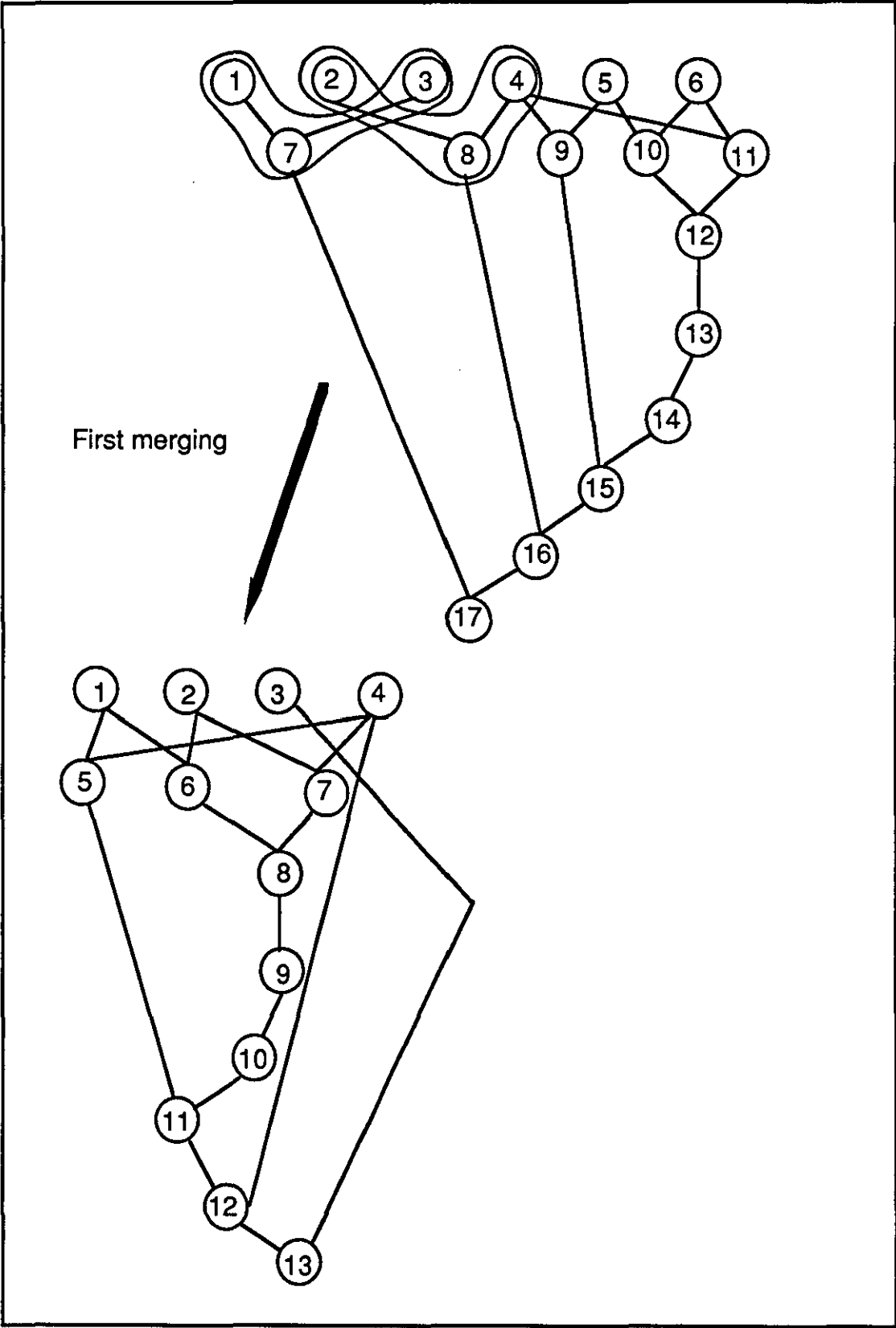


Figure 4.30: First merging of the third example (comm = 20)

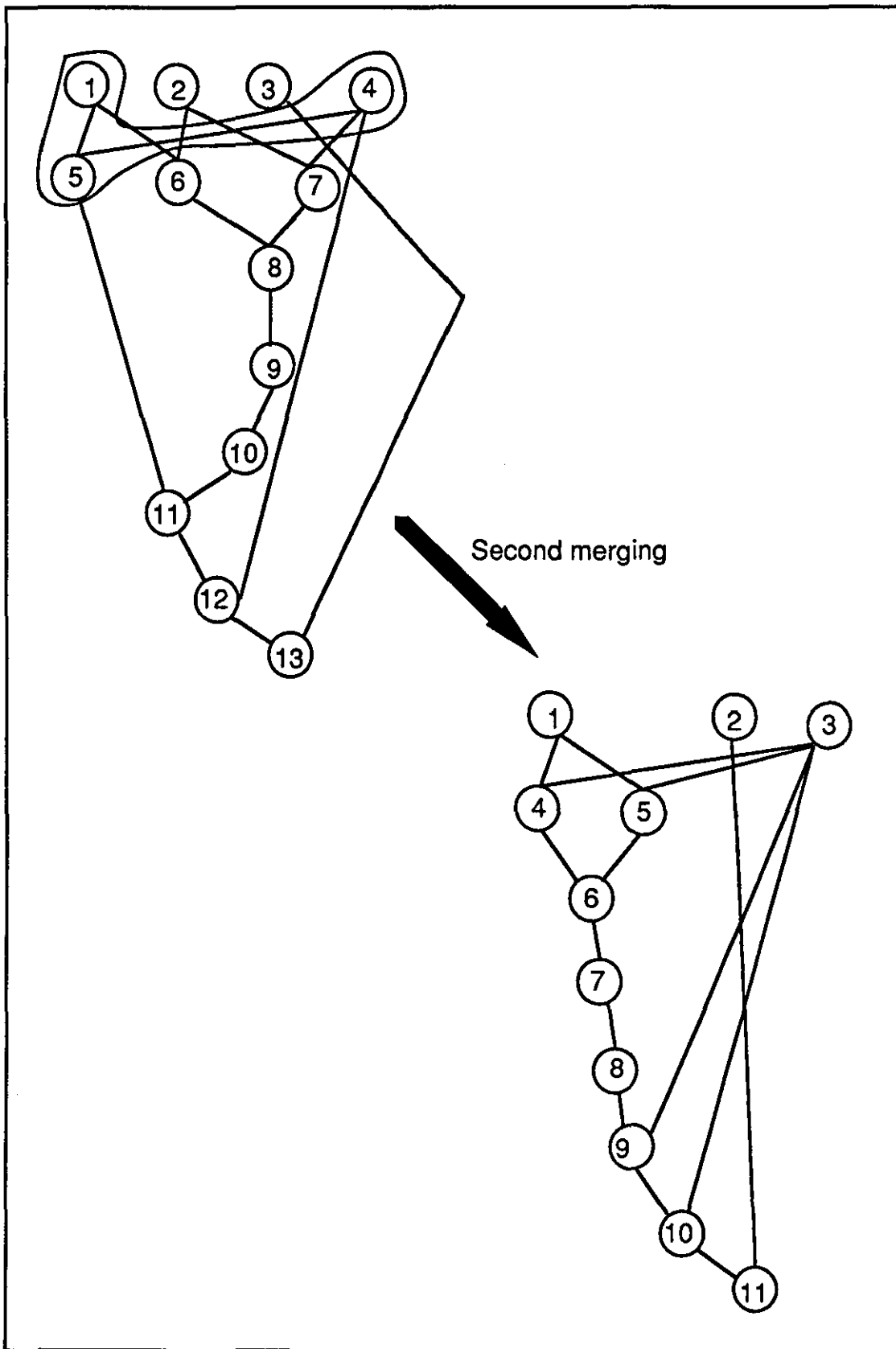


Figure 4.31: Second merging of the third example (comm = 20)

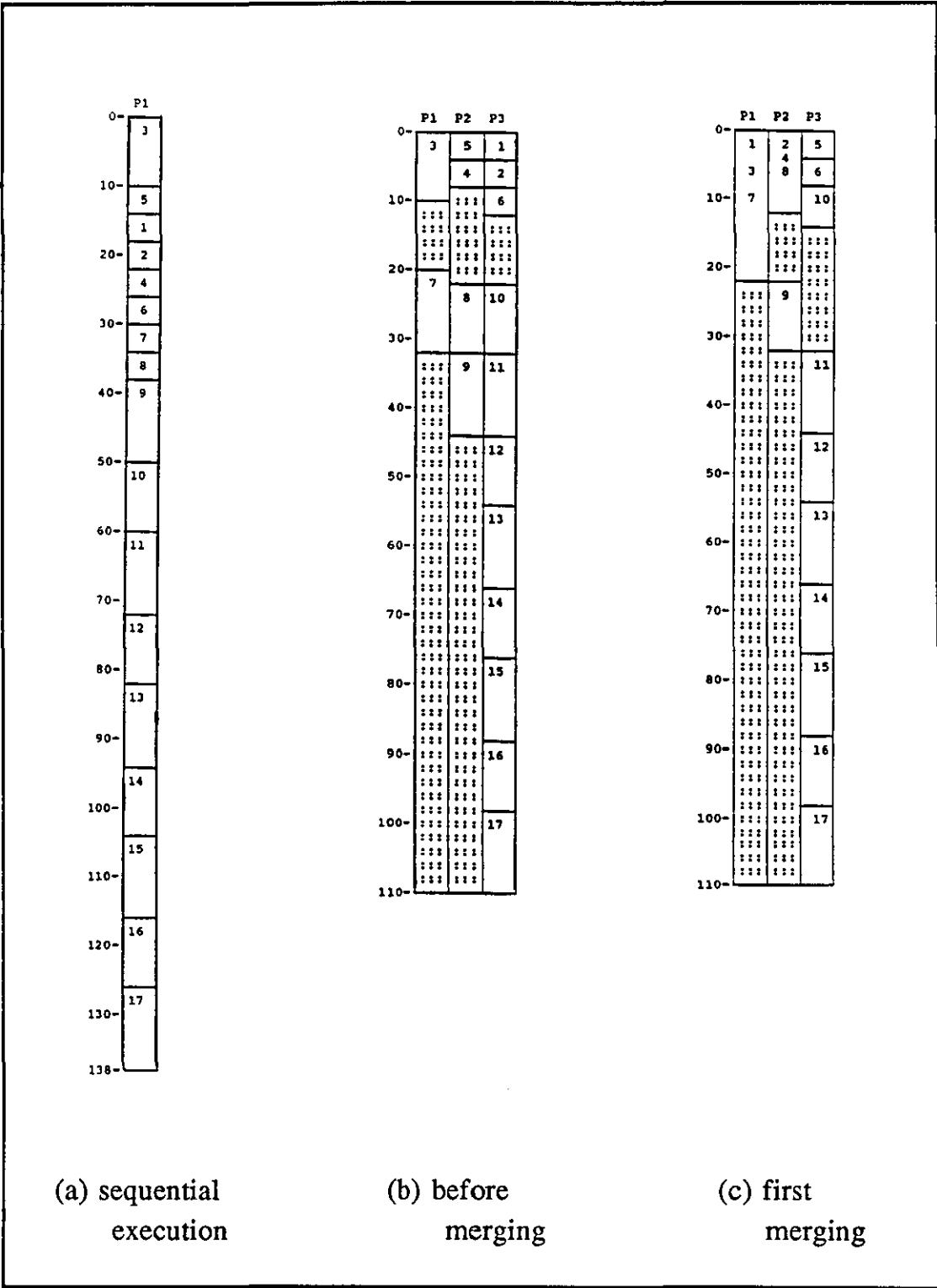


Figure 4.32: The schedules for the third example

4.7 SCHEDULING OF PARALLEL LOOPS

To parallelize a sequential program, the best opportunities can be found in the loops [Allen and Kennedy (1984a), Banerjee (1988), Li and Yew (1990), Mohd-Saman and Evans (1993), Padua and Wolfe (1986), Wolfe (1989b), Wolfe and Banerjee (1987)]. The loop iterations can be executed concurrently if they are independent. Data dependences are usually caused by references to array elements and scalar variables which are modified by more than one iteration. Determination of data dependences in loops has been a major research work. Mohd-Saman and Evans (1993) have developed the Bernstein Loop Tests (BLTs) to determine loop parallelism. Non-independent loops can only be parallelized after their dependences have been removed by loop transformations or by inserting synchronization statements. The BLTs and the loop transformation techniques are discussed in Chapter 5 of this thesis.

Scheduling of loops on multi-processor systems has been discussed by several researchers [Beckman and Polychronopoulos (1991), Foster (1991), Polychronopoulos (1988), Saltz et al. (1991)]. Loop iterations can be scheduled statically or dynamically. For static scheduling, the user or the compiler decides which iterations are to be allocated to a given processor. This eliminates the run-time overhead. However, information of the number of iterations to schedule and the number of processors are sometimes known at run-time. Furthermore, iterations can have different execution times due to the presence of conditional statements. This creates the problem of unknown sizes if they are to be assigned statically.

A general solution for this problem is to distribute the loop iterations to processors based on their availability. This can be done dynamically and three methods have been suggested [Polychronopoulos (1988)]. The first strategy is called self-scheduling where an idle processor selects a single iteration of a parallel loop for execution. This is good for load balancing

especially if the iteration is large compared to the iteration fetching overhead. The second scheme, chunk scheduling, allocates a fixed number of iterations to an idle processor. This reduces the overhead in fetching the iterations but it cannot provide a good balanced load for all of the processors. The third method combines the above two schemes. It is called guided-self scheduling. The strategy is that, at the beginning of the execution, a chunk of iterations is assigned and this is decremented in the next iteration fetching until all iterations are exhausted. It is claimed that the third scheme gives the best result in load balancing and reduces run-time overhead.

TAG can be extended to handle parallelization of loops and their scheduling. Thus, any loops in programs can be tested for parallelism and those found to be parallelizable, can be scheduled on a multi-processor system. Those loops whose iterations cannot be run in parallel will be treated as single stanzas. Thus, a granularity of program containing straight line codes and loops can easily be determined automatically.

4.8 SUMMARY

This chapter has discussed several related issues, namely, the detection of implicit parallelism in a sequential programs, the scheduling of stanzas and the determination of the best stanza granularity. In detecting any parallelism that may exist in a sequential program, the BTs have been used. A software tool called TAG has been developed to perform the above functions. The best grain size of a program is determined by merging and scheduling the stanzas repeatedly.

Stanza formation developed by Williams (1978) and Evans and Williams (1978) provides a useful way to determine potential parallelism that may exist in programs. The information available in a stanza is used in performing the dependence analysis. Most of the work done by other researchers has based their dependence analysis on the data-flow graphs [Allen and Cocke (1976), Allen and Kennedy (1987), Burke et al. (1988),

Muchnick and Jones (1981), Sarkar (1989), Wolfe (1989b), Wolfe and Banerjee (1987)].

The main goal of scheduling the stanzas of a program is to determine its shortest execution time. The study described in this chapter has shown that the communication overhead between processors due to data dependences can cause execution time degradation. Hence, the scheme that performs the scheduling should be able to assign stanzas to processors in such a way that it minimizes the overhead and maximizes the parallelism in order to obtain the optimal time. This problem of finding an optimal solution is intractable. However, near-optimal solutions using heuristics can be obtained and this has been discussed in detail in this chapter.

The process of merging of stanzas to form stanzas with bigger granularity has been used in order to find a faster execution time. This needs to be performed carefully. An improved execution time is only possible if the communication time that exists after merging is less with respect to the stanza size and the program still has adequate parallelism. This, however, is not always the case. Merging can also cause execution time degradation. Thus, the heuristic proposed in this chapter only merges stanzas if it proves to give better results.

CHAPTER 5

DETECTION OF LOOP PARALLELISM AND TRANSFORMATIONS

5.1 INTRODUCTION

The parts of a sequential program which offer the best opportunities for parallelism are the **loops**. The iterations in a loop can be parallelized if all of them are independent, i.e., there is no data dependence between them. Inter-iteration data dependences are usually caused by references by an iteration to array elements modified by other iterations. In the case of scalar variables in the loops, data dependences occur if they are involved in store operations in the different iterations.

To detect any data dependences in a loop, the statements in its body have to be analysed. There are several research works on the detection of parallelism in loops. Most of them are based on the graph theory and the Diophantine Equations [Allen and Kennedy (1987), Burke et al. (1988), Kong et al. (1991), Krothapalli and Sadayappan (1991), Li (1989), Li et al. (1989), Saltz et al. (1989), Tang et al. (1990), Wolf and Lam (1991), Wolfe (1988)]. In order to execute the iterations which have data dependences concurrently, the loops have to be modified to eliminate any dependences that exist. In cases where data dependences cannot be removed, synchronization statements are inserted in the loops [Midkiff and Padua (1986)].

In this chapter, a technique to detect parallelism in loops is described. It is based on the Bernstein Sets (BSs) and the Bernstein Tests (BTs) [Bernstein (1966), Evans and Williams (1978), Williams (1978)]. This technique, called the **Bernstein Loop Tests (BLTs)**, is well suited for loops containing array variables as well as scalar variables. The notations called the **Data Reference Directions (DRDs)** are used for the array variables to indicate how they are being referenced by the loop body in the different iterations. These directions are formulated so that the dependence tests performed on the statements in the loop body will derive results that will show whether the loop iterations are parallelizable or not. This technique is discussed from Section 5.2 to Section 5.5.

This chapter also discusses the various transformation techniques that can be applied to loops in order to execute them in parallel. Before any kind of transformation is performed, extensive analysis on the loop body has to be carried out in the DDA. The information provided by the BSs and the results of the BLTs will be used in making the decision on how the loops are best transformed. The techniques discussed in this chapter are well-known techniques in compiler writing and parallelization of programs [Aho et al. (1986), Padua and Wolfe (1986), Polychronopoulos (1988), Wolfe (1989b), Zima and Chapman (1990)]. They will be discussed from Section 5.6 to Section 5.8. In Section 5.9, related issues concerning loop dependences are discussed while Section 5.10 summarises the whole chapter.

5.2 PARALLELISM IN LOOPS

The example in figure 5.1 shows a loop with three statements and their corresponding BSs. By inspecting the stanzas of each iteration shown in figure 5.1(c), they show that there are data dependences caused by variables a, b and d. These variables are involved in store operations. This is clearly indicated by the contents of the BSs of the loop body as in figure 5.1(b) where a and b are the variables in the X set and variable d in the Y set. Thus, the types of BSs can indicate the data dependences. They are X, Y and Z sets that contain variables involved in store operations.

On the other hand, loop iterations which contain array variables can be executed in parallel if there are no conflicting fetch/store operations to the same array variables with two or more different indices. This means, there must be no multiple fetch and store operations on the same array elements by the different iterations. As an example, for the loop in figure 5.2(a), each iteration will be making references to array elements a, b and c without having to cross the boundary as figure 5.2(b) shows. Thus, the iterations can be executed in parallel. This is called **loop-independent dependence** [Allen and Kennedy (1987), Padua and Wolfe (1986), Wolfe (1989b), Wolfe and Banerjee (1987)].

```

for i := 1 to n do
begin
    a := b + c;      ... s1
    b := a - 10;     ... s2
    d := c + d;      ... s3
end;

```

(a) A loop with scalar variables

	W	X	Y	Z
s1:	c,b	a	-	-
s2:	a	b	-	-
s3:	c	-	d	-

(b) The BSs of the statements of the loop in (a)

<u>iteration 1</u>	<u>iteration 2</u>	<u>...</u>	<u>iteration n</u>
a := b + c;	a := b + c;		a := b + c;
b := a - 10;	b := a - 10;	...	b := a - 10;
d := c + d;	d := c + d;		d := c + d;

(c) n iterations with data dependences

Figure 5.1: A non-parallelizable loop due to store and fetch operations

```

for i := 1 to n do
begin
    a[i] := b[i] + c[i];
    b[i] := a[i] + k;
end;

```

(a) a loop with n iterations

<u>iteration 1</u>	<u>iteration 2</u>	<u>...</u>	<u>iteration n</u>
a[1] := b[1]+c[1];	a[2] := b[2]+c[2];	...	a[n] := b[n]+c[n];
b[1] := a[1]+k;	b[2] := a[2]+k;		b[n] := a[n]+k;

(b) n iterations which can be executed on n processors

Figure 5.2: A parallelizable loop containing array variables

```

for i := 1 to n do
begin
    a[i+1] := b[i] + c[i];    ...S1
    b[i-1] := a[i] + k;      ...S2
end;

```

(a) A loop with data dependences

<u>iteration 1</u>	<u>iteration 2</u>	<u>...</u>	<u>iteration n</u>
a[2] := b[1]+c[1];	a[3] := b[2]+c[2];	...	a[n+1] := b[n]+c[n];
b[0] := a[1]+k;	b[1] := a[2]+k;		b[n-1] := a[n]+k;

(b) n iterations with data dependences

Figure 5.3: Data dependences due to array variables

However, for a loop such as in figure 5.3(a) the iterations are not independent because data dependences cross the boundary for the array a and array b. Iteration 2 fetches the element a[2] while in iteration 1, the same element is being stored with a value. The same problem applies to array b. There are data dependences between i-th iteration and $(i \pm 1)$ -th iteration. This is shown in figure 5.3(b). The data dependences caused by the arrays a and b are called the **forward dependence** and the **backward dependence** respectively. These are also called **loop-carried dependences** [Lewis and El-Rewini (1992), Padua and Wolfe (1986), Wolfe (1989), Zima and Chapman (1990)].

As mentioned in Chapter 3, Williams (1978) has implemented a set of dependence tests called the **Bernstein Tests (BTs)** to determine whether any two stanzas are contemporary or not, that is, whether they can be executed in parallel or not. The tests can easily handle sets containing scalar variables as in the stanzas shown in figure 5.1 or individual single array elements such as a[10] (that is a single array 'a' element with a constant index value of 10).

To determine if loops can be parallelized or not, Williams considers each iteration as a stanza with individual array elements as part of the contents in the BSs. Then the BTs are applied to all stanzas to determine which iterations are parallelizable. In this technique, a lot of stanzas have to be formed, i.e., depending on the size of the lower and upper limit of the loops, as well as the tests to be applied. As an example, consider the example in figure 5.4. The loop in figure 5.4(a) will give the individual stanzas (with the loop body taken as a stanza) for each iteration as in figure 5.5, based on the general BSs as shown in figure 5.4(b). Clearly, the BTs developed by Williams will show that there are data dependences between iterations due to the array references caused by a[i+1] and a[i] and b[i-1] and b[i].

```

for i := 1 to 10 do
begin
    a[i+1] := b[i-1] + c[i];
    b[i] := a[i] + k;
    c[i] := b[i] + m;
end;

```

(a) A loop with array references

W	X	Y	Z

b[i-1]	a[i+1]	c[i]	b[i]
a[i]			
k, m			

(b) The general BSs for loop body in (a)

Figure 5.4: Williams' general iteration stanzas

For the loop tests developed in this chapter, instead of forming a number of stanzas for each iteration, the BSs are formed either for the whole loop body (i.e., one stanza only) or for each statement in the loop body (i.e., many stanzas). These BSs will contain array variables with Data Reference Directions (DRDs), denoting how they are being referenced. A set of tests called the **Bernstein Loop Tests (BLTs)** will handle specifically these BSs to detect any data dependences caused by the array variables. No modification is needed for scalar variables.

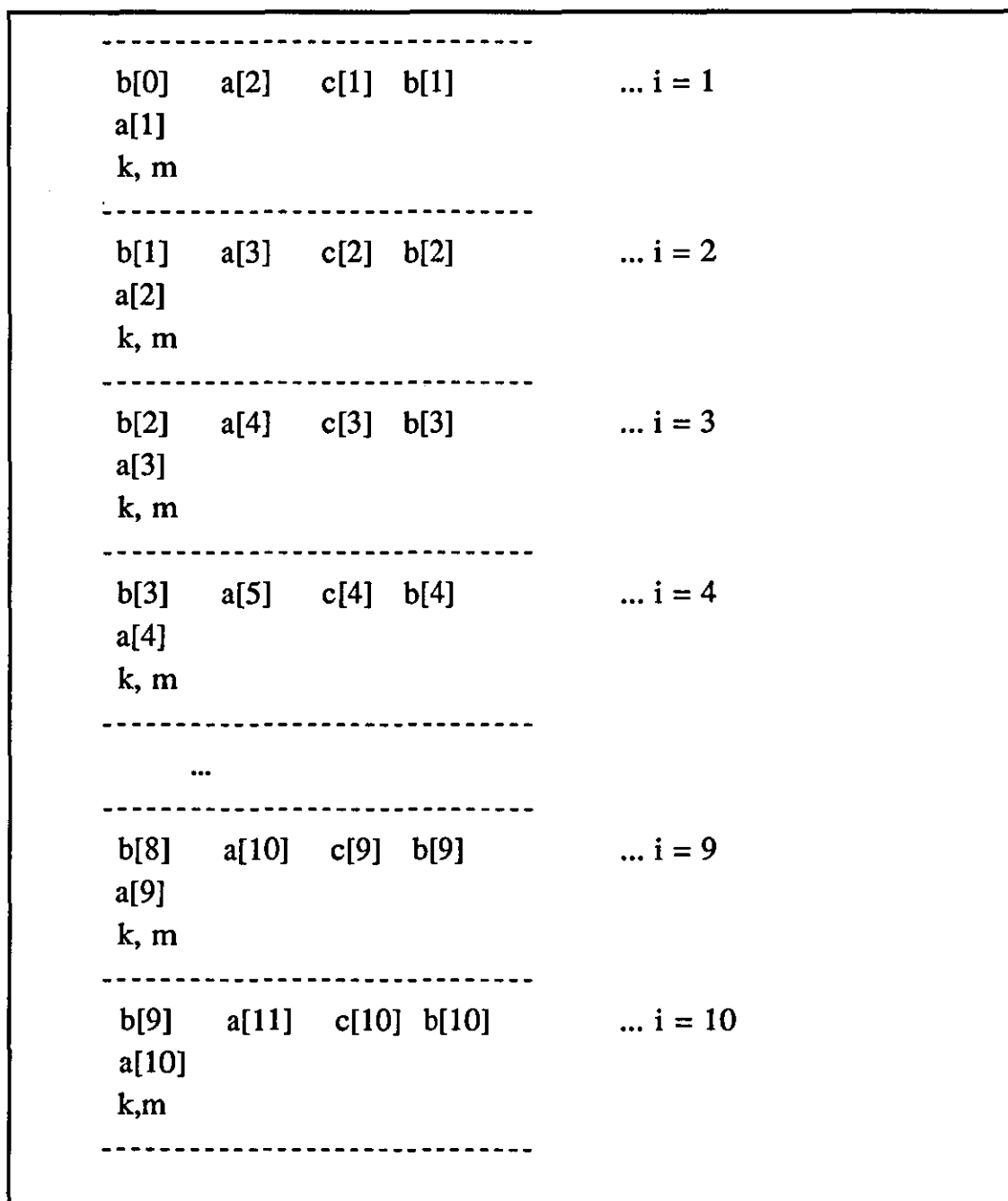


Figure 5.5: Williams' individual iteration BSs for the loop in figure 5.4.

5.3 DATA REFERENCE DIRECTIONS

Allen and Kennedy (1987), Allen et al. (1987), Wolfe (1989), Wolfe and Banerjee (1987) and Zima and Chapman (1990) have used the **data dependence directions** for specifying how iterations in loops are data dependent-related. They define the notations $<$, $>$ and $=$ to denote these directions. Forward direction ' $<$ ' denotes a dependence which crosses an iteration boundary forward (i.e., from iteration i to iteration $i+1$). Backward direction ' $>$ ' denotes a dependence which crosses an iteration boundary backward (i.e., from iteration i to iteration $i-1$). Equal direction ' $=$ ' denotes dependence which does not cross an iteration.

The same notations will be used in this chapter to enable the loop iterations to be tested for data dependence. However, here they denote how array elements are being referenced with respect to the i -th iteration. They will be called the **Data Reference Directions (DRDs)**. The symbols $<$, $>$ and $=$ denote **Forward Reference**, **Backward Reference** and **Equal Reference** respectively. Associated with these reference directions and the results of the loop tests are the **distances of the directions** and the **distances of dependences**. They are defined below.

DEFINITIONS 5.1

Let A to be an array variable, i to be a loop index and $const$ to be a constant. Data Reference Directions (DRDs) are:

- a. a Forward Reference ($<$) which is a reference to an array variable of the form $A[i+const]$ with respect to i -th iteration.*
- b. a Backward Reference ($>$) which is a reference to an array variable of the form $A[i-const]$.*
- c. an Equal Reference ($=$) which has an array reference of the form $A[i]$*

A study by Shen et al. (1989) shows that over 75% of array subscripts found in library packages are of the form $[i \pm \text{const}]$. Hence, the subscript expressions defined above are adequate enough to be used in extraction of some parallelism in programs. Loops containing array variables with expressions other than those defined above will be assumed to be non-parallelizable.

DEFINITIONS 5.2

- (i) *For a subscript $[i \pm \text{const}]$, the distance of direction is the value of const. This applies to forward and backward references.*
- (ii) *A distance of dependence is the difference between two iterations in accessing the array location. Let $d1$ be the distance of direction of one reference and $d2$ be the other distance. Then the distance of dependence is $|d1 - d2|$.*

These DRDs and the distances of directions will be augmented with the array names in the BSs. The notation " $a[>1]$ " denotes an array variable a of one dimension with a backward reference direction of distance -1 from the i -th iteration. An Equal Reference Direction has a zero distance. The single-statement stanzas for the example in figure 5.3 will give the BSs with DRDs as shown in figure 5.6. The loop body in figure 5.4(a) has a stanza with the corresponding BSs with DRDs as shown in figure 5.7. A stanza of the following form:

$a[i] := a[i] + b[i-1] + c[i+3];$

will have the following BSs with directions.

W	X	Y	Z

$c[<3], b[>1]$	-	$a[=]$	-

	W	X	Y	Z

S1:	b[=]	a[<1]	-	-
	c[=]	-	-	-

S2:	a[=]	b[>1]	-	-
	k	-	-	-

Figure 5.6: BSs for the loop in figure 5.3

W	X	Y	Z

b[>1]	a[<1]	c[=]	b[=]
a[=]			
k,m			

Figure 5.7: BSs with DRDs and direction distances for the loop in figure 5.4(a)

To test for the data dependences between the iterations, the BLTs will be applying a slightly different \cap (AND) operation. Since the array variables in the BSs have reference directions augmented to them, the \cap operations will be based on the intersection operations as given in figure 5.8. A new symbol "*" is introduced to represent "<>" as a single character.

5.4 THE BERNSTEIN LOOP TESTS (BLTS)

As mentioned earlier, the BTs implemented by Williams can be applied to stanzas with scalar variables or specific individual single array elements (such as `a[10]` in Pascal). To detect any data dependences involving general array references, the tests are extended to enable them to handle variables with DRDs. These tests will be called the Bernstein Loop Tests (BLTs). The BLTs will be derived to handle two different cases: fetch/store dependence and store dependence. They will handle all forms of scalar and array references. In the following discussion, a stanza is derived from statements in the loop body. Let there be n stanzas S_i where $1 \leq i \leq n$, each containing W , X , Y and Z sets. Let WYZ be $(W \cup Y \cup Z)$ and XYZ be $(X \cup Y \cup Z)$.

\cap operation	=	>	<
=	=	>	<
>	>	>	*
<	<	*	<

Figure 5.8: AND ' \cap ' operations for DRDs

CASE 1: Fetch-Store Dependence

There must be no dependences between a set of memory locations between the stanzas in the loop body that are fetched during execution of the stanza S_i and those that are stored during the execution of stanza S_j and vice versa. This means that, the following condition:

$$\text{BLT}_1: \quad \text{WYZ}_i \cap \text{XYZ}_j \\ \text{where } 1 \leq i, j \leq n, n = \text{number of stanzas}$$

must be \emptyset . If array variables are involved, all DRDs resulted from the tests must be of Equal Reference Direction or of Forward/Backward Reference Directions with zero dependence distances. This kind of data dependence is usually caused by the loop statements of the following form (with forward and backward directions):

```
for i:=1 to n do
begin
    a[i] := .....;    { array variable dependence }
    ... := a[i±c] ...;
    ... := b[i±c] ...;
    b[i] := .....;
    c[i] := c[i±c].....;
    ... := d ...;    { scalar variable dependence }
    d := .....;
    e := .....;
    ... := e ...;
    f := f ...;
end;
```

Therefore, BLT_1 will detect any fetch/store dependences due to array and scalar variables.

CASE 2: Store Dependence

For any scalar and array variables in the loop, there must not be any dependence between iterations due to the memory locations stored between any one or more stanza. In the case of scalar variables, if the following test:

$$\text{BLT}_2: \quad \text{XYZ}_i \cap \text{XYZ}_j \\ \text{where } 1 \leq i \leq j \leq n, n = \text{number of stanzas}$$

does not give \emptyset results, then it indicates data dependence exists. This kind of dependence is usually caused by the loop statements of the form (for all i and j):

```
for i:=1 to n do
begin
    ...;
    a := ...;
    b := b + ...;
    ...;
end;
```

On the other hand, if array variables are present, then BLT₂ must give results having Forward and/or Backward directions with non-zero dependence distances to indicate data dependence. Results which are \emptyset or contain variables with Equal direction or with zero dependence distances imply that there are no data dependences. If $i \neq j$ then the dependence is usually caused by the loop statements of the following form:

```
for i:=1 to n do
begin
    ...;
    a[i] := ...;
    a[i±c] := ...;
    ...;
end;
```

This test does not apply for $i = j$ since the dependence distances produced will always be zero.

It should also be noted that, if there exists data dependences as indicated by the results of the BLTs, the dependence distances can be checked further. If they are greater than or equal to the loop bound, then it can be concluded that there are no data dependences. The following example has a dependence distance of 5, which is equal to the loop bound and thus the iterations are independent.

```
for i := 1 to 5 do
begin
    a[i] := ...
    ... := a[i+5] ...
end;
```

Furthermore, the dependence distances (either forward or backward) may indicate partial parallelization if they are greater than 1. Zero distances imply independence between the iterations.

5.4.1 Summary of the BLTs

In order for the original BTs developed by Williams to handle array variables, they are extended to handle the DRDs to become BLTs and become as follows.

BLT₁: $WYZ_i \cap XYZ_j$
 where $1 \leq i, j \leq n$, n = number of stanzas
BLT₂: $XYZ_i \cap XYZ_j$
 where $1 \leq i, j \leq n$, n = number of stanzas

To determine the existence of data dependences in a loop that contains n stanzas S_i and S_j (where $1 \leq i, j \leq n$):

a. for scalar variables:

$BLT1(S_i, S_j) \neq \emptyset$ for all i and j

$BLT2(S_i, S_j) \neq \emptyset$ for all i and j , $i \leq j$

b. for array variables:

$BLT1(S_i, S_j) \neq \emptyset$ or produces Forward/Backward directions with non-zero dependence distances, for all i and j

$BLT2(S_i, S_j) \neq \emptyset$ or produces Forward/Backward directions with non-zero dependence distances, for all i and j , $i < j$

For dependences due to array variables, the results produced by the above tests will be checked based on the decision rules as in figure 5.9. An implementation of the BLTs is given in Appendix D

If the result produced by the BLTs is or contains:

- | | | |
|------|-------------|--|
| i. | \emptyset | no dependence |
| ii. | = direction | no dependence |
| iii. | < direction | forward dependence |
| iv. | > direction | backward dependence |
| v. | * direction | $\langle \rangle$ dependence which indicates that there is a partial parallelization between two iterations with distances of directions d_1 and d_2 and $d_1 < d_2$. The distance of the dependence is $d_2 - d_1$. |

Figure 5.9: Decisions rules for the data dependence in loops

5.4.2 Nested Loops

The DRD notations can be extended for nested loops where a **vector of reference directions** is formed for each array variable. For each direction, a distance is associated with it. Note that only perfectly nested loops with proper loop level indices are considered in this section. They are of the following form.

```
for i1 := ...
begin
  for i2 := ...
  begin
    ...
    for in := ...
    begin
      BODY(i1,i2,...,in);
    end {for in}
    ...
  end {for i2}
end {for i1}
```

Consider the following perfectly nested loop with two levels.

```
for i1 := 1 to n do
begin
  for i2 := 1 to n do
  begin
    a[i,j] := b[i,j] + c[i,j];
  end {for i2}
end {for i1}
```

The corresponding BSs for the loop body are as follow.

W	X	Y	Z

b[=,=]	a[=,=]	-	-
c[=,=]			

In applying the BLTs, the positions of the directions should match those on the same level. For example, consider the following BSs and the array b.

W	X	Y	Z

b[=,>2]	a[=,=]	-	-
a[=,=]	b[=,>1]		

The outcome of the test $WYZ \cap XYZ$ is decided as follows.

	b	\cap	b	giving	result
level	-----				
1	=	\cap	=	giving	=
2	>2	\cap	>1	giving	>1

5.5 EXAMPLES OF THE APPLICATION OF THE BLTS

In this section, examples will be shown to test the validity of the BLTs.

EXAMPLE 5.1: In the following example, each statement in the loop body is treated as a single stanza.

```

for i := 1 to n do
begin
    a[i] := a[i] + a[i+1];    ...S1
    a[i-1] := k;             ...S2
end;
```

	W	X	Y	Z
S1:	a[<1]	-	a[=]	-
S2:	k	a[>1]	-	-

	WYZ	XYZ
S1:	a[<1], a[=]	a[=]
S2:	k	a[>1]

The results of the BLTs are as follows.

- $WYZ_1 \cap XYZ_2 = \{ a[*2], a[>1] \}$
- $XYZ_1 \cap WYZ_2 = \emptyset$
- $XYZ_1 \cap XYZ_1 = \emptyset$ { scalar only }
- $XYZ_2 \cap XYZ_2 = \emptyset$ { scalar only }
- $WYZ_1 \cap XYZ_1 = \{ a[<1], a[=] \}$
- $WYZ_2 \cap XYZ_2 = \emptyset$
- $XYZ_1 \cap XYZ_2 = \{ a[>1] \}$

The conclusions of the tests are that there are 3 dependences due to array variables (one within S1 and two between S1 and S2) and there is no dependence due to scalar variables.

EXAMPLE 5.2: In this example, the loop body is taken as a stanza.

```
program prloop;
begin
  for i:=1 to 10 do
    begin
      a[i+1] := b[i-1] + c[i];
      b[i] := a[i] * k;
      c[i] := b[i] - 1;
      d[i] := d[i+1] * k;
      aa := bb + cc;
      gg := aa - hh + i;
    end;

    for i:=1 to n do
      begin
        a[i] := a[i+1] * mm;
        b[i-1] := b[i+1] * mm;
        kk := kk + 10;
        aa := bb + cc;
      end;
    end.
end.
```

CONTENTS of all STANZAS

STANZA #	W sets	X sets	Y sets	Z sets
1	b[>1] a[=] k d[<1] b b cc h h i	a[<1] d[=] gg	c[=]	b[=] a a
2	a[<1] m m b[<1] b b cc	a[=] b[>1] a a	k k	-

CONTENTS of all WYZ and XYZ sets

STANZA #	WYZ sets	XYZ sets
1	i	a[<1]
	b[>1]	c[=]
	c[=]	b[=]
	b[=]	d[=]
	a[=]	a a
	k	g g
	d[<1]	
	a a	
	b b	
	cc	
	h h	
2	a[<1]	b[>1]
	b b	a[=]
	cc	a a
	m m	k k
	b[<1]	
	k k	

LOOP DEPENDENCE ANALYSIS

[DIRECTION symbols: < : forward, > : backward, = : equal, * : <>]

WYZ \cap XYZ for stanza 1:

Array dependence : { b[>1], c[=], b[=], a[<1], d[<1] }

Scalar dependence - { aa }

XYZ \cap XYZ for stanza 1:

Scalar dependence - { aa, gg }

WYZ \cap XYZ for stanza 2:

Array dependence : { a[<1], b[*2] }

Scalar dependence - { kk }

XYZ \cap XYZ for stanza 2:

Scalar dependence - { aa, kk }

The conclusions of the tests are as follows.

- (i) for loop 1, there are dependences due to scalar variables aa (fetch/store dependence) and gg (store dependence). There are also dependences caused by the array variables a, b and d.
- (ii) for loop 2, there are data dependences due to scalar variables aa (store dependence) and kk (fetch/store dependence). There are also dependences caused by array variables a and b.

EXAMPLE 5.3: In the following example, a nested loop of two dimensions, adapted from Zima and Chapman (1990), is considered.

```

for i := 1 to n do
begin
  for j := 1 to n do
  begin
    c[i,j] := a[i,j] * b[i,j];           ...S1
    a[i+1,j+1] := c[i,j-2]/2 + c[i-1,j]*3; ...S2
    d[i,j] := d[i-1,j-1] + 1;           ...S3
    b[i,j+4] := d[i,j] - 1;             ...S4
  end
end
end

```

CONTENTS of all STANZAS

STANZA #	W sets	X sets	Y sets	Z sets
S1	a[=,=] b[=,=]	c[=,=]	-	-
S2	c[=,>2] c[>1,=]	a[<1,<1]	-	-
S3	d[>1,>1]	d[=,=]	-	-
S4	d[=,=]	b[=,<4]	-	-

CONTENTS of all WYZ and XYZ sets

STANZA #	WYZ sets	XYZ sets
S1	a[=,=] b[=,=]	c[=,=]
S2	c[=,>2] c[>1,=]	a[<1,<1]
S3	d[>1,>1]	d[=,=]
S4	d[=,=]	b[=,<4]

LOOP DEPENDENCE ANALYSIS

[DIRECTION symbols: < : forward, > : backward, = : equal, * : <>]

WYZ \cap XYZ for stanza 1:

Array dependence = \emptyset

Scalar dependence = \emptyset

WYZ \cap XYZ for stanza 2:

Array dependence = \emptyset

Scalar dependence = \emptyset

WYZ \cap XYZ for stanza 3:

Array dependence - { d[>1,>1] }

Scalar dependence = \emptyset

WYZ \cap XYZ for stanza 4:

Array dependence = \emptyset

Scalar dependence = \emptyset

$XYZ \cap XYZ$ for all stanzas:

Scalar dependence = \emptyset

$WYZ \cap XYZ$ for stanzas 1 and 2:

Scalar dependence = \emptyset

Array dependence : { $a[<^1,<^1]$, $c[=,>^2]$, $c[>^1,=]$ }

$WYZ \cap XYZ$ for stanzas 1 and 3:

Scalar dependence = \emptyset

Array dependence = \emptyset

$WYZ \cap XYZ$ for stanzas 1 and 4:

Scalar dependence = \emptyset

Array dependence = { $b[=,<^4]$ }

$WYZ \cap XYZ$ for stanzas 2 and 3:

Scalar dependence = \emptyset

Array dependence = \emptyset

$WYZ \cap XYZ$ for stanzas 2 and 4:

Scalar dependence = \emptyset

Array dependence = \emptyset

$WYZ \cap XYZ$ for stanzas 3 and 4:

Scalar dependence = \emptyset

Array dependence : { $d[=,=]$ }

The conclusions of the tests are as follows.

1. The test between stanzas S1 and S2 shows that there are forward and backward dependences (i.e., a cycle) between the two of them, due to array a and array c.
2. There is a forward dependence between stanzas S1 and S4 in the inner loop.
3. There are backward dependences in stanza S3.

5.6 TRANSFORMATION OF LOOPS

Loop transformation has been a major focus in the parallelization of sequential programs [Allen (1988), Allen and Kennedy (1984a), Appelbe and Smith (1989), Banerjee (1988), Burke et al. (1988), Callahan et al. (1987), Cytron (1986), Jackson (1985), Kuck et al. (1981), Li (1989), Midkiff and Padua (1986, 1987), Padua and Wolfe (1986), Saltz et al. (1989), Wolf and Lam (1991), Wolfe (1986, 1988)]. Before any transformation can be performed, extensive analysis has to be carried out to determine the data dependence between the iterations. The BLTs discussed above are also able to carry out this analysis.

This section describes the application of some transformation techniques on loops for parallelization. They include the code modification to eliminate data dependences, if possible, or the insertion of proper synchronization constructs so that the iterations will still be able to be executed in parallel even though dependences exist. Most of the work done by other researchers uses the DDG to decide on which transformation methods^{are} to be applied. However, the techniques described in this section are based on the BSs and the BLTs. The discussion assumes a source-level transformation.

5.6.1 Loop Parallelization and Vectorization

There are two types of program transformations: **parallelization** and **vectorization** [Padua and Wolfe (1986), Polychronopoulos (1988), Zima and Chapman (1990)]. Vectorization is a process of transforming loops into vector codes for vector computers, whereas parallelization is a transformation process mainly targeted for shared-memory parallel machines.

For vector codes such as in Fortran 8x [Albert et al. (1988)], if arrays A, B and C are declared as:

```
DIMENSION A(1:n,1:m), B(1:n,1:m), C(1:n,1:m)
```

then the statement:

$$A(1:n) = B(1:n) * C(1:n)$$

is functionally equivalent to the following loop.

```
for j := 1 to m do
begin
  for i := 1 to n do
  begin
    A[i,j] = B[i,j] * C[i,j]
  end
end
```

The assignment statement:

$$A(i,1:m:2) = B(i-1,m:1:-2)$$

is equivalent to the loop:

```
for j := 1 to m step 2 do
begin
  A[i,j] = B[i-1,m+1-j];
end
```

The inner j loop of the following nested loop:

```
for i := 1 to n do
begin
  for j := 1 to n do
    C[i,j] = C[i-1,j] - D[i-1,j+1];
  end;
end;
```

can be vectorized as follows.

```

for i := 1 to n do
begin
    C[i,1:n] = C[i-1,1:n] - D[i-1,2:n+1]
end

```

In the parallelization of loops, there are two types of parallel loops to be generated: the **doall** loops and the **doacross** loops. Iterations of a **doall** loop must be completely independent whereas iterations in a **doacross** loop have to be synchronized to achieve parallelism. Figure 5.10 shows examples of a **doall** loop and a **doacross** loop adapted from Polychronopoulos (1988).

```

doall i = 1,100
    a(i) = b(i) * c(i) + d(i);
    b(i) = c(i) / d(i-1) + a(i);
    if c(i) < 0 then c(i) = a(i) * b(i);
endall

```

(a) a **doall** loop

```

doacross i = 1,100
S:    a(i) = b(i) * c(i) + d(i);
      SEND_SIGNAL(S);
      WAIT_SIGNAL(S,i-3);
S1:   b(i) = c(i) / d(i-1) + a(i-3);
endall

```

(b) a **doacross** loop with synchronization statements

Figure 5.10: Parallelization of loops

5.6.2 Definitions of fetch and store directions

The application of the BLTs on loops will produce results that can be classified as the Forward/Store (FS), Forward/Fetch (FF), Backward/Store (BS) or Backward/Fetch (BF). These definitions will be used to decide how the loops can be transformed.

a. Results of the test $(XYZ_i \cap XYZ_j)$

This test will detect any dependences caused by simultaneous store operations, so the letter X will be appended to each classification.

i. X-Forward/Store (XFS) - if the forward dependence involves a store as for the array a below:

$a[i] := p;$	OR	$a[i+1] := q;$
$a[i+1] := q;$		$a[i] := p;$

ii. X-Backward/Store (XBS) - if the backward dependence involves a store as for the array a below:

$a[i-1] := q;$	OR	$a[i] := q;$
$a[i] := p;$		$a[i-1] := p;$

b. Results of the test $(WYZ_i \cap XYZ_j)$

This test will detect any data dependences due to fetch first and store later operations (that is Y) or dependences due to store first and fetch later operations (that is Z). Hence a letter Y or Z will be appended to each classification.

i. Y and Z-Forward/Store (YFS and ZFS) if the forward dependence involves a store as for the array a below. The values fetched are new values.

<u>YFS</u>	<u>ZFS</u>
p := a[i];	a[i+1] := p;
a[i+1] := q;	q := a[i];

ii. Y and Z-Forward/Fetch (YFF and ZFF) if the forward dependence involves a fetch as for the array a below. The values fetched are old values.

<u>YFF</u>	<u>ZFF</u>
q := a[i+1];	a[i] := p;
a[i] := p;	q := a[i+1];

iii. Y and Z-Backward/Store (YBS and ZBS) if the backward dependence involves a store as for the array a below. The values fetched are old values.

<u>YBS</u>	<u>ZBS</u>
p := a[i];	a[i-1] := p;
a[i-1] := q;	q := a[i];

iv. Y and Z-Backward/Fetch (YBF and ZBF) if the backward dependence involves a fetch as for the array a below. The values fetched are new values.

<u>YBF</u>	<u>ZBF</u>
q := a[i-1];	a[i] := p;
a[i] := p;	q := a[i-1];

5.7 TRANSFORMATION TECHNIQUES FOR SCALAR VARIABLES

If a loop with scalar variables is tested to have data dependences, the BSs of the loop body can be used to show how a loop can be transformed. In this section, some of the well known transformation methods are discussed. The application of these methods is based on the contents of the BSs. For example, consider the loop and its corresponding BSs shown in figure

5.11. The variable c is involved in store and fetch operations and this causes data dependence between iterations. Undefined results may be stored or fetched if the loop is parallelized. The variable c is a member of Z set. If the iterations are to be executed in parallel, care must be taken to ensure that the value of c is properly maintained.

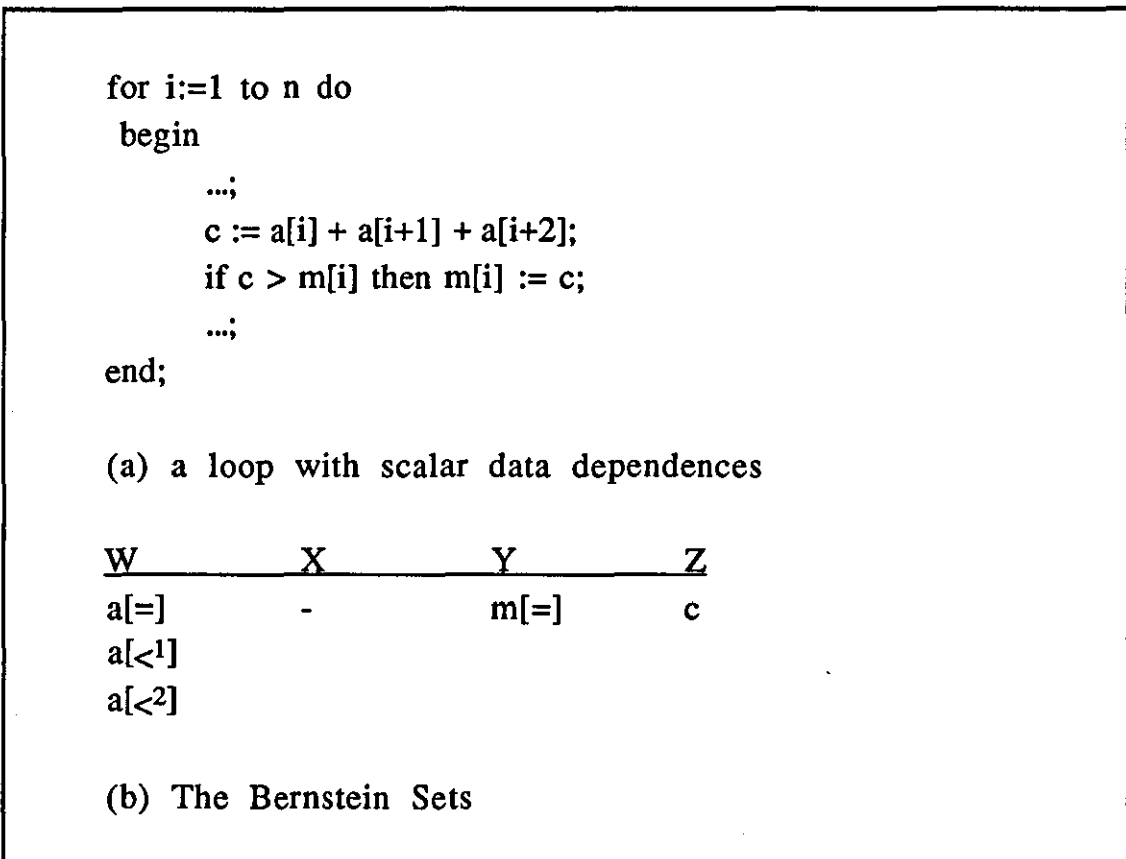


Figure 5.11: Scalar variables data dependences

Based on the types of the BSs which contain the variables, the following transformation decisions can be made. Note that a stanza is the whole loop body.

I. Variables in the W set

Scalar variables in this set do not have any effect since they involve only fetch operations, so, each iteration can be executed in parallel.

II. Variables in the X set

Scalar variables in this set involve only store operations and this could give undefined values by the concurrent execution of the iterations in the loop. Since their values will not be fetched by any iteration, they become redundant and may be moved out of the loop. However, their values in the last iteration have to be computed and saved for later references. Alternatively, they can be renamed with array variables or declared as local variables.

III. Variables in the Y set

The values of the variables in this set will be first fetched and later stored, so the data dependences can be eliminated by scalar renaming (discussed below) or declared as local variables for each iteration.

IV. Variables in the Z set

The values of these variables will be stored and later fetched, thus they can be treated as local variables in each iteration or eliminated by scalar renaming or scalar forward substitution.

Below, some of the well known transformation techniques suitable for transforming loops with scalar variables are discussed [Aho et al. (1986), Padua and Wolfe (1986)], Polychronopoulos (1988), Zima and Chapman (1990)].

I. Scalar forward substitution

In this technique, each occurrence of a variable is substituted with its corresponding expression. In the following example:


```

for i := 1 to n do
begin
    ...
    x := i*i - k;                ... s1
    ...
    sum[x] := sum[x,x+5] + a[i]; ... s2
    ...
end

```

the variable x causes the data dependence. Since its value is first evaluated, its occurrence in the succeeding statements can be replaced by the expression evaluated earlier. This will eliminate the data dependence. Hence, the statement $s2$ can be modified to:

$$\text{sum}[i*i - k] := \text{sum}[i*i - k, i*i - k + 5] + a[i];$$

Since the variables involved must be initially stored and later fetched, then they must be members of the Z set (with the loop body taken as a single stanza). If the variables involve in a new store operation, then care has to be taken so that the new value substituted is the latest assigned expression. A main disadvantage of this technique is that it increases the run-time needed to evaluate each expressions repeatedly.

II. Scalar renaming (expansion)

In this technique, the scalar variable is given a different name to eliminate any data dependences. This is usually done by renaming it with a temporary array variable. For example, consider the following loop.

```

for i:=1 to n do
begin
    ...
    a := 10;
    x := a + b;
    ...
end;

```

The variables a and x which cause the data dependences, can be renamed with array names such as NEWa[i] and NEWx[i] respectively. The transformed loop is as follows which does not have any dependences.

```

for i:=1 to n do
begin
    ...
    NEWa[i] := 10;
    NEWx[i] := NEWa[i] + b;
    ...
end;

```

This technique applies to scalar variables in the X, Y and Z sets of the BSs. However, if the variables in Y set are involved in a statement such as $a := a + 1$ (i.e., a is called a reduction variable), then they cannot be renamed as in the above method since each iteration will be using and updating the value. One main shortcoming of this technique is that it increases the use of variables in the program.

III. Constant Propagation

This is a common technique in optimizing compilers which determines the values of constants in programs. These values are then propagated throughout the programs. This technique eliminates the need for run-time evaluation of the values. Consider the following example.

```

for i:=1 to n do
begin
    pi := 3.142;                ...S1
    twopi := 2*pi;              ...S2
    arr[i] := twopi * rad[i] * rad[i];    ...S3
end;

```

The variables pi and twopi can be determined as constants since they are assigned with constant values. Consider the BSs of the loop body.

	W	X	Y	Z

S1:	-	pi	-	-
S2:	pi	twopi	-	-
S3:	twopi	arr[i]	-	-
	rad[i]			

From the first BSs, pi is assigned a constant value since there are no other variables in the other sets. This value is then fetched and later stored in twopi, thus showing that twopi is also a constant. Therefore, the loop can be transformed into vector instructions as follows.

```

pi := 3.142;
twopi := 6.284;
arr[1:n] := twopi * rad[1:n] * rad[1:n];

```

5.8 TRANSFORMATION TECHNIQUES FOR ARRAY VARIABLES

In this section, some transformation methods suitable for loops containing array variables are discussed. These methods are standard techniques for program transformation and are widely discussed in the literature [Allen and Kennedy (1987), Ebenstein and Mcdermott (1990), Lewis and El-Rewini (1992), Padua and

Wolfe (1986), Polychronopoulos (1988), Smith and Appelbe (1989), Wolfe (1989b), Zima and Chapman (1990)]. Here, the rules for transformation are based on the contents of the BSs and the results of BLTs.

I. Loop distribution

In this technique, a loop is broken into several loops to distribute the control over groups of statements in its body. This is particularly convenient for vectorization. As an example, the single loop below has a YFF dependence caused by array a.

```

for i:=1 to n do
begin
    c[i] := a[i+2] * b[i];    ...S1
    a[i] := b[i] + c[i];    ...S2
end

```

It can be transformed to become two loops as follows.

```

for i:=1 to n do
begin
    c[i] := a[i+2] * b[i];
end

for i:=1 to n do
begin
    a[i] := b[i] + c[i];
end

```

Then they can be vectorized to become:

```

c(1:n) = a(1:n+2) * b(1:n)
b(1:n) = b(1:n) + c(1:n)

```

The semantics of the statements are preserved since the fetched element of a[i+2] in S1 contains its old value. In the following example, the array a has a XFS data dependence.

```

for i:=1 to n do
begin
    a[i] := p;
    a[i+1] := q;
end;

```

It can also be distributed into two loops to become:

```

for i:=1 to n do
begin
    a[i] := p;
end;

for i:=1 to n do
begin
    a[i+1] := q;
end;

```

The vectorized statements are as follows.

```

a[1:n] := p;
a[2:n+1] := q;

```

In general, the rules for this transformation are as follows:

- a. the variables are store dependent This usually appears in initialization loops.
- b. the statements do not have forward and backward dependences (i.e., in a cycle) such as in the following example. Statements S1 and S2 contain an array a that has a forward dependence and array b that has a backward dependence. This means that, they cannot be separated into different loops.

```

for i:=1 to n do
begin
    a[i+1] := b[i-1] + c[i];           ... S1
    b[i] := a[i] * d[i];               ... S2
    c[i] := b[i] + d[i];               ... S3
end;

```

However, the statement S3 has an Equal dependence with S1 and S2 and hence the whole loop can be distributed as follows.

```

for i:=1 to n do
begin
    a[i+1] := b[i-1] + c[i];           ... S1
    b[i] := a[i] * d[i];               ... S2
end;

```

```

for i:=1 to n do
begin
    c[i] := b[i] + d[i];               ... S3
end;

```

- c. the variables are NOT YBF or YFS because loop distribution destroys the dependences and the new values of the arrays are not properly accessed.

ii. Statement reordering

This method involves exchanging the textual positions of two statements in a loop body. The following loop cannot be vectorized due to the presence of a data dependence on array a which is YBF. The values fetched are new values that are assigned by the other statement, except for the first element.

```

for i:=1 to n do
begin
    c[i] := a[i-1] - 4;
    a[i] := b[i] * 2;
end

```

If the statements are reordered, the loop becomes:

```

for i:=1 to n do
begin
    a[i] := b[i] * 2;
    c[i] := a[i-1] - 4;
end

```

Now the data dependence of array a has a ZBF dependence where the fetched values of a[i-1] are new values. Thus, the loop can be distributed and vectorized.

```

a[1:n] = b[1:n] * 2
c[1:n] = a[0:n-1] - 4

```

In general, the conditions for this kind of transformation are as follows.

- a. For the scalar variables, they are members of the W or X sets only and NOT members of the Y and Z sets
- b. For the array variables, the types of dependence are YFS and YBF. These dependences involve new values that are being fetched initially. Hence, the statement with the fetch operation may be reordered to appear later in the sequence of iteration execution.
- c. For the array variables, there are NO equal (=) dependence directions after the BLTs have been applied, i.e., there are no loop-independent dependences on any variables such as the array a in the following example. Note that array b has a

YFS dependence.

```
for i:= 1 to n do
begin
    a[i] := b[i];
    b[i+1] := a[i] + ...
end
```

iii. Loop interchange

In this technique, any two levels of a perfectly nested loop are exchanged. For the example below, the BLTs will indicate that the inner loop is unparallelizable, since there is a forward dependence for the j loop for array a[=,<]. If the loop statements are interchanged, the new inner loop can then be parallelized.

```
for i:=1 to n do
begin
    for j:=1 to n do
    begin
        a[i,j+1] := a[i,j] * b[i,j];
    end
end
```

The interchanged version is as follow.

```
for j:=1 to n do
begin
    for i:=1 to n do
    begin
        a[i,j+1] := a[i,j] * b[i,j];
    end
end
```

After interchanging, the outer j loop will be executed sequentially while the inner i loop can be executed concurrently. This is suitable for vectorization as in the following form.


```

for j:=1 to n do
begin
    a[i:n,j+1] := a[i:n,j] * b[i:n,j];
end

```

However, for parallelization, the earlier version (before interchanging) is preferable since then there will be less overheads incurred in executing the outer loop concurrently.

For a nested loop such as in the above example, this will involve more than one direction for the array variables, that is, a vector of reference directions is needed. Let D be a vector of data dependence directions:

$D = (d_1, d_2, \dots, d_n)$ with $d_i = (<, >, =, *)$,
for all $1 \leq i \leq n$, n = number of dimensions

The necessary condition for this kind of transformation is that, given a nested loop of n dimensions, two loop statements with array directions $A[\dots, d_i, \dots, d_j, \dots]$ cannot be interchanged if one of them (i.e., d_i or d_j) has a forward direction and the other one has a backward direction. This means that if an array has directions such as $A[\dots, <, \dots, >, \dots]$, it indicates that the two loops are not interchangeable.

iv. Index set splitting

In this technique, the loop is divided into two or more loops with partial size. The loops can then be executed concurrently. The following example:

```

for i := 1 to 200 do
begin
    b[i] := a[201-i] + c[i];
    a[i] := c[i-1] * 2;
end

```

can be split into two loops to become:

```
for i := 1 to 100 do
begin
    b[i] := a[201-i] + c[i];
    a[i] := c[i-1] * 2;
end
```

```
for i := 101 to 200 do
begin
    b[i] := a[201-i] + c[i];
    a[i] := c[i-1] * 2;
end
```

This technique requires an extraction of the distance of the dependence in order for the loop bound to be split properly. In the above example, the distance is 200.

v. Node splitting

This method breaks expressions occurring in statements into several parts. This involves a lower level treatment of expressions in statements of the loops. The arrays involved must be those which do not contribute to any results in the BLTs such as arrays c and d as in the following example.

```
for i := 1 to n do
begin
    b[i] := a[i] + c[i] * d[i];           ...S1
    a[i+1] := b[i] * (d[i] - c[i]);       ...S2
end
```

The expression in S1 can then be split as in the following example, with t1 and t2 as temporary variables.

```
for i := 1 to n do
begin
    t1[i] := c[i] * d[i];
    t2[i] := d[i] - c[i];
    b[i] := a[i] + t1[i];
    a[i+1] := b[i] * t2[i];
end
```

The first two statements of the loop can be vectorized as follows.

```
t1[1:n] := c[1:n] * d[1:n];
t2[1:n] := d[1:n] - c[1:n];
for i := 1 to n do
begin
    b[i] := a[i] + t1[i];
    a[i+1] := b[i] * t2[i];
end
```

vi. Loop blocking.

One way to transform a loop with a dependence distance ≥ 2 is to perform the loop blocking transformation [Padua and Wolfe (1986), Polychronopoulos (1988)]. It creates doubly nested loops out of a single loop, by organizing the computation in the original loop into chunks of approximately equal size. This is also called partial parallelization mentioned in Section 5.4 above. It is often used to manage vector registers, caches or local memories with small sizes. Consider the following loop.

```
for i := 1 to n do
begin
    p := a[i];
    a[i+k] := ...;
end;
```

For the index reference in the form ' $i+k$ ', where $k \geq 2$, the loop can be transformed into the following form.

```
for j := 1 to n STEP k do
  parfor i := j to min(j+k-1,n) do
    begin
      ...;
      p[i] := a[i];
      a[i+k] := ...;
      ...;
    end;
```

vii. Array Alignment

Array alignment is a technique which involves adjusting the array reference to eliminate the data dependences. It transforms a loop-carried dependence into a loop-independent dependence. In the following example, there is a loop-carried dependence caused by array a.

```
for i:=1 to n do
begin
  a[i] := b[i+1] - c[i-1];
  d[i] := a[i-1] + e[i];
end;
```

The data dependence can be eliminated by array alignment as in the following code:

```
for i:= 0 to n do
begin
  if i > 0 then a[i] := b[i+1] - c[i-1];
  if i < n then d[i+1] := a[i] + e[i+1];
end;
```

Variables involved are usually of the types YBS and ZBS such as a[i-1] above where they can be aligned to a[i]. This is allowed

since the backward reference is a reference to an old value.

viii. Loop unrolling

This technique makes one or more copies of the loop body and thus increases the stride. This will reduce the control overhead in executing the loops as in the following example.

```
for i := 1 to 1000 do
begin
    a[i] := b[i+2] * a[i-1];
end
```

It can be unrolled to become a loop with a stride of 2 which has only 500 (i.e., 50% less) iterations to be generated.

```
for i := 1 to 1000 step 2 do
begin
    a[i] := b[i+2] * a[i-1];
    a[i+1] := b[i+3] * a[i];
end
```

A similar technique called loop replication makes copies of statements in loop body without changing the stride [Allen and Kennedy (1987)]. Consider the following example.

```
for i := 1 to n do
begin
    a[i] := b[i] * c[i];
    d[i] := a[i] * a[i-1];
end
```

It has a ZBF dependence caused by operations on array elements $a[i]$ and $a[i-1]$. To be able to perform array alignment, as discussed above, the first statement can be replicated and the array element $a[i]$ in the second and third statement renamed to $NEWa[i]$.

```

for i := 1 to n do
begin
    a[i] := b[i] * c[i];
    NEWa[i] := b[i] * c[i];
    d[i] := NEWa[i] * a[i-1];
end

```

The loop can then be distributed and aligned as follows, thus eliminating all data dependences.

```

for i := 1 to n do
    NEWa[i] := b[i] * c[i];
for i := 0 to n do
begin
    if (i > 0) then a[i] := b[i] * c[i];
    if (i < n) then d[i+1] := NEWa[i+1] * a[i];
end

```

ix. Array renaming (or variable copying)

Scalar renaming is useful for eliminating data dependences involving scalar variables. For array variables, they can also be renamed to eliminate data dependences. Consider the following example where there is a data dependence on array a which is ZFF.

```

for i:=1 to n do
begin
    a[i] := b[i] + c[i];
    d[i] := a[i] + a[i+1];
end;

```

It can be transformed into the following version after renaming the array a[i] to NEWa[i], thus eliminating the data dependence.

```

for i:=1 to n do
    NEWa[i] := a[i];
for i:=1 to n do
begin
    a[i] := b[i] + c[i];
    d[i] := a[i] + NEWa[i+1];
end;

```

The loops can then be vectorized as follows.

```

NEWa[1:n] := a[1:n];
a[1:n] := b[1:n] + c[1:n];
d[1:n] := a[1:n] + NEWa[2:n+1];

```

For this technique, the array variables must be of the types YFF, ZFF, YBS or ZBS as derived by the BLTs, to enable them to be renamed with different array names. This is because the forward reference is a reference to an old value. In the previous example, $a[i+1]$ refer to old values of a and thus can be renamed.

x. Synchronization statements.

In some cases, data dependences cannot be eliminated at all. When array variables with complex array subscripts, such as coupled subscripts or array subscripts or those other than $(i \pm \text{const})$, are met, usually data dependence is assumed to exist. Since the dependences are difficult to be eliminated, the synchronization instructions such as LOCK and UNLOCK (for shared memory machines) are used [Midkiff and Padua (1987), Tang et al. (1990), Wolfe (1988)]. These instructions will create the critical regions in which only one process will be able to execute its critical region at a time. The following example shows one loop with LOCK and UNLOCK statements.

```

for i:=1 to n do
begin
    ...
    LOCK;
    if a[i] > max then
        max := a[i];
    UNLOCK;
    ...
end;

```

xi. Idiom recognition

In this technique program sections are detected and recognized to perform some particular functions for which an efficient special implementation exists. Examples of such functions are SUM(a) and PRODUCT(a) [Zima and Chapman (1990)].

Table 5.1 gives a summary of the transformation techniques that can be performed, based on the results of the BLTs.

5.9 RELATED ISSUES ON LOOP DEPENDENCES

As discussed in Section 5.4 above, the BLTs can handle BSs containing array variables. DRDs are attached to the array names, indicating how they are being referenced. The subscripts that are allowed are simple expressions of the forms $[i \pm \text{constant}]$. However, Shen et al. (1989) have showed that there are other forms of complex subscript expressions commonly found in programs, although they are not found as frequent as the simple expressions allowed by BLTs. These complex expressions include coupled subscripts (i.e., loop indices appearing at any level), nonlinear subscripts, array subscripts and symbolic subscripts. The BLTs will assume that there are data dependences when these kinds of array subscripts are encountered.

X-Forward/Store (XFS)	Loop distribution
X-Backward/Store (XBS)	Loop distribution
Y-Forward/Store (YFS)	Statement reordering Loop blocking
Y-Forward/Fetch (YFF)	Loop distribution Array renaming Loop blocking
Y-Backward/Store (YBS)	Loop distribution, Array renaming, Array alignment
Y-Backward/Fetch (YBF)	Statement reordering
Z-Forward/Store (ZFS)	Loop distribution Loop blocking
Z-Forward/Fetch (ZFF)	Statement reordering Array renaming Loop blocking
Z-Backward/Store (ZBS)	Statement reordering, Array renaming, Array alignment
Z-Backward/Fetch (ZBF)	Loop distribution,

Table 5.1: Loop transformations and the dependence types

Handling complex subscript expressions has been studied by several researchers using numerical methods [Banerjee (1988), Li and Yew (1990), Wolfe (1989) and Zima and Chapman (1990)]. They are discussed in Chapter 3. Solutions for array subscripts have been discussed by Polychronopoulos (1988). Apart from these problems, loops sometimes contain non-uniform loop indexing, procedure calls and conditional statements. This increases the complexity of the loop analysis for data dependence. For procedure calls, they can be handled by the Inter-procedural Analysis (IPA) and this is discussed in Chapter 6. The problem caused by non-uniform loop indexing needs modification before any analysis and transformation can be performed. An example of such a loop is as follows.

```
for i := 100 downto 1 do
    for j := 5 to 100 step 3 do
```

This can be overcome by normalizing the loop indexing so that each loop will start from 1 with a stride of 1. This, however, will complicate the subscript expressions in the loop body and thus making it more difficult to be handled by the BLTs.

Another problem that is encountered by the dependence analysis is symbolic subscripts where the subscripts contain variables [Haghighat (1990)]. Sometimes, this can be solved by constant propagation. However, in some cases, the actual values of the symbolic expressions are only known at run-time. One simple solution is to generate conditional vectorized statements [Luecke et al. (1991)]. Consider the following example.

```
for i:=1 to n do
    a[i+k] := a[i]/b[i] + c[i];
```

If the value of k is not known at compile time, the translation could look like the following code.

```

if (k < 1 or k >= n) then
    a[k+1:n+k] := a[1:n] / b[1:n] + c[1:n]
else
    for i:=1 to n do
        a[i+k] := a[i]/b[i] + c[i];

```

The presence of complex control flow in loops also poses problems for the BLTs. This creates the control dependence between two or more statements in a loop as mentioned in Chapter 3. This dependence prevents the execution of one statement while executing the other [Allen et al. (1983), Ferrante et al. (1987), Padua et al. (1980), Riseman (1972)]. One simple solution to this problem is to convert them into data dependences. Logical variables are introduced to control execution of statements. Consider the following example.

```

for i := 1 to n do
begin
    if a[i] > 0 then
        a[i] := b[i] + c[i];
end

```

The control dependence can be removed as follows.

```

for i := 1 to n do
begin
    t := a[i] > 0;
    if (t) then a[i] := b[i] + c[i];
end

```

The vectorized form of the above loop makes use of the `where` statement, as in Fortran 8x, and they are as follows.

```

t[1:n] := a[1:n] > 0;
where (t[1:n]) a[1:n] := b[1:n] + c[1:n];

```

5.10 SUMMARY

This chapter has described the Bernstein Loop Tests (BLTs) to detect parallelism in loops. Their iterations can be parallelized if there are no data dependences. These inter-iteration data dependences are usually caused by the references (i.e., fetch and store operations) of the same array elements by the different iterations.

The BLTs are able to detect parallelism in loops containing array variables as well as the scalar variables. The Data Reference Directions (DRDs) have been used to indicate how the array variables are being referenced by the various iterations. These directions (<, > and =) are augmented with the array names in the BSs. However, the subscript expressions allowed by the BLTs are simple expressions and this may not uncover most of the parallelism that exists.

This chapter has also discussed the transformation techniques that can be carried out on loops. The rules for transformation developed in this chapter are based on the BLTs as well as the contents of the BSs. Once the data dependences have been ascertained, the loops will then be transformed by modifying their codes to remove the dependences. Combinations of techniques can be used to do the transformations. The process must ensure that the semantics of the loops are maintained. For those loops whose dependences cannot be eliminated, certain synchronization constructs such as LOCK and UNLOCK are inserted. As a conclusion, this chapter has showed that the BSs and the results of the BLTs are very useful in making decisions on the type of transformation methods to be carried out in the parallelization of programs.

CHAPTER 6

INTER-PROCEDURAL ANALYSIS

6.1 INTRODUCTION

Much effort has been made on performing the dependence analysis on sequential programs in the attempt to parallelize them, especially the loops [Allen and Kennedy (1984a, 1984b, 1987), Allen et al. 1987b), Banerjee (1988), Burke et al. (1988), Callahan et al. (1987), Kuck et al. (1981, 1984), Li and Yew (1990), Mohd-Saman and Evans (1993), Padua and Wolfe (1986), Wolfe (1989a, 1989b), Wolfe and Banerjee (1987)]. However, program restructurers sometimes have to make some conservative assumptions as to whether to carry out the parallelization or not, due to insufficient information caused by the invocations of procedure calls [Banning (1979), Barth (1978), Burke and Cytron (1986), Callahan et al. (1986), Callahan and Kennedy (1987), Cooper and Kennedy (1988, 1989), Cooper et al. (1986), Havlak and Kennedy (1991), Li (1989), Li and Yew (1988), Schouten (1990), Triolet (1985), Triolet et al. (1986)].

Introduction of procedures in a program causes certain information on the usage of the variables to be hidden from being analysed for optimization and parallelism. Procedures that are called may be modifying some common or global variables and may inhibit any form of parallelism. Hence, an **Inter-procedural Analysis (IPA)** is greatly needed as part of the dependence analysis. It will give the compiler more information to decide on the status of the data dependence in loops and programs. This is one of the most important aspects in the implementation of a parallelizing software tool.

This chapter discusses the needs for IPA and its methods in collecting information regarding procedure calls and the usage of variables in procedure bodies. The focus is how to propagate information about variables in a procedure body back to the calling point. It proposes a new way to handle inter-procedural information by using the Bernstein Sets [Bernstein (1966), Mohd-Saman and Evans (1993), Williams (1978)]. The discussions in this chapter are as follows. Section 6.2 discusses the aliasing problem. Section 6.3 to Section 6.6 present algorithms to collect

information related to procedure calls. This information is to be used in the BTs and the BLTs. Calls in loops are treated in Section 6.7 and Section 6.8 summarises the chapter.

6.2 THE ALIASING PROBLEM

Discussions on the Inter-procedural Analysis (IPA) are usually focused on the problem of aliasing caused by procedure calls. They include determining information on the aliased variables in procedures and collection of reference information of the objects in the procedure body. This reference information relates to when and how the objects are referenced [Schouten (1990)]. It will be used in the DDA to discover any parallelism that may exist. This chapter addresses the aliasing problem that is mainly caused by parameter passing in the procedure calls.

DEFINITION 6.1

Aliasing is a situation that occurs when two or more variable names refer to the same object.

In programming languages, variable aliasing usually occurs when the names are declared explicitly as in FORTRAN Equivalence or C Union or Pascal Variant Record. Aliasing may also occur when two or more parameters have the same name passed to the called procedure. In using pointers, aliasing may arise when two or more pointers are set to point to the same object. All these situations should indicate data dependence in the DDA although the names involved are different. Figure 6.1 shows the examples caused by program declarations aliasing, pointer aliasing and parameter aliasing. Parameter aliasing can also occur when a global variable is passed to a procedure where there are references or modifications on the actual global variable. Consider the following example where the parameter x is declared as a parameter for call by reference.

```

declare gvar global;
perform_proc(gvar);
procedure perform_proc(x);
begin
    ...; x := ...;
    ... := gvar; ...
end;

```

Since the global variable `gvar` is passed to `x`, then `x` will be aliased with the actual `gvar` in the body of the procedure.

Equivalence	A(1,1),B(1,1),C(100)	(FORTRAN)
Union	{ int x; int y; char c; float f }	(C)
Record		(Pascal)
	<pre> x : integer; case y of a: (p:integer;) b: (q:real;) end; </pre>	

(a) Aliasing by declaration

```

ptr := new(rec);
ptr2 := ptr;

```

(b) Pointer Aliasing

Call P(A,A,B)

PROCEDURE P(X,Y,Z)

(c) Parameter Aliasing

Figure 6.1: Examples of the various aliasings

6.2.1 Parallelization of procedure calls

Statements in straight line codes containing procedure calls needs further analysis to discover whether they are parallelizable or not. The statement C1 and loop L1 in figure 6.2 can be executed in parallel if the compiler can discover that the array R is not modified in the procedure P. By inspecting the statements, C1 and L1 are actually parallelizable since there are no data dependences that exist between them. However, if C1 is changed to P(R,S,Q,N), then the IPA should detect a dependence caused by the array R and hence C1 and L1 should not be parallelized.

In another case, the loop L2 in the procedure body in figure 6.2, needs IPA to discover if Y or Z are not aliased with X in the different iterations. If they are not, then the loop iterations can be run in parallel.

For the loop L1 shown in figure 6.3, the call in its body may present some problem to decide whether its iterations are parallelizable or not. They are parallelizable only if the compiler can establish the reference pattern of the array A in the procedure P. This means that, the compiler needs to know whether any other elements of A are being referenced or not when the procedure is called.

```
P(Q,R,S,N);          -----C1
FOR I := 1 TO N DO -----L1
    SUM := SUM+R[I]

PROCEDURE P(X,Y,Z,N)  -----P 1
BEGIN
    FOR I :=1 TO N DO-----L2
        X[I] := Y[I] + Z[I]
    END
```

Figure 6.2: Parallelizable straight line codes of C1 and L1

```

FOR I := 1 TO N DO-----L1
    A[I] := P(A,I,N)

PROCEDURE P(X,I,N)          -----P1
BEGIN
    FOR I:= 1 TO N DO      -----L2
        X[I] := X[I]/N
    END

```

Figure 6.3: Parallelizable loop L1

6.2.2 In-line Expansion

A way to solve the above problem so that the DDA can be carried out without IPA, is to perform *in-line expansion* on the procedure calls [Cooper et al. (1991), Davidson and Holler (1988)]. This means that the body of the called procedure will replace each point of the procedure call. Great care must be taken in substituting the appropriate procedures so that the expanded program still preserves the semantics of the original program. An example of in-line expansion is shown in figure 6.4.

Apart from eliminating the need for IPA, in-line expansion has other advantages. It gives an opportunity to discover more fine grain parallelism since all statements are at the same level. It also removes the overheads normally incurred when calls are made. These overheads occur when the contents of registers are saved and restored and in transferring control to and from the called procedure.

This method however, has several limitations. In the worst case, it will increase the program size exponentially. It also destroys the modularity of the program and makes it very difficult to understand. Also the program becomes less maintainable. Since the modularity is lost, linking compiled procedures cannot be done

and this may incur more compiling time. Furthermore, recursive calls are difficult to handle. Apart from this, local variables at lower levels now become global, thus introducing additional data dependence and may inhibit parallelism. However, there are several studies that show in-line expansion is still a useful method for reducing execution time and it is implemented in several compilers [Cooper et al. (1991), Davidson and Holler (1988)].

```

nos := square(p);
for i := 1 to 10 do
    perform_add_array(a,b,c,nos);

procedure square(q);
begin
    r := q * q;
    return r;
end;

procedure perform_add_array(x,y,z,n);
begin
    for i := 1 to n do
        x[i] := y[i] + z[i];
    return x;
end;

```

(a) An example segment of a program before in-lining

```

nos := p * p;
for i := 1 to 10 do
    for i2 := 1 to nos do
        a[i2] := b[i2] + c[i2];

```

(b) An in-lined program

Figure 6.4: Example of an in-line expansion

6.2.3 Inter-procedural Constant Propagation

Inter-procedural Constant Propagation (ICP) is an important strategy to determine, at compile time, the values of some variables passed to procedures, which are constant during execution [Aho et al. (1986), Callahan et al. (1986), Schouten (1990), Wegman and Zadeck (1991)]. This will eliminate the occurrence of variable aliasing. Since it is performed at compile time and the values of the variables have already been determined, it will save execution time.

ICP has several important effects. Different codes may be generated if values of some variables are known such as in the following source code.

```
if (i <> 0) then perform_P(a,b,c);
```

If it can be determined that i will always have a zero value then the codes for 'perform_P' do not have to be generated at all.

Another example where ICP will be useful is that in the determination of a parallelizable loop. Consider a loop in the following example.

```
procedure proc(a,k)
begin
    ...;
    for i := 1 to 10 do
        a[i] := a[i+k] + 10;
    ...;
end;
```

If ICP can determine that $k = 0$, then the loop is parallelizable. If $k > 0$ or $k < 0$, then there is a forward or backward dependence of loop iterations. If nothing is known then the loop has to be executed sequentially.

ICP is useful especially for in-line expansion. It can improve the source code considerably. Consider the program segment in figure 6.5(a). Figure 6.5(b) shows the in-lined code produced after ICP has been performed.

The simplest way to perform the ICP is to trace the execution of the program and gather the information about the values of variables involved. If any of them have constant values, they can be substituted. However, a variable may take different values, depending on the input and the parameter passed. Discussions on the various implementations of this technique are given in several literature [Aho et al. (1986), Callahan et al. (1986), Wegman and Zadeck (1991)]

6.2.4 Collection of Reference Information

Several techniques have been proposed on how to collect the information in IPA about the effects of procedure calls on the variables that will be used in the dependence tests. Barth (1978) refers this information as 'summary data flow information'.

(a) Li (1989) uses data structures called Atom and Atom Images, to keep track of every linear subscript expression in every array reference in a procedure. These data structures also keep information on the bounds on the iteration variables in terms of loop invariants and outer loop iteration variables. Then any standard dependence tests can be applied on these atom images to detect any data dependences.

(b) Triolet has suggested a system of linear inequalities to represent referenced regions of an array, resulting in a convex hull in the k-dimensional array reference space [Triolet (1985), Triolet et al. (1986)]. This system of inequalities is used in the dependence tests.

```

program prog;
begin
    perform_proc(1,2,3);
end.

procedure perform_proc(a,b,c);
begin
    i := 10 * c;
    if (i <> 0) then j := a else j := b;
    k := j * j;
    l := 10;
    write(i,j,k,l);
end;

```

(a) An example of program

```

program prog;
begin
    write(30,1,1,10);
end.

```

(b) An in-lined version after ICP

Figure 6.5: Example of Inter-procedural Constant Propagation

(c) Burke and Cytron (1986) have suggested the use of array linearization. For a statement such as the following:

$a[i,j] := b[i,j];$

the two references are $MEM[N*(i-1)+j+K_a]$ for array 'a' and $MEM[N*(i-1)+j+K_b]$ for array 'b'. K_a and K_b are the starting locations of arrays 'a' and 'b', both of size $N \times N$, in memory. This automatically detects any aliasing.

(d) Restricted Regular Section Descriptors (RRSD) are another form of representation of sections of array references [Callahan and Kennedy (1987)]. Any two regions that intersect can be found by merging them.

(e) Triplets described in [Schouten (1990)], extends the RRSD to use triplets of the form $ij = l : u : s$, where l is the lower limit on ij , u is an upper limit and s is the stride. For nested loops:

```
for i1 = l1 to h1 step s1 do
  for i2 = l2 to h2 step s2 do
    ...; A[i1,i2, ... ] := ...; ...;
  end
end
```

the triplet region is $A[l1 : h1 : s1, l2 : h2 : s2, \dots]$. This triplet region is then used in the dependence tests as described by Banerjee and Wolfe [Banerjee (1988) and Wolfe and Banerjee (1987)].

(f) Data Access Descriptors (DAD) described in [Balasundram and Kennedy (1989)] give array regions with simple sections. Simple sections can be merged to determine any intersection.

(g) Cooper and Kennedy (1988, 1989) have developed a method called the binding multigraph to solve the IPA problem. A binding multigraph represents bindings of formal parameters.

Each node in a graph represents a formal parameter and an edge represents a binding of parameter n to parameter q through some call. The graph is traversed depth-first, propagating the information upwards.

In this chapter another method, using the BSs is proposed as a way to collect information in the IPA [Bernstein (1966), Mohd-Saman and Evans (1993), Williams (1978)].

6.3 BERNSTEIN SETS FOR PROCEDURE CALLS

The BSs have proved to be very useful in testing for parallelism in programs. Williams (1978) has developed a set of tests, termed as the Bernstein Tests (BTs) that can determine parallelism in straight line codes. Mohd-Saman and Evans (1993) have developed the Bernstein Loop Tests (BLTs) for testing loop dependence. These tests are discussed in Chapter 5 of this thesis. The presence of procedures in programs, however, needs further treatment on how the BSs can be used to maintain the flow of information from the calling part of the program to the called procedure body.

6.3.1 Simple-Call Algorithm

To derive the actual BSs of procedure calls, Williams has suggested the method of 'joining' the BSs of the call statement and the BSs of the procedure body that will give the actual BSs of the call that can be used in the BTs. In this section, this algorithm, which is termed as Simple-Call Algorithm (SCA), will be discussed to derive the actual BSs of the procedure call. Later, this scheme will be improved.

Consider the program segment in figure 6.6 which has a statement $S1$ and a call statement $C1$ to the procedure body $P1$. The task now is to determine whether $S1$ can be executed concurrently with $C1$ or not. The BSs of $S1$ are trivial but for $C1$, the procedure body needs to be analysed. First, the BSs of $P1$ (body of the procedure) are formed. The variable t in $P1$ is just a local variable

which will not have any effect on the dependence test between S1 and C1 and hence it (and any local variables) will be ignored. Other variables found in the procedure that are not declared locally are taken as global variables. They will be included in the the BSs of the procedure body because any modification on these global variables in the procedure body may affect the dependence.

The contents of the BSs of C1 initially depend on the mechanisms of the parameter passing used in the definition of the procedure. If call-by-name is used, then the actual parameters become members of the W set. This is because initially the values of the actual parameters will be fetched and passed to the formal parameters of the procedure. There are no values stored back to the actual parameters at the end of the call. On the other hand, if call-by-reference is used, the actual parameters will be included in the Y set since they will be fetched first and will later be stored at the end of the call. Then the BSs of C1 and P1 are merged to form the actual BSs of C1 that can be used in the BTs and BLTs. The process of merging is the actual propagation of information from the body of the procedure back to the calling statement. This Simple-Call Algorithm (SCA) is given in figure 6.7. The algorithm for MERGE for any two stanzas is given in Chapter 4. It should be noted that the MERGE operation is not commutative because MERGE(a,b) may not give the same result as MERGE(b,a).

```

a := b + c; -----S1
CALL perform_p(a,b,c); -----C1

PROCEDURE perform_p(q,r,s); -----P1
VAR t INTEGER;      { t - local variable, u global variable }
BEGIN
    t := r + q;
    s := t;
    u := s;
END;
```

Figure 6.6: Parallelism of S1 and C1

SIMPLE-CALL ALGORITHM (SCA)**INPUT:** A call C to procedure P with actual parameters AP**OUTPUT:** BSs of the call C (BS_{call})**BEGIN**

- (1) BS_{init} = FORM_BS(C): form the initial BSs of the call.
- (2) BS_{proc} = FORM_BS(P): form the BSs of P by including only the global variables.
- (3) BS_{call} = MERGE(BS_{init},BS_{proc}): propagation of information from body of P to C

END

Figure 6.7: Simple-Call Algorithm

6.3.2 Example of handling call-by-value parameters

For the program segment in figure 6.6 above, if the mechanism for passing of parameters is call-by-value, then the initial BSs of the call, C1, will have all of its actual parameters classified as members of the W set (BS_{init}). Then the BSs of the call are merged with the BSs of P (BS_{proc}) to give the actual BSs of C1 (BS_{call}). Figure 6.8 shows the BSs of C1 before and after merging and the dependence tests between S1 and C1. The BTs can then be applied on the BSs of S1 and C1 to show that S1 and C1 cannot be run concurrently due to conflicting read and write operations on the variable a (see figure 6.8(e)).

<u>W</u>	<u>X</u>	<u>Y</u>	<u>Z</u>
b,c	a	-	-

(a) The BSs for S1

<u>W</u>	<u>X</u>	<u>Y</u>	<u>Z</u>
a,b,c	-	-	-

(b) The BS_{init} for C1 for call-by-value

<u>W</u>	<u>X</u>	<u>Y</u>	<u>Z</u>
-	u	-	-

(c) The BS_{proc} of P1

<u>W</u>	<u>X</u>	<u>Y</u>	<u>Z</u>
a,b,c	u	-	-

(d) The actual BS_{call} of C1 after propagation

$$\begin{aligned} WYZ_{S1} \cap XYZ_{C1} &= \emptyset \\ XYZ_{S1} \cap WYZ_{C1} &= \{ a \} \\ XYZ_{S1} \cap XYZ_{C1} &= \emptyset \end{aligned}$$

(e) BTs on S1 and C1

Figure 6.8: Dependence tests for the example in figure 6.6

6.3.3 Example of handling call-by-reference parameters

If call-by-reference is used, the actual parameters will be classified as members of the Y set in the initial BSs of C1. This has to be dealt in two cases. In the first case, if there is no actual parameter duplication, i.e., all of the actual parameters are of different names, such as in $p(a,b,c)$, then the actual BSs of C1 can be derived by using the same strategy used for handling call-by-value as discussed in Section 6.3.2 above. Figure 6.9 shows an example of call-by-reference derivation of BSs of C1 for the program segment in figure 6.6. In the second case, calls with duplicating parameters create situations that have more than one occurrence of the same variable in the BSs. This will be discussed in Section 6.3.5 below.

<u>W</u>	<u>X</u>	<u>Y</u>	<u>Z</u>
-	-	a,b,c	-

(a) The initial BSs for C1 (BS_{init})

<u>W</u>	<u>X</u>	<u>Y</u>	<u>Z</u>
-	u	a,b,c	-

(b) The actual BS_{call} of C1 after propagation

Figure 6.9: Deriving actual BSs for C1 for call-by-reference for example in figure 6.6

6.3.4 Handling array variables

For the program segment such as in figure 6.2, the task is to determine if array A is modified by the procedure P1 or not. In this case, the textual array name such as A(I) will be used in the BSs. To solve this, the same method discussed in Sections 6.3.1, 6.3.2 and 6.3.3 above, will be used to form the BSs of P1 as figure 6.10 shows. It shows that all of the sets are empty since there is no global variable found in the procedure body (figure 6.10 (a)). The actual parameters of C1 will form the initial BSs of C1 as in figure 6.10(b). Figure 6.10(c) shows the final BSs of C1 after merging.

6.3.5 Limitations of the Simple-Call Algorithm

The SCA that has been described in Section 6.3.1 above poses two problems:

- (a) it is not accurate enough to determine more parallelism
- (b) the problem of handling aliasing.

For the first problem, since it assumes that for call-by-reference, the values of the actual parameters are first fetched and later written at the end of the call, it could fail to detect that the parameters passed were actually not modified at all in the body of the procedure. This may inhibit parallelism since there is no store operation involved. An example of such a call is shown in figure 6.11. The parameters passed are only referenced in the procedure body but not modified at all. The above technique will put the variables b and c in C1 as members of the Y set. Hence it will detect a data dependence between A1 and C1 but in actual case there is none since the variables should be part of the W set. However, if the call C1 is changed to p(a,b), then a data dependence should be detected due to conflicting read and write operations on the variable a.

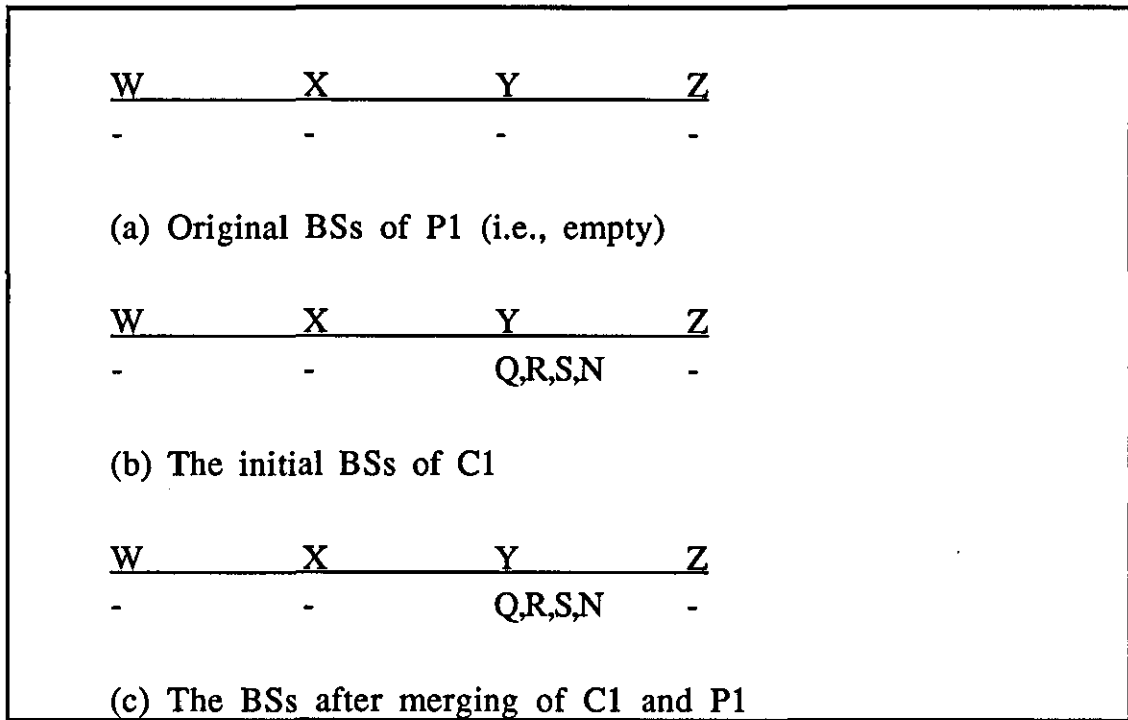


Figure 6.10: Formation of the BSs of the call C1 in figure 6.2

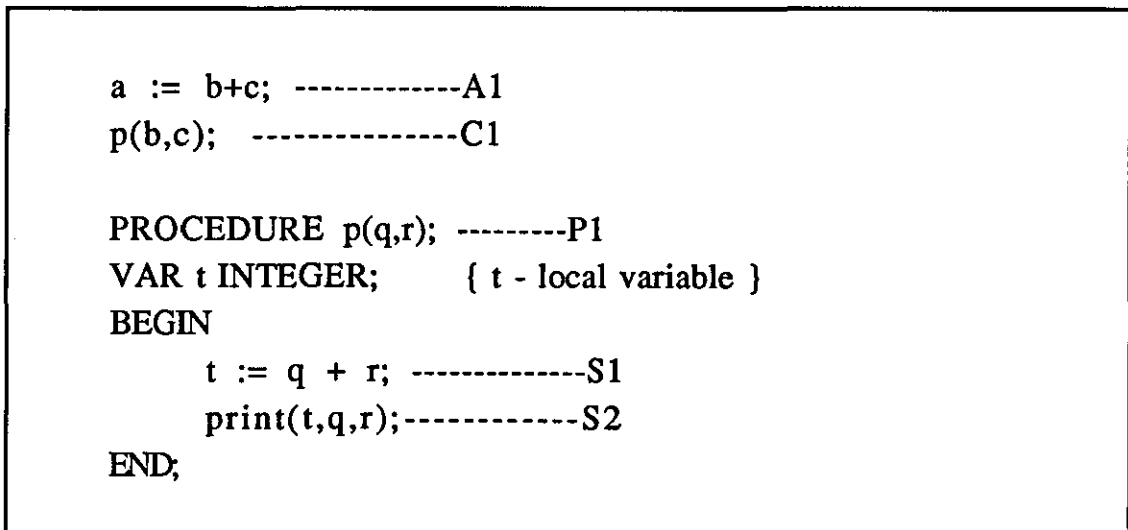


Figure 6.11: A procedure call with fetch operations only

For the second problem (also mentioned earlier in Section 6.3.3 above), duplicating actual parameters in a call (such as $p(a,a,b)$ for program in figure 6.6), presents the aliasing problem. This means that variables q and r both are referring to variable a . One way to solve this problem is that only one of the aliased variables is chosen. If the aliased names are in the same Y set, then only one of the actual duplicating variables will be used. It is enough to contribute for the dependence test. If the duplicating names appear in different sets, such as, one in the W set and the other in the Y set, then the name in the Y set is used. The one in the W set will be discarded. This is because a member of Y set contributes stronger in the dependence analysis since the Y set forms a main part in the BTs and the BLTs.

To illustrate this point, consider the program in figure 6.6 and assume that the first parameter of procedure `perform_p` is passed by call-by-name and the second and third by call-by-reference. Now consider a call `perform_p(a,a,b)`. An example of the BSs resulted from the aliasing problem for procedure `perform_p` is shown in figure 6.12.

<u>W</u>	<u>X</u>	<u>Y</u>	<u>Z</u>	
a	-	a,b	-	<----- only one a in Y set is used
(a) The BSs for C1 before propagation				
<u>W</u>	<u>X</u>	<u>Y</u>	<u>Z</u>	
-	u	a,b	-	
(b) The BSs of C1 after merging with the BSs of P1				

Figure 6.12: Call-by-reference with aliasing problem

6.4 AN IMPROVED ALGORITHM

In this section, an improved method of IPA, called the **BS-Call Algorithm (BCA)**, using the BSs will be presented to solve the problems mentioned in Section 6.3.5 above. This method will check the data flow in the body of the procedure and determine the actual BSs of the call which will indicate if any of the variables passed have been modified or not. If any modifications are made, the variables will be part of either X or Y or Z set, otherwise they will be in the W set. This is particularly important if aliasing has occurred, because it will alter the contents of the BSs. The improved algorithm will automatically handle this problem during the analysis. Call-by-reference type of parameter passing will be used in the following discussion. For call-by-name, since the actual parameters will not be modified at all, they will always be part of the W set of the call. First, a **Parameter Bind Set (PBS)** is defined.

DEFINITION 6.2

Parameter Bind Set (PBS) is a set of $\langle ap, fp \rangle$ tuples where ap is an actual parameter of the call and fp is a corresponding formal parameter defined in the called procedure.

An example of a PBS for call C1 in figure 6.11 is $\{\langle b, q \rangle, \langle c, r \rangle\}$ where b will be passed to q and c to r . The principle idea of the BCA is as follows. First, the initial BSs of the call (BS_{init}) and the BSs of each statement in the procedure body (BS_{indiv}) are formed. All actual parameters will be in W set of (BS_{init}) while (BS_{indiv}) contain all global variables and call-by-reference formal parameters. Then the actual parameters in PBS are substituted in the procedure BSs (BS_{indiv}). Finally, all of the BSs are merged to form the actual BSs of the call (BS_{call}). The steps to derive these BSs are given in figure 6.13. It assumes that there are no other procedure calls in the bodies of the procedures called. This will be discussed later in Section 6.5.

The MERGE operation in step (5) of the BCA automatically places any aliased variables in the same set, thus eliminating its potential problem. The BS_{proc} that are formed will contain any referenced-only parameters in the W set and any modified parameters in the X, Y or Z set. This is the actual information that needs to be propagated back to the calling statement and this is achieved in the last MERGE operation in step (6). Figure 6.14 shows how the actual BSs of the call in figure 6.11 is derived. The BSs in figure 6.14(e) shows that the actual parameters are members of the W set but not part of the Y set which could indicate a data dependence if analysed by the simple-call algorithm.

BS-CALL ALGORITHM (BCA)

INPUT: A call C to procedure P with actual parameter AP and formal parameters FP

OUTPUT: The BSs of the call (BS_{call})

BEGIN

- (1) $BS_{init} = FORM_BS(C)$: put AP in the W set
- (2) $BS_{indiv} = FORM_BS(P)$: BSs for procedure body
- (3) $CREATE_PBS(AP,FP)$: include only the call-by-reference parameters
- (4) $SUBSTITUTE(PBS,BS_{indiv})$: substitute each PBS into BS_{indiv}
- (5) $BS_{proc} = MERGE(All\ BS_{indiv})$: eliminating aliases
- (6) $BS_{call} = MERGE(BS_{init},BS_{proc})$: propagation

END

Figure 6.13: The BS-Call Algorithm (BCA) to derive the actual BSs of a call statement

	<u>W</u>	<u>X</u>	<u>Y</u>	<u>Z</u>
	b,c	-	-	-
(a) The BS _{init} of C1				
	<u>W</u>	<u>X</u>	<u>Y</u>	<u>Z</u>
S1:	q,r	-	-	-
S2:	q,r	-	-	-
(b) The BS _{indiv} of the 2 statements in P1				
{<b,q>,<c,r>}				
(c) The PBS for the call				
	<u>W</u>	<u>X</u>	<u>Y</u>	<u>Z</u>
S1:	b,c	-	-	-
S2:	b,c	-	-	-
(d) The BS _{indiv} of the 2 statements after PBS substitution				
	<u>W</u>	<u>X</u>	<u>Y</u>	<u>Z</u>
	b,c	-	-	-
(e) The final BSs of BS _{call}				

Figure 6.14: Finding actual BSs of the call for figure 6.11 using the BS-Call Algorithm

6.5 A GENERAL SOLUTION

The method described in Section 6.4 above, assumes that there are no calls in the procedure body. This is not always the case because a program may contain more than one procedure. The main program may have more than one call in its body and subsequently in the called procedures. In this section, the effects of calls, whether recursive or non-recursive, found in the procedure body will be studied. A general solution, which is an iterative algorithm, enhanced from the BS-Call Algorithm (BCA), will be developed to handle information in all procedures involved in the calls.

DEFINITION 6.3

Recursive Procedures are procedures which call themselves, either directly or indirectly through some other procedures that they call.

Figure 6.15 shows a direct recursive procedure and indirect recursive procedures. An iterative algorithm, which is called the **BS-Program Algorithm (BPA)**, will analyse all procedure calls found in a program. It will form sets of BSs for each procedure in terms of the actual parameters passed from any procedures. Note that the main program is considered as a procedure.

Hence, for any calls found in a procedure, the bodies of the called procedures will be analysed and the information gathered is propagated back to the caller in terms of the actual parameters of the first call. In order to handle this, a **Call Chain** is first determined.

DEFINITION 6.4

*Given a call to procedure P with actual parameter AP , a **Call Chain (CC)** is a sequence of procedures connected through calls in P and in the called procedures, each time substituting AP in the calls.*

This Call Chain (CC) can be determined by tracing the first call to a procedure, say P, and to calls in the body of P, each time substituting the actual parameters of P in a call. Figure 6.16 shows the algorithm to derive a call chain for calling procedure P with actual parameter A. The worklist chain, generated by JoinChain, is a set of $\langle P, A \rangle$ tuple of the called procedure P and with actual parameter A. For a non-recursive procedure this call chain shows a sequence of procedures starting from procedure P until a procedure, say Q, that does not have any calls (checked by NOCALL(P)), is reached. For a recursive call, the last procedure in the chain is the one which contains a call with the actual parameters already performed above in the chain. When the worklist chain does not change in its contents (checked by SAME(A)), it shows the last procedure in the call chain has been reached.

In general, the main procedure in a program will be calling the procedures defined in it and a procedure may be called from more than one calling point or site with different arguments. This will give a general structure for calls in a program called the Call Chain Tree.

DEFINITION 6.5

A CALL CHAIN TREE (CCT) is the complete call chain for a given program, starting with the main program as the root node of the tree and the procedures called as its internal nodes. The leaves are the last procedures in the call chains.

For an example of a CCT, consider a program called PROG whose main routine calls three procedures, P1, P2 and P3. The first procedure P1 calls Q which calls R which then calls S. The second procedure, P2 only calls Q. The third, P3 calls T which calls R and U. For this program, the CCT is shown in figure 6.17.

```
procedure p(a)
begin
    ...; p(i);...
end;
```

(a) Direct recursive procedure

```
procedure p(a)
begin
    ...;q(i);...
end;
```

```
procedure q(a)
begin
    ...
    if a > 10 then p(i) else print(a);
    ...
end;
```

(b) Indirect recursive procedures

Figure 6.15: Recursive and non-recursive procedures

```

procedure p(a)
begin
    ...;q(i);...
end;

procedure q(a) { no calls in this procedure }
begin
    ...
    ...
end;

```

(c) Non-recursive procedures

Figure 6.15: Recursive and non-recursive procedures (continued)

```

FIND_CALL_CHAIN(P,A)
INPUT:    A called procedure P and actual parameter A
OUTPUT:  A Call Chain(P) and maxchain, the number of
procedures in the chain
BEGIN
    if NOCALL(P) or SAME(A) then RETURN (P)
    else
        chain = JoinChain(P, Find_Call_Chain(Q,A));
END
maxchain = Length(Chain);

```

Figure 6.16: Recursive derivation of Call Chain

Starting from the procedures in the leaves of the tree, the BSs are formed and propagated upwards by traversing the reversed branches until the main program is found.

There are branches of a CCT which are the same such as Q, R, and S, called from P1 and P2, and R and S called from Q and T, as shown in figure 6.17. If the branches have the same actual parameters passed to them (for example, P1 calls Q(a) and P2 also calls Q(a)), they can be combined to form only one branch. Figure 6.18 shows the CCT with combined branches. This will save space and time to derive the BS of any call in the upper part of the tree.

After building the CCT, the BPA will analyse all the called procedures and propagate the information upwards in bottom-up fashion along the branches of the CCT. The algorithm is given in figure 6.19. In this algorithm, Find_Call_Chain (P,A) builds the CCT. FORM_BS(CCT(prog)) forms the individual BSs for statements in the procedure body. SUBSTITUTE(A,BSs) will substitute the actual parameter A in the individual BSs. The first MERGE operation in that algorithm merges all individual BSs of a procedure while the second MERGE propagates the variable information upwards. The diagram in figure 6.20 illustrates the whole picture in deriving the BSs for a procedure P which has a call chain to Q, R and S. The initial BSs of any call found in a procedure will have the actual parameters in the W set.

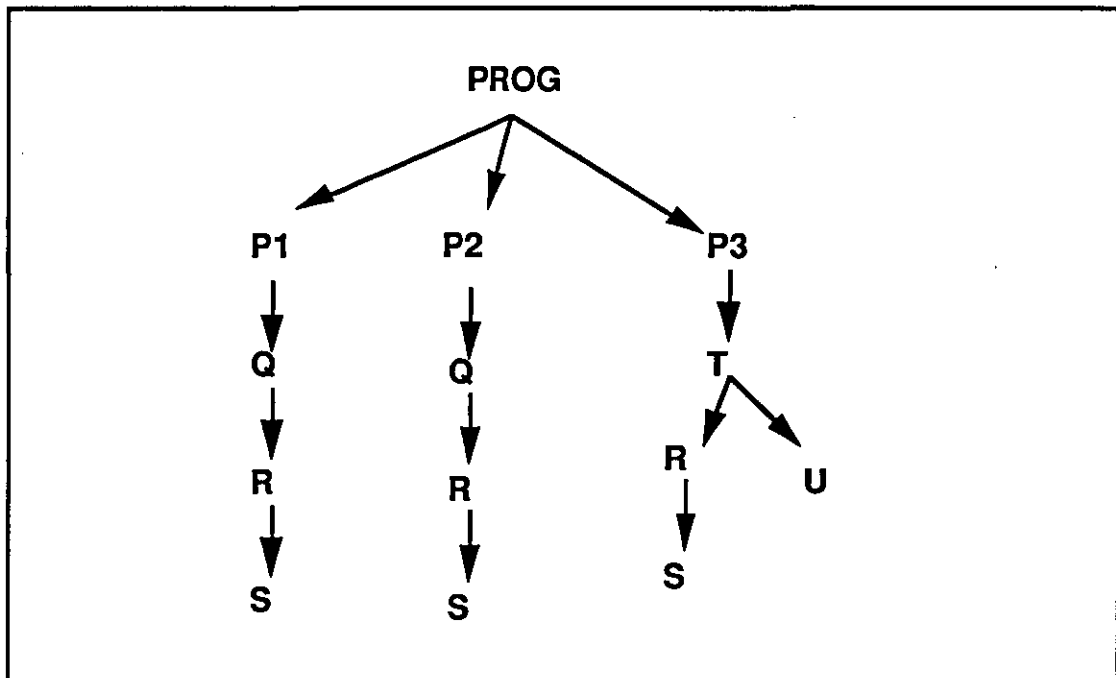


Figure 6.17: A Call Chain Tree (CCT)

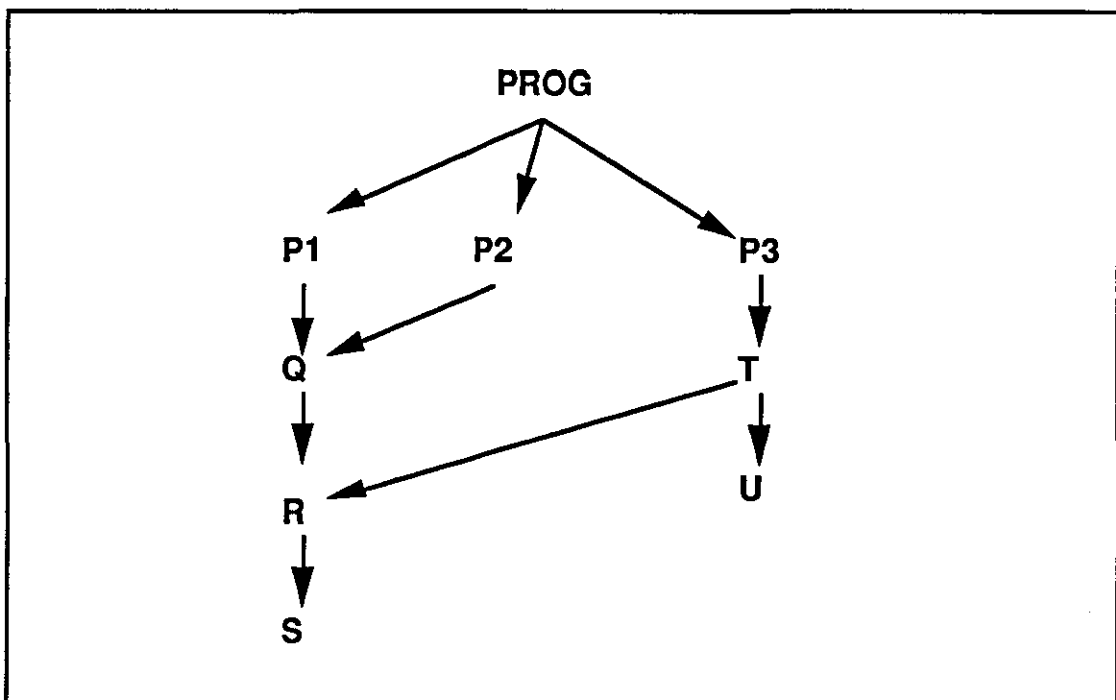


Figure 6.18: Combined branches of CCT

BS-PROGRAM ALGORITHM

INPUT: A program PROG with a set of calls and a set of procedures Ps

OUTPUT: Sets of BSs for PROG and Ps

BEGIN

FOR each call to P in PROG

BEGIN

(1) CCT(PROG) = FIND_CALL_CHAIN(P,A)

(2) BS_{indiv}(i) = FORM_BS(CCT(PROG)),
1 ≤ i ≤ maxchain (maxchain derived in (1))

(3) SUBSTITUTE(A,BS_{indiv}(i)), 1 ≤ i ≤ maxchain

(4) FOR ch = (maxchain-1) DOWNT0 1

BEGIN

(4.1) BS_{proc}(ch+1) = MERGE(All BS_{indiv}(ch+1))

(4.2) MERGE(BS_{indiv}(ch,i),BS_{proc}(ch+1)):
propagate BS_{proc}(ch+1) by merging
BS_{indiv}(ch,i) (i.e., the i-th BS_{indiv}(ch) that
contains the call) with BS_{proc}(ch+1))

END

END

END

Figure 6.19: BS-Program Algorithm (BPA)

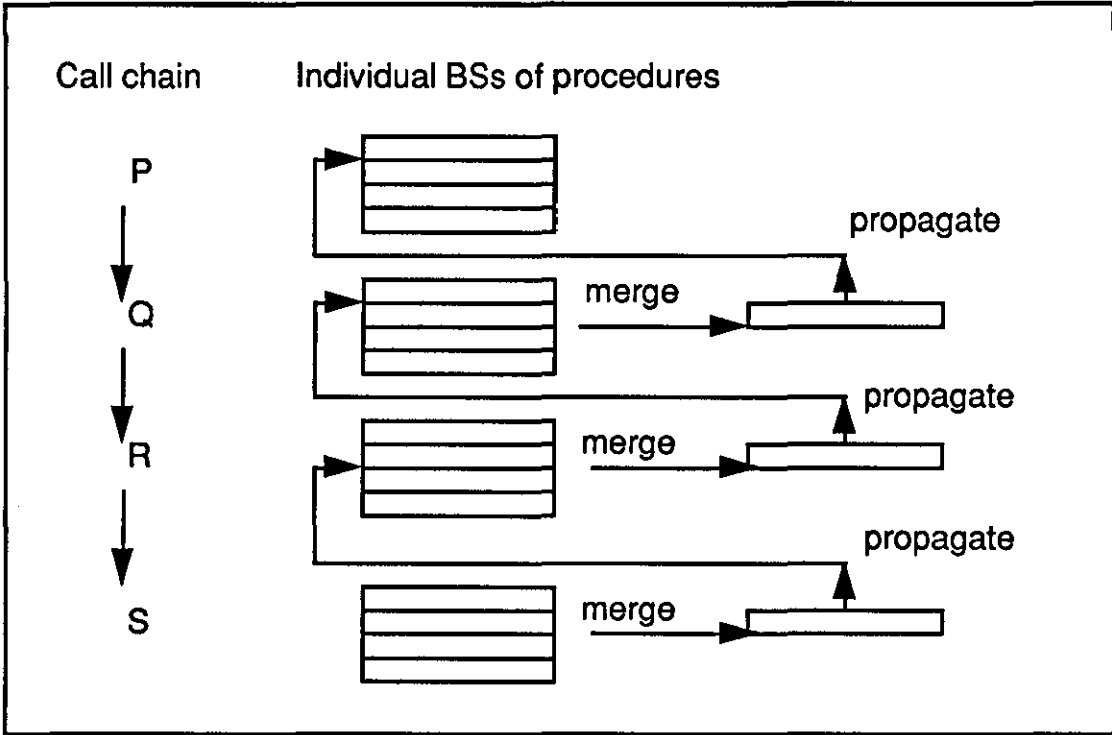


Figure 6.20: Example of call chain and derivation of BSs

6.6 EXAMPLE OF SOLUTIONS

In this section examples will be given to demonstrate the application of IPA using the Bernstein Sets.

Example 6.1: Non-recursive procedures

```
Program Ex1
Global variable: x, y;
main program;
begin
    read(i,j);
    P(i,j);
end;

procedure P(a,b);
begin
    Q(a); x := a; a:=10;
end;

procedure Q(b);
begin
    b:= b/100; R(x); y := x;
end;

procedure R(c);
begin
    print(c);
end;
```

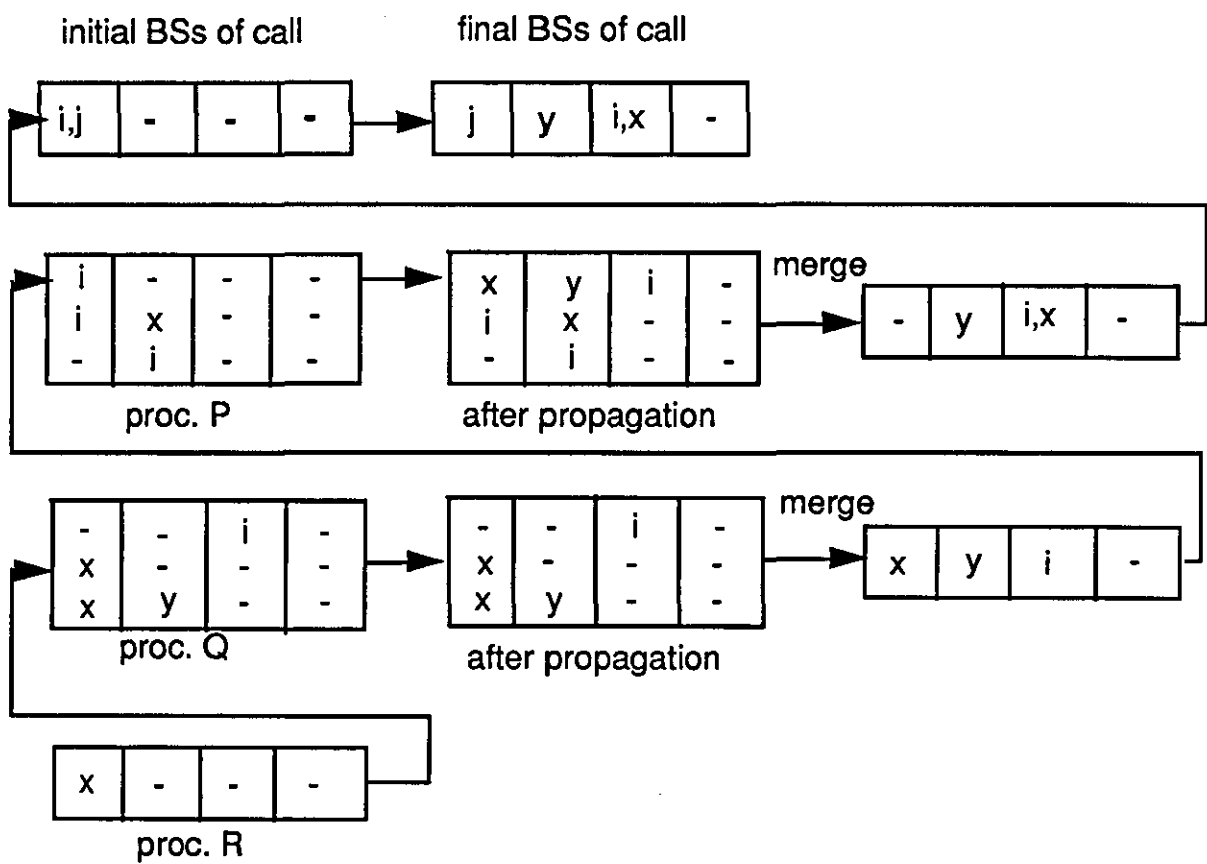
By tracing the call, the following sequence of call chain is formed.

call P(i,j) -> P calls Q(i) -> Q call R(x) -> R

The sequence stops at R since it does not have any other call in its body. Thus, the call chain derived for the calls in the program Ex1 is as follows.

main call ---> P ---> Q ---> R

For each procedure in the call chain, the BSs are formed. Then the merging and propagating of information in the BSs are carried out to determine the actual BSs of the main call. The whole process is depicted as follows. Note that the four columns represent W, X, Y and Z sets.



Example 6.2: Direct recursive procedure.

Consider a call $P(x,y,i)$ to the following recursive procedure which is called directly. Variables x and y are global.

```
procedure P(a,b,c)
begin
    b := x; if (..) then P(a,b,c); ... := y;
end;
```

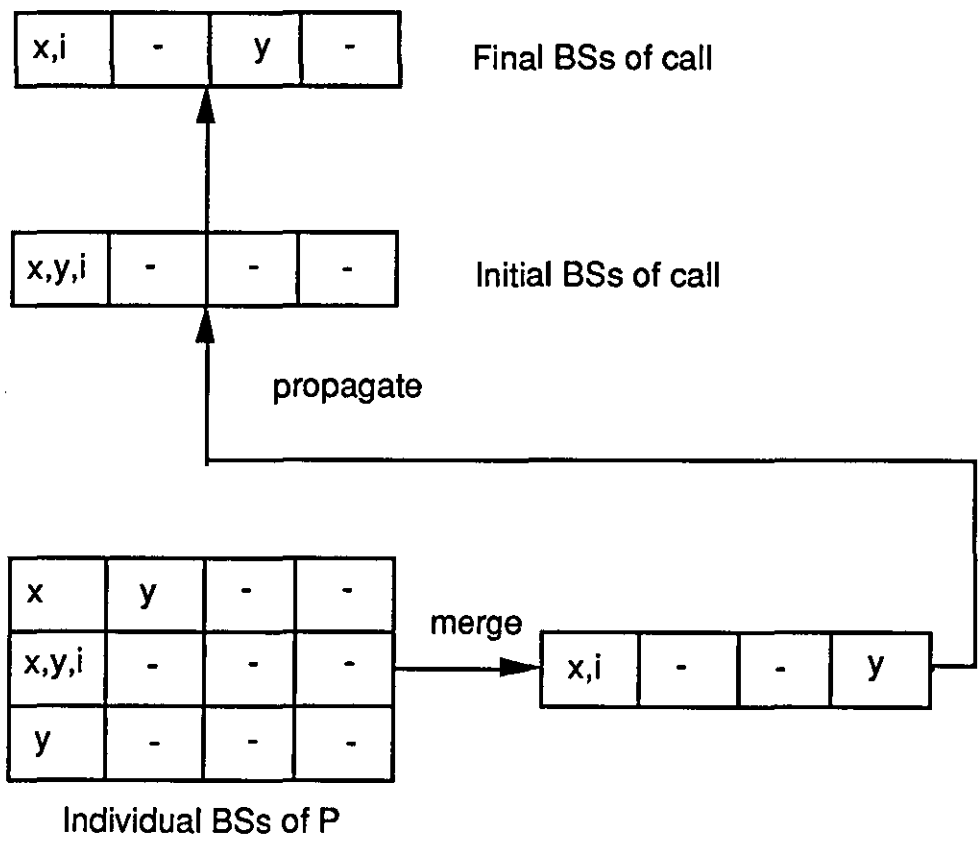
The call trace is as follows.

main calls $P(x,y,i)$ -> P calls $P(x,y,i)$

thus giving the call chain:

main call ---> P

The formation of the BSs is shown in the diagram below.



Example 6.3: Indirect recursive procedures

Consider the indirect recursive procedures below and a call P(a,b,c).

```
procedure P(a,b,c)
begin
    Q(a,b,c);
end;

procedure Q(x,y,z)
begin
    x := ...;
    ... := y;
    if (...) then P(x,y,g);    { g is a global variable }
end;
```

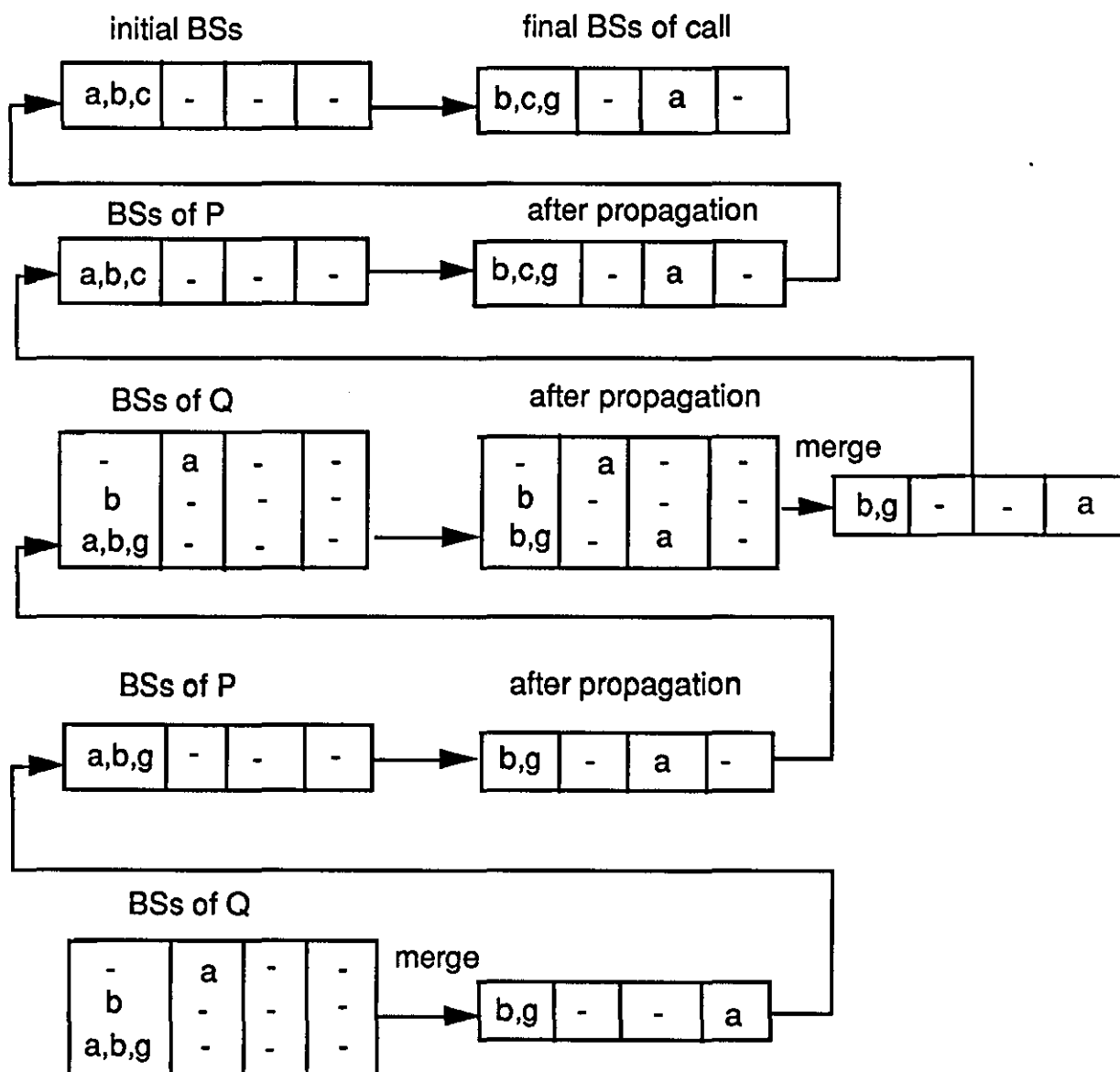
By tracing the calls, there is a repeating call in the last procedure Q which calls P(a,b,g). Hence, that will be the end of the call chain.

Call P(a,b,c) -> P calls Q(a,b,c) -> Q calls P(a,b,g) ->
P calls Q(a,b,g) -> Q calls P(a,b,g)

The call chain is as follows.

main call ---> P ---> Q ---> P ---> Q

The formation of the final BSs of the call is shown below.



6.7 PROCEDURE CALLS IN LOOPS

As stated in Chapter 5, loops have been a major topic in the research in parallelization of programs. They provide the best opportunities for parallelism where each iteration in the loops may be executed in parallel. [Allen and Kennedy (1984a, 1984b, 1987), Banerjee (1988), Li (1989), Mohd-Saman and Evans (1993), Wolfe (1986, 1988, 1989b), Wolfe and Banerjee (1987)]. The presence of procedure calls within a loop body makes the dependence analysis of the loop more difficult. One of the main criticism on some commercial compilers/vectorizers is that they do not vectorize all loops containing procedure calls in them [Blume and Eigenmann (1992), Eigenmann and Blume(1991)].

Figure 6.21 shows a loop L1 with a procedure call C1 to procedure P1. The problem is to ascertain whether the loop iterations of L1 can be executed concurrently or not. If information on the usage of the shared variables in P1 is not available, then the compiler/restructurer has to assume that data dependence exists between loop iterations. If scalar variables are involved as in figure 6.21, the BS-Call Algorithm (BCA) described above is sufficient to determine the BSs of the call. On the other hand, for array variables, they present a problem on how to decide on the reference directions before the BLTs can be applied.

6.7.1 Handling array variables

The BLTs discussed in Chapter 5 will detect loop parallelism containing array references but it assumes that the loops do not contain any procedure calls. The tests use the Data Reference Directions (DRDs) (=, > and <) to indicate the pattern of array references in the loops. Hence, the existence of procedure calls to procedures with array variables needs to be carefully analysed to include the use of the DRDs in the procedure body. Consider the program segment in figure 6.22 which has a loop containing a procedure call with the manipulation of array elements. Assume that the parameter passing is call-by-reference. The task is to

determine whether the loops L1 and L2 are parallelizable or not. The method described in Section 6.5 will be used but in forming the BSs of the call to a procedure which contains array variables, DRDs will be included.

```

FOR i := 1 to 10 DO-----L1
BEGIN
    p(a,b,c); -----C1
END;
PROCEDURE p(q,r,s); -----P1
VAR t INTEGER;      { t - local variable, u global variable }
BEGIN
    t := r + q;
    s := t;
    u := s;
END;

```

Figure 6.21: A loop with a procedure call

```

FOR I := 1 TO N DO-----L1
FOR J := 1 TO N DO-----L2
BEGIN
    A[I,J] := B[I,J] + C[I,J]; -----S1
    P(A,B,C,I); -----C1
END

PROCEDURE P(Q,R,S,L); -----P1
VAR INTEGER K;
BEGIN
    FOR K := 1 TO N DO
        Q[L,K] := R[L,K] + S[L,K];
    END;

```

Figure 6.22: A loop with array variables

The general strategy in forming the BSs of the call is the same as the one discussed in ^{the}previous section. However, since array names are involved, the formation of the BSs is done as follows.

FORM_BS(procedure with array):

Form the BSs for each statement in procedure P to include all the array variables. A '*' or '<' or '>' direction will be used for the direction of the array variables if the array is referenced or modified over a different range of indices. Otherwise the '=' direction is used as part of the array name. '*' is used if more than one array element is being modified or referenced.

Figure 6.23 shows the contents of the BSs of C1 for the loop in figure 6.22. The resulting BSs of C1 shown in figure 6.23(d) can then be used in the BLTs which should show that the outer I loop is parallelizable but not the inner J loop.

There are several other cases that have to be considered in determining the direction of an array name when forming the BSs of procedures. If a whole array is passed during a call, the directions formed are the same as shown in figure 6.23. In another situation, if the same call is made but the body of the procedure contains loop statements such as in the following example:

```

for L := 1 to n do
  for K := 1 to n do
    Q[L,K] := R[L,K] + S[L,K];
  
```

then the arrays with directions are Q[*,*], R[*,*] and S[*,*]. This is because Q, R and S are being fetched/stored over a range of values. If there are no loops, then the arrays with directions are Q[=,=], R[=,=] and S[=,=].

If the body of the procedure contains the following loop:

```
for K := 1 to n do
    Q[L+const,K] := ...;
```

then the directions of Q will be Q[<,*]. On the other hand, if the array reference is of the form Q[L-const,K], then the directions are Q[>,*]. If a single array element is passed such as in the following call:

```
call P(a[i,j],...);
```

W	X	Y	Z
B[=,=]	A[=,=]	-	-
C[=,=]			
(a) The BSs of S1			
W	X	Y	Z
R[=,*]	Q[=,*]	-	-
S[=,*]			
(b) The BSs of P1			
{<A,Q>,<B,R>,<C,S>,<I,L>}			
(c) The PBS of the call			
W	X	Y	Z
B[=,*]	A[=,*]	-	-
C[=,*]			
(d) The BSs of C1 after parameter replacement of P1			

Figure 6.23: Dependences with array references of the loop in figure 6.22

then it will be treated as a single variable and the array with directions will be of the form $a[=,=]$.

The examples above can be extended to any array of any dimension. Once the BSs of the call ~~have~~ been derived, the BLTs can be applied to determine the loop parallelism.

6.8 SUMMARY

Most compilers which perform global optimization face problems when they encounter procedure calls. They do not have enough information on whether any global variables are being modified or not by the called procedures. The main problem caused by procedures in programs is that they may introduce aliasing of variables. Thus, conservative assumptions are usually made, i.e., data dependence exists and no parallelization is carried out. This chapter has discussed the needs for Inter-procedural Analysis (IPA) to be carried out as part of the dependence analysis. It analyses the usage of variables in procedure bodies when calls are made.

The Bernstein Sets (BSs) have been shown to provide a good and efficient way to handle inter-procedural information which can then be used by the BTs and the BLTs. The strategy proposed in this chapter shows how to capture the inter-procedural information as part of the contents of the BSs. It involves deriving the initial BSs of the calls which are later merged with the BSs of the statements in the procedure body to form the actual BSs of the call. This method can also handle direct or indirect recursive calls. A call chain is first determined and then the BSs are formed from the procedures in the chain.

One main advantage of using the BSs as shown in this chapter is that no modification is needed either on the BTs and BLTs or on how the BSs are formed. No extra information needs to be stored in the BSs except the variables that are involved in the programs.

CHAPTER 7

VERIFICATION OF PARALLEL PROGRAMS

7.1 INTRODUCTION

One of the main and fundamental objectives in writing computer programs is ensuring their reliability so that they are free from any errors and bugs. The programmer should take every precaution that the program written will at least provide safe operations. This is particularly important for systems with critical applications such as the aircraft flight control and the nuclear reactor safety which demand high safety and reliability requirements [Leveson (1986), Moser and Melliar-Smith (1990)]. It has been recognized that the proofs for program correctness are of great importance as an attempt to achieve error-free programs. To prove the correctness of programs means that the programs are verified to behave as they were intended to do [Dijkstra (1976)].

With the advent of parallel machines, there is a greater need to make sure that parallel programs also behave reliably. The results of parallel programs can be very unpredictable as many processes are executed at the same time. This will greatly enhance the probability of a programmer to make mistakes.

This chapter deals with methods to prove the correctness of parallel programs where they present problems that are not found in sequential programs. This is because more than one part of the programs can be executed concurrently. The organization of this chapter is as follows. Section 7.2 briefly explains the various techniques used for program verification. In Section 7.3, the Symbolic Execution (SE) method for verifying the correctness of programs is presented. Then an overview of the work by other researchers is presented in Section 7.4. In Section 7.5, a method of proving the correctness of parallel parts of programs called stanzas, based on the Bernstein Tests developed by Williams (1978), is presented, followed by Section 7.6 which discusses a method of verifying the correctness of fixed parallel loops based on the Bernstein Loop Tests (BLTs). Section 7.7 summarises the whole chapter.

7.2 METHODS FOR PROVING PROGRAMS

One way to obtain the confidence in one's program is to test it with several small test data. This is called **program testing** [Cherniavsky and Smith (1986), Frankl and Weyuker (1986), King (1976), Sneed (1986)]. A proper choice of sample data is critical in this method so as to ensure that the program will operate correctly and safely over some domain of inputs. However, this method does not ensure complete correctness over all sets of data. Another method that has been proposed is **program proving** [Hantler and King (1976), Hoare (1969), King (1976), Owicki and Gries (1976)]. This allows a programmer to prove, formally, that a program meets its requirements or specifications for all executions without having to be executed at all. This is done by giving some exact specifications of the intended behaviour of the program and these specifications will be verified for their consistency. This method lets the programmer verify a program over wider ranges of the intended data.

In this chapter, the concept of a verification method called the **Symbolic Execution (SE)** [Hantler and King (1976), King (1976), Young and Taylor (1986)] will be extended to verify parallel programs. The main idea is that, apart from verifying the correctness of each individual program in sequential manner, other forms of correctness properties that are inherent in parallel systems, such as mutual exclusion and freedom from modifying shared variables, will have to be proved. This will be solved by introducing new assertions, called the **BT Assertion** and the **BLT Assertion**. This chapter assumes a general programming language such as Pascal extended with parallel constructs such as Lock and Unlock statements.

Several researchers have been working on proving the correctness of sequential programs [Clarke and Richardson (1984), Hantler and King (1976), Kemmerer and Eckman (1985), King (1976)]. There are also researchers who have been dealing with the correctness of parallel programs such as ADA and CSP

[Apt (1986), Dillon (1988, 1990), Dillon et al. (1988), Guaspari et al. (1990)], Misra and Chandy (1981), Owicki and Gries (1976), Gries (1977), Williams (1978)]. An overview of the different techniques for proving the correctness of programs will be given and they are divided into four approaches:

- (a) the SE approach
- (b) the stanza approach,
- (c) the axiomatic approach and
- (d) formal methods.

7.3 SYMBOLIC EXECUTION (SE)

In program proving, one method that has been used is the **Symbolic Execution (SE)**. It has been shown to be a useful and successful approach in verifying the correctness of sequential and parallel programs [Clarke and Richardson (1984), Cohen et al. (1982), Dillon (1988, 1990), Dillon et al. (1988), Hantler and King (1976), Howden (1977), Kemmerer and Eckman (1985), King (1976)]. In this method, the intended behaviour of the program is specified in terms of **correctness assertions** or simply assertions. Sometimes they are called the specification of the program. These assertions are usually in the form of predicate logic. By using this approach, algebraic symbols are used to represent input values of a program. These symbols are then manipulated by the program to derive some logical expressions. These expressions are then checked against the restrictions on the values of the variables in the program, written in the form of assertions, to deduce the program correctness.

DEFINITIONS 7.1

- (i) *A procedure is said to be correct (with respect to its input and output assertions) if the truth of its input assertion upon procedure entry ensures the truth of its output assertions upon procedure exit.*

- (ii) *An Input (or entry) Assertion specifies assumptions of the values of the variables when the program is invoked.*
- (iii) *An Output (or exit) Assertion specifies the intended behaviour of the program when it reaches some later or final stage, i.e., when it terminates.*
- (iv) *Loop Assertion (or Loop Invariant or Inductive Assertion) specifies the condition of execution for conditional loops.*

When a program is executed by giving some symbolic values for its inputs, symbolic expressions representing the values of the variables encountered are formed. **Path Conditions (PCs)** (or **Verification Conditions (VCs)**) are also generated. These PCs are used in deciding which path to take when conditional statements are met. At the end of the execution the values of PCs (or VCs) are then verified against the assertions provided in the program. The type of correctness to be established is called **partial correctness**, i.e., a program or part of a program is said to be correct if the truth of its entry ensures the truth of its exit and there is no guarantee of termination [Hantler and King (1976), Williams (1978)].

The SE is useful in program testing, debugging and verification and has several advantages. It may represent a large class of normal execution. If normal numerical values are used, it will only show an instance of the execution for some specific data. Another advantage is that the symbolic expressions that are formed for each variable encountered, can show the relationship each one has with other variables and its environment. If specific testing is needed for specific input values at any stage, then the numerical values need only to be substituted in the symbolic expressions to derive their actual values. This technique can also easily be automated by designing a software tool, called **Symbolic Executor** [Dillon (1988), Harrison and Kemmerer (1988)]. SE has been used as the main strategy in some automated systems to verify the correctness of sequential programs such as Dissect

system [Howden (1977)], Effigy system [King (1976)], SELECT [Boyer et al. (1975)] and Unixex system [Kemmerer and Eckman (1985)].

As discussed in Hantler and King (1976), to prove the correctness of programs, input and output assertions are provided as part of the programs. The input assertion is a statement of the form **ASSUME** (<expression>) and usually appears on an entry of a procedure. The output assertion is a statement of the form **PROVE** (<expression>) and appears immediately before the **RETURN** statement of a procedure. For the loop assertion, the statement is of the form **ASSERT**(<expression>). An example of a simple procedure with input and output assertions, slightly modified from Hantler and King (1976), is shown in figure 7.1.

```
1  procedure ABS(X);
2  var X,Y : integer;
3  begin
4      ASSUME (true);
5      if X < 0 then
6          Y := -X;
7      else Y := X;
8      PROVE ((Y = X' or Y = -X') and
9             Y >= 0 and X = X');
9      return (Y);
10 end;
```

Figure 7.1: Procedure ABSOLUTE with correctness assertions

To verify the correctness of procedure ABS in figure 7.1, during the SE, the input variable X is given a symbol (such as val1 or val2, etc) and then one hopes to obtain logical expressions over the input symbols as the values of the output symbols. These results are then checked against the output assertions (such as the one stated in the PROVE assertion in line 8 of the example) to deduce its correctness or incorrectness. When the SE is

performed, the state of program execution is maintained. This consists of the values of the variables in symbolic form, the statement counter showing the next statement to be executed and a path condition (PC) which describes the conditions when conditional statements are found. A symbolic execution tree may also be generated showing the graphical representation of the SE. It will lead to results whether the program is verified correct or not. To verify the correctness of loops in programs, the loop assertion can be inserted and checked. Figure 7.2 shows an example showing the use of a loop assertion.

```

procedure GCD(m,n)
var m,n,a,b: integer;
begin
    ASSUME(m>0 and n>0);
    a := m;
    b := n;
    while (a <> b) do
    begin
        ASSERT((a,b) = (m,n) and a <> b);
        if (a>b) then
            a := a - b
        else b := b - a;
    end;
    PROVE (a = (m,n));
    return(a);
end;

{ Note:  $Cp,q)=r$  means  $r$  is the GCD of  $p$  and  $q$ . }

```

Figure 7.2: Procedure with loop assertion

Symbolic Evaluation as described in Clark and Richardson (1984), is an SE technique for software testing. It provides path selection and test data selection based on the symbolic values of PCs. Ploedereder (1984) has also described the use of Symbolic

Evaluation in deriving information about the static and dynamic semantics of programs. This information is then stored in a program data base to be used by tools supporting program development and validation. Examples of such tools are assignment set/use analysis, dynamic lifetime analysis, analysis of aliasing and verification of program correctness.

Dillon (1988, 1990), Dillon et al. (1988), Guaspari et al. (1990) and Harrison and Kemmerer (1988) have presented several ideas for proving ADA Tasking based on the SE. The central issues involved in proving ADA programs are the concurrent tasks and the handling of communication through rendezvous. The ideas developed by Dillon, Harrison and Kemmerer are based on the SE with the interleaving approach and isolation approach. Guaspari et al. (1990) have developed an automated system called the Penelope verification editor, a prototype system for an interactive development and verification of ADA programs. Using the same strategy as in the SE approach, VCs are generated by the system and the verification is carried out incrementally. Penelope uses a specification language called Larch/ADA Specification Language to describe the specification of ADA subprograms and packages (as annotations). VCs generated are verified by the user within Penelope.

Good et al. (1979) have adopted a different way of keeping information about the values of the program that can be used for verification. It uses a method called message buffers as the sole process coordination. They have developed a language called Gypsy which allows concurrent processing. All inter-process communication is via these message buffers. A complete history of process interaction, based on the information from message buffers, is kept to be analysed during verification. Gypsy allows the programmers to place assertions in the program for proving its correctness. These assertions refer to message buffers and the respective transaction history of the program.

7.4 OTHER RELATED WORK

Apart from the SE technique, other approaches have been taken to analyse program correctness.

7.4.1 The stanza approach

Williams (1978) has developed several relationships to test for parallelism between blocks of statements called stanzas. She has also discussed the correctness of parallel stanzas using the technique of SE to show the behaviour of the stanzas based on the new relationships. The following terms are used in conjunction with proving the correctness of parallel stanzas.

DEFINITIONS 7.2

- (i) *antecedent* - a condition expected to be true on entry to a program.
- (ii) *consequent* - a condition expected to be true when a program ends.
- (iii) *symbolic execution networks* - a method to analyse the symbolic execution of parallel programs or stanzas which allow variables to be accessed by more than one stanza.

Williams has described the proofs for correctness of a parallel program written explicitly using the relationships she has developed. The technique of SE is used to indicate how the correctness of programs using the new relationships may be proved. The correctness of parallel stanzas for a machine with shared memory and one with a private memory has been proved by developing an execution tree network.

7.4.2 The axiomatic approach

Owicki and Gries (1976) and Gries (1977) have discussed a method to verify properties of parallel programs. They base their work on the axiomatic approach developed by Hoare (1969) who has formulated a set of axioms for partial correctness of programs. The axioms give the meanings of program statements in terms of assertions about variables in the programs.

The notation $\{P\} S \{Q\}$, which informally means: if P is true before execution of S , then Q will be true when S terminates, is called a statement of partial correctness; where termination of S must be established by other means. P is called the **precondition** and Q the **postcondition**. They are the assertions inserted in the programs. Misra and Chandy (1981) have described a similar solution written for CSP programs. The termination problem of parallel programs has been addressed by Apt (1986).

Owicki and Gries have defined a deductive system to offer a better approach in proving the correctness of parallel programs. They use auxiliary variables added to a parallel program as an aid to prove that it is correct. With this technique, the properties of parallel programs such as the mutual exclusion, freedom from deadlock and program termination can be proved to behave correctly. To prove these, an assertion $I(r)$, the **invariant for resource r** , is introduced. $I(r)$ must be true when parallel execution begins and remain true at all times outside the critical section for r . They have defined two axioms: parallel execution axiom and critical section axiom:

a. Parallel Execution Axiom

If $\{P_1\} S_1 \{Q_1\}$ and $\{P_2\} S_2 \{Q_2\}$ and ... and $\{P_n\} S_n \{Q_n\}$ and no variable free in P_i or Q_i is changed in S_j ($i \neq j$) and all variables in $I(r)$ belong to resource r , then:

$$\{P_1 \wedge P_2 \wedge \dots \wedge P_n \wedge I(r)\}$$

resource r: cobegin S1 || S2 || ... || Sn coend

$$\{Q_1 \wedge Q_2 \wedge \dots \wedge Q_n \wedge I(r)\}$$

b. Critical Section Axiom

If $\{I(r) \wedge P \wedge B\} S \{I(r) \wedge Q\}$ and $I(r)$ is the invariant from the cobegin statement, and no variable free in P or Q is changed in another process, then $\{P\}$ with r when B do $S \{Q\}$

7.4.3 Formal methods

Formal methods are mainly used in the development of programs [Boiten et al. (1992), Bowen (1988), Ehrig et al. (1992a, 1992b), Hall (1988), Masterson et al. (1988), McParland and Kilpatrick (1988), Wordsworth (1988)]. The main idea is to develop a formal specification of a problem before attempting to write the program. Once these specifications have been developed, they can be verified to ensure their correctness. Then they are transformed into correct programs. This is claimed to lead to a more accurate solution. The Vienna Development Method (VDM) and the Z specification language have been the most widely used methods to form specifications of programs.

7.5 VERIFYING PARALLEL STANZAS

Williams (1978) has defined a stanza as a group of one or more statements that has to be executed sequentially. She has also developed a set of relationships between stanzas and a set of tests to determine parallelism between two or more stanzas. Based on this idea, this chapter shows how the tests can be adapted in the SE technique to verify parallel stanzas.

In order to perform the verification, the problem will be tackled in two separate ways. First, the correctness of each stanza in a sequential manner is determined. In this case, it can be verified by using the sequential SE model as outlined in Section 7.3 above. Second, the correctness of the parallel stanzas is

determined by checking the consistency of their parallel properties. This involves ensuring that any shared variables are not being fetched and stored at the same time by the concurrent stanzas. Any conflicting fetch and store operations may cause the shared variables to be undefined if not properly synchronized.

For the sequential verification, the programmer places input, output and loop assertions which will be verified during the SE. Formation of Boolean assertions for programs has been discussed thoroughly in Gries (1981). On the other hand, to verify the parallel properties of the stanzas, their BSs are first formed. The BT Assertion and BLT Assertion will be formulated to be part of the verification scheme, that can be used with the BSs.

7.5.1 The BT Assertion

Williams has defined that two or more stanzas are contemporary (or concurrent), i.e., they can be executed in parallel, if the application of the Bernstein Tests (BTs) is successful. The BTs consist of the following tests which should give empty results to indicate that data dependence does not exist:

- a. $WYZ_i \cap XYZ_j = \emptyset$ for all $1 \leq i, j \leq n, i \neq j$
- b. $XYZ_i \cap XYZ_j = \emptyset$ for all $1 \leq i, j \leq n, i < j$

In this section, the BTs will be used as an assertion in order to verify the parallel property between two or more stanzas. It will be called the **Bernstein Test Assertion** or simply the **B T Assertion (BTA)**. It will be as follows:

PROVE(BTA).

The BTA is of the following form, where there are n stanzas i and j ($i \neq j$).

$$\text{BTA:} \quad ((WYZ_i \cap XYZ_j) = \emptyset) \wedge ((XYZ_i \cap XYZ_j) = \emptyset)$$

During the course of the SE, the BSs can be determined for the stanzas. Hence, by combining the input and output assertions together with the BTA, any two or more parallel stanzas in a straight line code can be proved for their correctness. The model for this kind of proof is shown in figure 7.3. It will show, if there exists or not, any conflicting fetch and store operations that can cause indefiniteness of results. If the BTA gives non-empty results, then it can be concluded that the parallel execution of the two stanzas is not correct. Figure 7.4 and figure 7.5 show two examples of parallel stanzas and their BSs. The BTA will verify that the parallel property of the two stanzas is preserved because there are no shared variables involved.

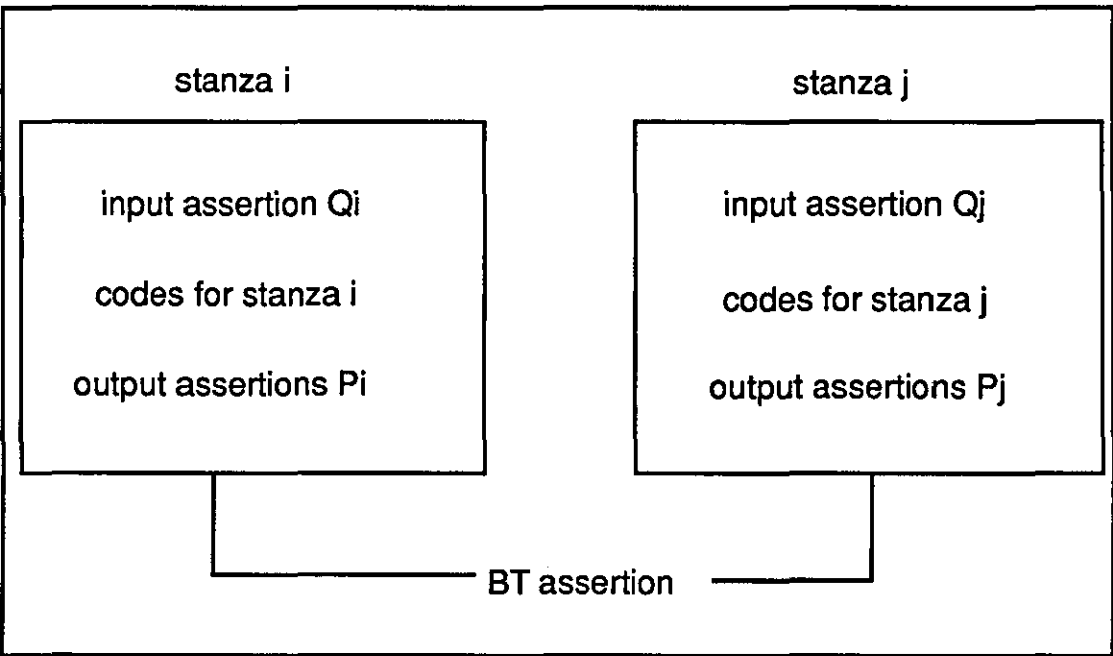


Figure 7.3: Model for verifying the correctness of parallel stanzas using the BT Assertion

7.5.2 Critical Sections

The correctness of parallel stanzas, when critical sections are included, poses another problem. Critical sections are implementations of the parallel property called the mutual exclusion. They will ensure that the stanzas are executed sequentially to preserve the correctness of the values of shared variables. In this case, another set, called the Shared Variable

Set (SVS) will be introduced. An SVS contains all variables that are found in a critical section of a stanza. Since the critical section has eliminated the problem of concurrent fetch and store, the members of SVSs will be removed from the WYZ and the XYZ sets in the BTA. This will ensure the correctness of the parallel stanzas. Hence, the assertion is modified as follows.

$$\begin{aligned} \text{BTA:} \quad & (((\text{WYZ}_i - \text{SVS}_i) \cap (\text{XYZ}_j - \text{SVS}_j)) = \emptyset) \wedge \\ & (((\text{XYZ}_i - \text{SVS}_i) \cap (\text{XYZ}_j - \text{SVS}_j)) = \emptyset) \end{aligned}$$

Figure 7.6 shows two parallel stanzas with critical sections. The codes in between the LOCK and UNLOCK regions are the critical regions. Only one stanza can execute its own critical region at one time. When the SVS is deleted from the XYZ and WYZ sets, the BTA will show that there is no concurrent fetch and store operations.

```

procedure ROOT(a,b,c)
var  a,b,c,t1,t2,t3,t4 : integer;
    r1, r2 : real;
begin
  ASSUME(a>0);
  if (a > 0) then
  begin
    t1 :=  $\frac{-b}{2a}$ ;
    t2 := b * b;
    t3 := 4 * a * c;
    t4 := 2 * a;
    if t2 > t3 then
    begin
      r1 := (t1 + SQR(t2-t3))/t4;
      r2 := (t1 - SQR(t2-t3))/t4;
    end;
  end;
  PROVE ((a > 0 and (b*b>4*a*c)) or
        (a > 0 and (b*b<4*a*c)) or
        (a <= 0))
end of ROOT;

```

(a) Root program

Figure 7.4: Examples of parallel stanzas

```

procedure REM(x,y)
var x,y,r,q:integer;
begin
    ASSUME(0<=x and 0<y);
    r := x;
    q := 0;
    ASSERT((x=y*q+r) and (r>=0) and (y>0));
    while r >= y do
        begin
            r := r - y;
            q := q + 1;
        end;
    PROVE(0 <= r < y and x = y*q+r);
end of REM;

    (b) Remainder program

```

Figure 7.4: Examples of parallel stanzas (continued)

W	X	Y	Z	W	X	Y	Z
-----				-----			
a	r 1	-	t 1	x	-	-	r
b	r 2		t 2	y			q
c			t 3	-----			
			t 4				

(c) BSs for ROOT				(d) BSs for REM			

Figure 7.5: The Bernstein Sets for programs in Fig. 7.4.

ASSUME(true); max:=0; LOCK; read(a); if (a > 0 and a > max) then max := a; print(a,max); UNLOCK; PROVE(max>=0)	ASSUME(true); min:=999; LOCK; read(a); if (a < min) then min := a; print(a,min); UNLOCK; PROVE(min <= 999)																														
(a) Stanza 1	(b) Stanza 2																														
<table> <tr> <th><u>W</u></th> <th><u>X</u></th> <th><u>Y</u></th> <th><u>Z</u></th> <th><u>SVS</u></th> </tr> <tr> <td>-</td> <td>-</td> <td>max</td> <td>-</td> <td>a</td> </tr> <tr> <td></td> <td></td> <td>a</td> <td></td> <td>max</td> </tr> </table>	<u>W</u>	<u>X</u>	<u>Y</u>	<u>Z</u>	<u>SVS</u>	-	-	max	-	a			a		max	<table> <tr> <th><u>W</u></th> <th><u>X</u></th> <th><u>Y</u></th> <th><u>Z</u></th> <th><u>SVS</u></th> </tr> <tr> <td>-</td> <td>-</td> <td>min</td> <td>-</td> <td>a</td> </tr> <tr> <td></td> <td></td> <td>a</td> <td></td> <td>min</td> </tr> </table>	<u>W</u>	<u>X</u>	<u>Y</u>	<u>Z</u>	<u>SVS</u>	-	-	min	-	a			a		min
<u>W</u>	<u>X</u>	<u>Y</u>	<u>Z</u>	<u>SVS</u>																											
-	-	max	-	a																											
		a		max																											
<u>W</u>	<u>X</u>	<u>Y</u>	<u>Z</u>	<u>SVS</u>																											
-	-	min	-	a																											
		a		min																											
(c) BSs for stanza 1	(d) BSs for stanza 2																														

Figure 7.6: Stanzas with critical sections

7.6 VERIFYING PARALLEL LOOPS

Detection of parallelism in loops has been a major research topic reported in several papers [Allen et al. (1987), Mohd-Saman and Evans (1993), Banerjee (1988), Wolfe (1989b)]. There is a great need to ensure that the loop iterations that can be executed in parallel will give the correct results. In this section, a verification technique for fixed parallel loops is presented. Mohd-Saman and Evans (1993) have proposed the Bernstein Loop Tests (BLTs) to test for parallelism in loops. These tests have been fully described in Chapter 5 of this thesis. Similar to the BTA described in Section 7.4 above, the BLTs can be used as an assertion in verifying the correctness of fixed parallel loops.

A loop with n iterations can be treated as having n stanzas, one stanza for each iteration. Then, the BTA, described in Section 7.4, can be used to test the correctness of any two or more iterations and thus the correctness of the loop. However, the presence of array references in the loop body creates difficulty in determining the dependence between iterations. The BLTs which use the data reference directions of $>$, $<$ and $=$, for array variables, have been shown to be an efficient way to test for loop parallelism.

Another assertion called the **BLT Assertion (BLTA)** will be formed to be part of the method to verify the correctness of the parallel loops. In order to do this, input and output assertions together with the BLTA will be placed in the loops. Hence, it must be shown that, given an input assertion P , for the loop to be true,

- (a) the output assertion Q must be true in a sequential manner
- (b) the BLTA must be non-empty (of any scalar variables) and the results for directed variables (i.e., array) must be of Equal direction.

In order to check for (b), two sets R_1 and R_2 are defined as follows.

$$\begin{aligned} \text{Let } R_1 &= WYZ_i \cap XYZ_j \text{ for all } i \text{ and } j \\ \text{Let } R_2 &= XYZ_i \cap XYZ_j \text{ for all } i \text{ and } j, \quad i \leq j \end{aligned}$$

For all $1 \leq i, j \leq n$, n = number of stanzas in the loop body

Let $\text{ELEMENT}(D,R)$ be the direction:

$D(\text{EQUAL}, \text{FORWARD}, \text{BACKWARD})$

in set R . Assume that any FORWARD/BACKWARD direction with zero distance is similar to EQUAL. The BLTA will be of the following form.

PROVE(BLTA)

where BLTA consists of the following assertion:

$$(R1 = \emptyset \wedge \text{ELEMENT}(\text{EQUAL}, R1)) \wedge (R2 = \emptyset \wedge \text{ELEMENT}(\text{EQUAL}, R2))$$

This assertion reads as follows:

The parallel property of the loop is preserved if (R1 is \emptyset AND EQUAL is the only direction of all array variables in R1) AND (R2 must be empty AND EQUAL is the direction of all array variables in R2).

Consider the fixed parallel loop in figure 7.7 which has n iterations with three statements in the loop body. Assume that, when the loop is executed, n processes are created and scheduled to run on n processors. The execution tree is shown in figure 7.8. In verifying the correctness of such a loop, the individual BSs for the statements in the body of the loop are first determined. This can be done as the SE is performed on the body of the loop. At the end of the SE, if the loop stanza is verified to be correct sequentially, then the BLTA will be checked and verified. Therefore, for a loop to be verified as correct, the BLTA must be true and the correctness of its body in a sequential manner must also be determined.

To handle mutual exclusions in fixed parallel loops, similar to the method described in Section 7.5.2, SVSs for the loop body have to be formed first. Then $R1$ and $R2$ are modified as follows.

$$\begin{aligned} R1 &= (WYZ_i - SVS_i) \cap (XYZ_j - SVS_j) \\ R2 &= (XYZ_i - SVS_i) \cap (XYZ_j - SVS_j) \quad i \leq j \\ &\text{for all } 1 \leq i, j \leq n, \text{ where } n = \text{number of stanzas in a loop} \end{aligned}$$

For a conditional loop, the SE technique as described by Hantler and King (1976) with the loop assertions inserted in the program, will suffice in verifying its correctness.

```
S0;  
LOOP i := 1 TO n DO  
  BEGIN  
    ASSUME(Pi)  
    S1;  
    S2;  
    S3;  
    PROVE(Qi)  
  END  
  PROVE(BLT Assertion)  
S4
```

Figure 7.7: Verification of a fixed parallel loop

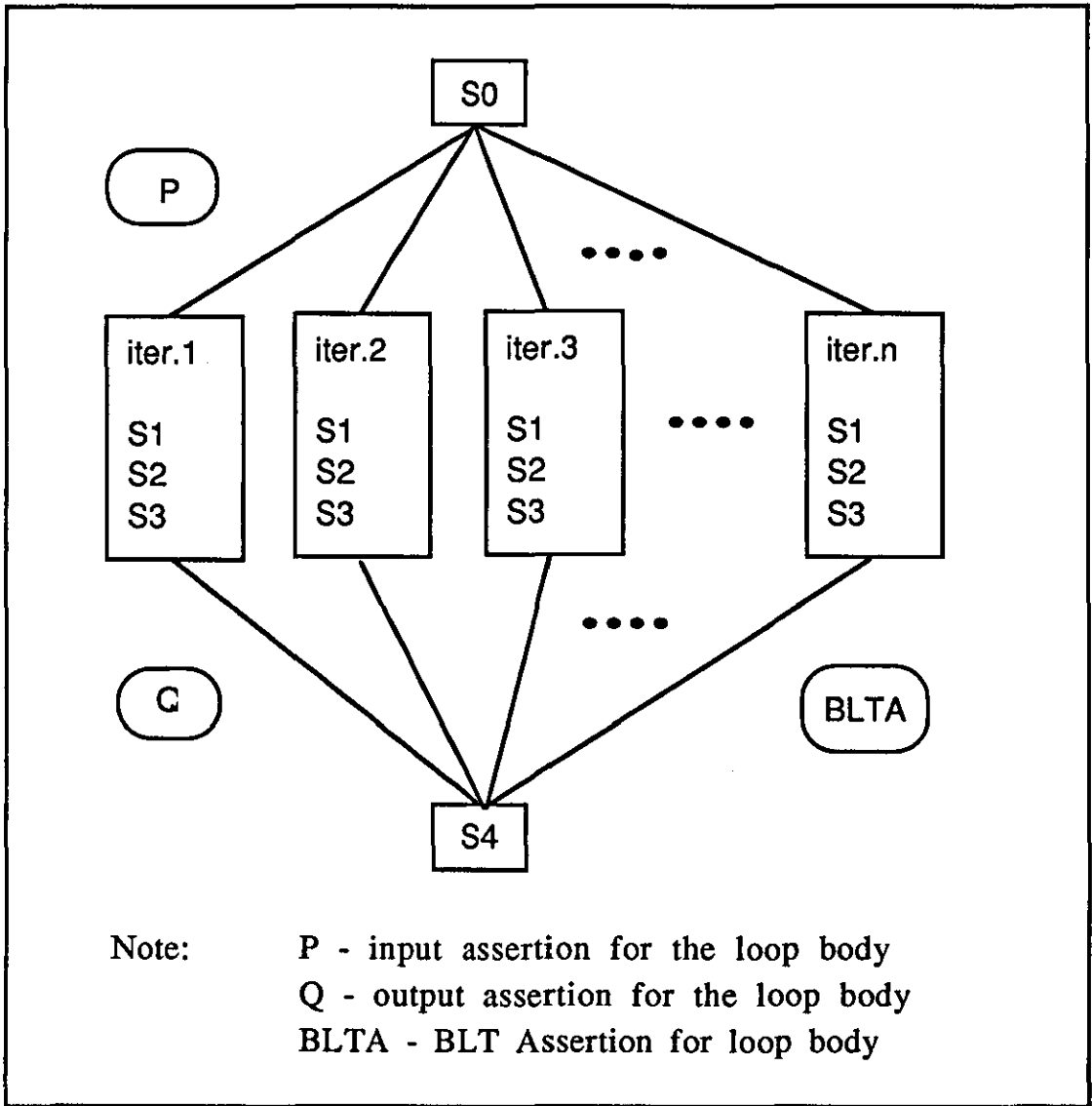


Figure 7.8: N processes for the loop in figure 7.7

7.7 SUMMARY

The research on proving the correctness of programs has received a lot of attention. When reliability and safety are critical then software development needs special tools to help the programmers develop correct and reliable software. This chapter has presented simple methods in verifying the correctness of parallel programs. It also has given an overview of the research work that have been carried out in this particular field.

For parallel programs, especially the fixed parallel loops, great care has to be given in proving them correct. This is because many processes are running at the same time and are able to modify variables shared among them. It has been demonstrated that the Bernstein Test developed by Williams (1978) and the Bernstein Loop Test developed by Mohd-Saman and Evans (1993) can be conveniently used to verify the properties of parallel stanzas and loops.

One of the methods for verifying the correctness of sequential programs is the Symbolic Execution (SE) [Hantler and King (1976), King (1976)]. In this method, correctness assertions are placed as part of the program. The program is then executed symbolically to determine if the assertions are correct or not. This method ensures a large class of inputs can be carried out for execution. In this chapter, it has ^{been} shown how the SE can be integrated with Bernstein Method assertions to verify the properties of parallel programs and loops. These assertions are called the BT Assertion and the BLT Assertion.

The use of SE in verifying the correctness of programs still needs a lot of attention and improvement. In the SE, Verification Conditions (VCs) are generated. VCs tend to be large formulae and this is one of the shortcomings of using the SE. If VCs cannot be proved, it could mean three possible explanations. First, the method of proving the VCs is inadequate. Second, there are actually errors in the program and third, the assertions supplied by the programmers are insufficient.

Interactive incremental development and verification of programs offers an applicable way to develop error free programs such as in the Penelope system [Guaspari et al. (1990)]. The role of the programmer with the help of a tool to verify a program interactively, may present a better development strategy rather than having a batch type of program verification. Assertions can be supplied one by one as the development of the program gets bigger and bigger. These assertions can be modified during development and verification may be carried^{out} on those parts which have only been modified. However, De Milo et al. (1979) have argued that program proving using the mathematical approach is still a very difficult task.

CHAPTER 8

SUMMARY AND CONCLUSIONS

8.1 INTRODUCTION

Most of the computer programs currently available are written in a sequential manner which suits the architecture of a single-processor computer. With the availability of the multi-processor computer, in which each processor can execute different parts of a program in parallel, the task of programming in parallel has increased. Sequential programs have to be transformed into their parallel version to take advantage of the fast and concurrent processing. Old programs (or dusty decks) have to be rewritten and this will take a lot of time and effort. Hence, a software tool that can automatically parallelize a sequential program is greatly needed to transform existing programs as well as to ease the programming tasks of programmers who are already familiar with sequential programming. A Parallelizing Compiler is one such tool that is able to perform the whole process of compiling and restructuring [Appelbe and Smith (1989), Guarna et al. (1989), Polychronopoulos et al. (1990)].

Numerous research work has been conducted since the sixties when the interest in parallel computing began to emerge [Allen and Cocke (1976), Baer (1973), Bernstein (1966), Burke et al. (1988), Callahan et al. (1987), Gonzalez (1972), Kuck et al. (1972), Padua and Wolfe (1986), Polychronopoulos (1988), Tjaden (1970), Williams (1978)]. The main objective of the various researches is to develop techniques to extract parallelism and to perform program transformation. Most of the work so far has concentrated on the Fortran language because it has been the most popular and widely used language for numerical computation. However, some work has been done on other languages such as C and Pascal [Gabber et al. (1993), Huson et al. (1986), Polychronopoulos et al. (1990), Tsuda and Kunieda (1992)].

This thesis has focused its study on the determination of implicit parallelism in sequential programs. It is based on the Bernstein Sets [Bernstein (1966)] and the sets of tests developed by Williams (1978). The topics that have been studied include:

- a. the design of a software tool that can determine parallel parts, called stanzas, of a sequential program and their scheduling on a shared-memory multi-processor system. The tool is also capable of solving the problem of determining optimal stanza granularity.
- b. the detection of parallelism in sequential loops and their transformation into parallel forms.
- c. the Inter-procedural Analysis which involves gathering information when a procedure call is made.
- d. verification of the correctness of parallel stanzas.

This chapter will summarise and then conclude the topics discussed throughout the thesis. Section 8.2 gives a summary and conclusion for Chapter 4 which has detailed discussion on the implementation of TAG, a software tool to detect parallelism in sequential programs and to determine stanza granularity. In Section 8.3, a summary and conclusion on the Bernstein Loop Tests and loop transformations, discussed in Chapter 5, will be presented. Inter-procedural analysis discussed in Chapter 6 is summarised in Section 8.4. The applicability of the Bernstein Method to form as assertions to be used for program verification will be concluded in Section 8.5. Finally, Section 8.6 will suggest some further research that can be carried out in this field.

8.2 DEPENDENCE ANALYSIS AND SCHEDULING OF STANZAS

Apart from the problem of the "dusty decks," programming in parallel is a very difficult and tedious task since most programmers are familiar with writing sequential programs. It involves identifying those parallel parts in the programs and coding them using parallel constructs. Hence, programmers need software tools such as the Parallelizing Compiler, to help them carry^{out} this type of programming as easily as possible without many difficulties.

Data dependence analysis (DDA) is a major operation that has to be carried out by a parallelizing compiler. It performs the detection of independent operations in user programs. Detailed examination of the program is carried out on how the data is referenced, especially in those involving array elements. It will determine whether the different references can take place simultaneously or not.

Much effort has been made on designing good algorithms for the DDA. Chapter 3 has surveyed the various techniques used. One of them is the Bernstein Method which forms the basis of the study discussed in this thesis. In Chapter 4, an implementation of the Bernstein Tests (BTs) in a software tool called TAG, is presented. It also includes discussions on the problem of scheduling and granularity of stanzas. In this thesis, the Bernstein Method has demonstrated to be a viable and useful way to perform the DDA. The BSs contain information that can be used for testing parallelism between stanzas of a program. The individual set in the BSs forms a natural way to indicate the flow of data in a stanza.

Scheduling is a scheme that allocates stanzas or tasks to processors in a parallel computer. One of the goals of scheduling parallel or concurrent stanzas is to achieve an optimal overall execution time of the program. This is not a simple task to do since utilizing more than one processor usually incurs some overhead caused by inter-task communication. This communication overhead is an extra time needed for the data transfer between processors and it is mainly caused by the data dependences that exists between the concurrent stanzas. This will make some of the processors idle waiting for some stanzas to finish their executions. Trying to achieve an optimal execution time for concurrent programs sometimes leads to unbalanced use of the processors. Therefore, the scheduling technique should have a capability to balance between maximizing the parallelism and minimizing the overhead of communication. This problem is also related to the determination of an optimal stanza grain size which has been acknowledged to be very difficult to solve.

However, the study conducted in this research has showed that a heuristic with repeated scheduling and merging can produce near-optimal stanza size (see Chapter 4).

8.3 LOOP DEPENDENCES AND TRANSFORMATIONS

Loops have been the main focus that are analysed because they provide the best opportunities for parallelism. Most of the research work has concentrated on designing accurate algorithms to deal with complicated array subscripts in the DDA. Once the dependences have been identified, the loops can then be transformed into parallel forms so that they can be executed concurrently.

Chapter 5 has described a set of tests, called the Bernstein Loop Tests (BLTs), that can be applied to loops to detect whether their iterations can be run concurrently or not. The BLTs use the Data Reference Directions (DRDs) for array references. This enables the tests to determine if multiple accesses to array elements occur or not between iterations.

Based on the results of the BLTs and the types of the BSs involved, loops can be transformed into parallel forms, as discussed in Chapter 5. There are several schemes that can be used to do the modifications and most of them are readily found in optimizing compilers. Examples of such techniques are forward substitution, scalar renaming and loop distribution. Those parts of the program whose data dependences cannot be eliminated can still be parallelized but only after synchronization statements are introduced. This is one of the main tasks of a Parallelizing Compiler, i.e, transforming serial programs into parallel versions.

8.4 INTER-PROCEDURAL ANALYSIS

Procedure calls in programs is another important factor that may hinder detection of parallelism. The nature of procedures is that they hide certain information such as the detailed array references from being analysed directly by a simple DDA. This

information is usually passed through parameters. Hence, the DDA should be extended to perform the Inter-procedural Analysis (IPA) to uncover more parallelism that may exist when procedure calls are made.

In Chapter 6, the collection of information in IPA, based on the Bernstein Method, has proved to be an easy and feasible method. This information can then be readily used in the BTs and the BLTs without any modifications to the tests or the BSs. The IPA can handle any call to procedure with call-by-value and call-by-reference types of parameter passing. It also handles recursive procedures efficiently.

8.5 VERIFICATION OF PARALLEL PROGRAMS

As mentioned earlier, writing a parallel program is a very difficult, time-consuming and tedious task. Programmers tend to make mistakes unknowingly. Therefore, there is a great need to ensure that the program written is error-free and correct. This problem of correctness relies heavily on the capabilities of the programmers. One simple way is by testing the finished programs with sets of data but this may not give total correctness. Proving correctness has also been done theoretically by using formal methods. Programs are modelled mathematically and then proved. This method has been argued to be a tedious and time-consuming process because the arguments involved can be large. However, the need for verifying the correctness will remain a very interesting area of research in the future.

One technique that has been widely used is the Symbolic Execution (SE). Chapter 7 has showed that this technique can be easily applied to verify parallel stanzas. This is done by performing the SE extended with a new assertion (apart from the input and output assertions) called the BT Assertion (BTA) to test for the parallel properties of any two stanzas. Another assertion, the BLT Assertion (BLTA) allows loops targeted for parallel execution to be verified. Information for the BTA and BLTA can be readily collected by the SE during its verification process.

Chapter 7 has also described how stanzas with the presence of critical sections can be verified using this technique.

8.6 FUTURE RESEARCH

As stated earlier, the goal of the research presented in this thesis is to study methods for the determination of parallelism in programs. Related topics include stanza scheduling, program transformations and verification of program correctness. This thesis has discussed a number of topics that certainly need further investigation.

- a. The Bernstein Method has not been pursued by many researchers. Hence its applicability and suitability has not been fully analysed and tested. This thesis has showed its power and usefulness. Now a full implementation in its complete form should present a radical change from the usual method conducted by other researchers.
- b. This thesis has not performed any comparative study between the Bernstein Method and other method for the DDA. This performance study should give the real indication of its usefulness.
- c. The BTs and the BLTs are tests applied on sets. The Diophantine Analysis (i.e., numerical method) described in Chapter 3 does not come into this category. It would be interesting to investigate the possibility of integrating the Bernstein Method with this numerical method.
- d. In this thesis, the BTs and the BLTs are used to detect parallelism involving scalar variables and array references. However, programming languages contain other forms of memory accesses. Notably is the use of pointers such as in the C and Pascal languages. Several techniques have been suggested [Hendren and Nicolau (1989)] but the Bernstein Method should be able to handle this kind of programming construct and this needs further investigation.

- e. Currently, object-oriented programming languages are becoming widely in use such as the C++ Language [Stroustrup (1986)]. These languages provide object encapsulation and thus can be represented as stanzas. This needs a further study since handling of objects throughout an object-oriented program needs an extensive use of the IPA. The methods described in this thesis could be useful in performing such ^aprocess.
- f. In verification of programs, the BTs and the BLTs have been used as assertions together with the Symbolic Execution (SE). Apart from the SE technique, the use of formal methods to verify programs is gaining acceptance. One way is to use a specification language such as the VDM or Z to develop a correct program [Bowen (1988), McParland and Kilpatrick (1988)]. Correctness is verified at the specification level before the development of the actual program. The BTs and the BLTs should offer an interesting way if they could be modelled in the specification program in the development of parallel software.

REFERENCES

- Adam T.L., Chandy, K.M. and Dickson, J.R. (1974) "A Comparison of List Schedules for Parallel Processing Systems", Communications of The ACM 17, no. 12, pp: 685-690
- Aho, A.V., Sethi, R and Ullman, J.D. (1986) "Compilers Principles, Techniques and Tools", Addison-Wesley
- Aiken, A. and Nicolau, A. (1990) "Fine-grain Parallelization and the Wavefront Method", in Languages And Compilers for Parallel Computing (D. Gelernter, A. Nicolau and D Padua: Editors), Pitman, pp: 1-16
- Albert, E., Knube, K., Lukas, J.D., Steele Jr, G.L. (1988) "Compiling Fortran 8x Array Features for the Connection Machine Computer System", Proc. ACM/Sigplan, pp: 42-56
- Allen, F.E. (1988) "Compiling for Parallelism: An Overview", Parallel Systems and Computations, Ed: G Paul and GS Almasi, North-Holland, pp: 3-13.
- Allen, F.E. and Cocke, J. (1976) "A Program Data Flow Analysis Procedure", Comm. of ACM Vol 19-3, pp: 137-147
- Allen, J.R., Kennedy, K., Porterfield, C. and Warren, J. (1983) "Conversion of Control Dependence to Data Dependence", 10th ACM Symp. on Principles of Prog. Lang, pp: 177-189
- Allen, J.R. and Kennedy, K. (1984a) "Automatic Loop Interchange", Proc. of SIGPLAN Symp. on Compiler Constructions, pp: 233-246..
- Allen, R.A. and Kennedy, K. (1984b) "PFC: A Program to Convert Fortran to Parallel Form", In Tutorial Supercomputers: Design and Applications, Ed: Hwang, K., IEEE Computer Soc., pp: 186-203
- Allen, R., Baumgartner, D., Kennedy, K. and Porterfield, A. (1986) "PTOOL: A Semi-automatic Parallel Programming Assistant",

Computer Science Technical Rep. COMP TR86-31, Rice University, Houston, TX.

Allen, R. and Kennedy, K. (1987) "Automatic Translation of FORTRAN programs to Vector form", ACM Trans. on Prog. Lang. and Syst., Vol. 9, No. 4, pp: 491-542.

Allen, R., Callahan, D. and Kennedy, K. (1987) "Automatic Decomposition of Scientific Programs for Parallel Execution", Proc. of 14th ACM Symp. on Principle of Prog. Lang, pp: 63-76.

Almasi, S.A and Gottlieb, A., (1989) "Highly Parallel Computing", The Benjamin/Cummings Publishing Comp. Inc.

Andrew, G.R. and Schneider, F.B., (1983) "Concepts and Notations for Concurrent Processing", Computing Surveys 15, pp: 3-43

Appelbe, B. and Smith, K., (1989) "Start/Pat: A Parallel-programming Toolkit", IEEE Software, pp: 29-38.

Apt, K.R. (1986) "Correctness Proofs of Distributed Termination Algorithms", ACM Transaction on Programming Languages and Systems, Vol. 8, No. 3, pp: 388-405

Axelrod, T.S. (1986) "Effects of Synchronization Barriers on Multiprocessor performance", Parallel Computing 3, pp: 129-140. North-Holland.

Bach, M.J. (1986) "The Design of the UNIX Operating System", Prentice-Hall

Baer, J.L. (1973) "A Survey of Some Theoretical Aspects of Multiprocessing", Computing Surveys, Vol. 5-1, pp: 31-80.

Balasundram, V. and Kennedy, K., (1989) "A Technique for Summarizing Data Access and its Use in Parallelism

Enhancing Transformation", Proc. SIGPLAN Conf. on Prog. Lang., pp:41-53.

Banerjee, U. (1988) "Dependence Analysis for Supercomputing", Kluwer Academics Publishers

Banerjee, U. (1990) "A Theory of Loop Permutations", in Languages And Compilers for Parallel Computing (D. Gelernter, A. Nicolau and D Padua: Editors), Pitman, pp: 54-74

Banning, J.P. (1979) "An Efficient Way to Find the Side Effects of Procedure Calls and the Aliases of Variables", 6th ACM Symp. on Prin. of Prog. Lang., pp: 29-41.

Barth, J.M. (1978) "A Practical Inter-procedural Data Flow Analysis Algorithm", Communication of the ACM 21-9, pp: 724-736

Beckmann, C.J. and Polychronopoulos, C.D. (1991) "The Effect of Scheduling and Synchronization Overhead on Parallel Loop Performance", CSRD Report No. 1111, Center For Supercomputing Research and Development, Univ. of Illinois at Urbana-Champaign

Bernstein, A.J., (1966) "Analysis of programs for Parallel Processing", IEE Trans. on Elec. Comp. Vol 1 EC-15, pp: 757-763.

Bershad, B.N., Lazowska, E.D. and Levy, H.M. (1988) "PRESTO: A System for Object-oriented Parallel Programming", Software-Practice and Experience, Vol 18(8), pp: 713-732

Bieler, F. (1990) "Partitioning Programs into Processes", Proc. of Joint Int. Conf. on Vector and Parallel Processing, CONPAR 90-VAPP IV, Springer-Verlag, pp: 513-524.

- Blume, W. and Eigenmann, R. (1992) "Performance Analysis of Parallelizing Compilers on the Perfect Benchmarks Programs", CSRD Report No. 1218, Center For Supercomputing Research and Development, Univ. of Illinois at Urbana-Champaign
- Boiten, E.A., Partsch, H.A., Tuijnman, D. and Volker, N. (1992) "How to produce correct Software - An Introduction to Formal Specification and Program Development by Transformations", The Computer Journal, Vol. 35, No. 6, pp: 547-554
- Bokhari, S.H. (1988) "Partitioning Problems in Parallel, Pipelined, and Distributed Computing", IEEE Trans. on Computers, Vol. 37, No. 1, pp: 48-57
- Bowen, J.P. (1988) "Formal Specification in Z as a Design and Documentation Tool", 2nd IEE/BCS Conf. Software Engin., pp: 164-168
- Boyer, R.S., Elspas, B. and Levitt, K.N. (1975) "SELECT - A Formal System for Testing and Debugging Programs by Symbolic Execution", Reliable Software Intl. Conf., pp: 234-245
- Burke, M and Cytron, R. (1986) "Inter-procedural Dependence Analysis and Parallelization", ACM SIGPLAN Symposium on Compiler Construction pp: 162-175
- Burke, M., Cytron, R., Ferrante, J., Hsieh, W., Sarkar, V., Shield, D., (1988) "Automatic Discovery of Parallelism: A Tool and an Experiment (Extended Abstract)", Proc. of the ACM/SIGPLAN PGEALS, pp: 77-84.
- Butt, W. (1993) "Load Balancing Strategies for Distributed Computer Systems", PhD Thesis, Loughborough Univ. of Tech

- Callahan, D., Cooper, K.D., Kennedy, K. and Torczon, L. (1986) "Inter-procedural Constant Propagation", SIGPLAN Notices Vol. 21, Part 7, pp: 152-161.
- Callahan, D. and Kennedy, K. (1987) "Analysis of Inter-procedural Side Effects in a Parallel Programming Environment", Supercomputing 1st Conf, Lecture Notes in Computer Science, No. 297, pp: 138-171.
- Callahan, D., Cooper, K.D., Hood, R.T., Kennedy, K., Torczon, L., Warren, S.K., (1987) "Parallel Programming Support in Parascope", Parallel Computing in Science and Engineering, Lecture Notes in Computer Science 295.
- Chandra, R., Gupta, A. and Hennesy, J.L. (1990) "COOL: a Language for Parallel Programming", in Languages And Compilers for Parallel Computing (D. Gelernter, A. Nicolau and D Padua: Editors), Pitman, pp: 126-148
- Cherniavsky, J.C. and Smith, C.H. (1986) "A Theory of Program Testing with Applications" Proc. of Workshop on Software Testing, Canada, Computer Society Press, pp: 110-121
- Clarke, L.A. and Richardson, D.J. (1984) "Symbolic Evaluation - an Aid to Testing and Verification", Software Validation, H. L. Hausen (editor), North-Holland, pp: 141-167
- Coffman Jr, E.G. (1976) "Computer and Job-shop Scheduling Theory", John-Wiley.
- Cohen, D., Swartout, W. and Balzer, R. (1982) "Using Symbolic Execution to Characterize Behaviour", ACM SIGSOFT Software Eng. Notes, Vol. 7 No. 5, pp: 25-32
- Cooper, K.D., Kennedy, K. and Torczon, L. (1986) "The Impact of Inter-procedural Analysis and Optimization in the R^n Programming Environment", ACM Trans. on Prog. Lang. and Syst., Vol. 8, No. 4, pp: 491-523.

- Cooper, K.D. and Kennedy, K. (1988) "Inter-procedural Side-effect Analysis in Linear Time", Proc. of SIGPLAN, Conf. on Prog. Lang.: Design and Implementation, 1988, pp: 57-66.
- Cooper, K.D. and Kennedy, K. (1989) "Fast Inter-procedural Alias Analysis", 16th ACM Symp. on Principles of Prog. Lang., pp: 49-59
- Cooper, K.D., Hall, M.W. and Torczon, L., (1991) "An Experiment with In-line Substitution", Software-Practice and Experience, Vol 21(6), pp: 581-601.
- Cowell, W.R. (1988) "Users' Guide to Toolpack/1 Tools for Data Dependency Analysis and Program Transformation", Argonne National Lab.
- Cowell, W.R. and Thompson, C.P., (1990) "Tools to aid in Discovering Parallelism and Localizing Arithmetic in Fortran Programs", Software-Practice and Experience, Vol 20(1), pp: 25-47.
- Cytron, R., (1986) "Doacross: Beyond Vectorization for Multiprocessors (extended abstract)", Proc. of Int. Conf. on Parallel Proc., pp:836-844.
- Cytron, R., Ferrante, J. and Sarkar, V. (1990) "Experiences Using Control Dependences in PTRAN", in Languages And Compilers for Parallel Computing (D. Gelernter, A. Nicolau and D Padua: Editors), Pitman, pp: 186-212
- Cytron, R., Ferrante, J, Rosen, B.K., Wegman, M.N. and Zadeck, F.K. (1991) "Efficiently Computing Static Single Assignment Form and the Control Dependence Graph", ACM Transaction on Programming Languages and Systems, Vol. 13, No. 4, pp: 451-490

- D'Hollander, E.H. (1989) "Partitioning and Labeling of Index Sets in Do Loops with Constant Dependence Vectors", Proc. Int. Conf. on Parallel Proc, pp: 113-144.
- Davidson, J.W. and Holler, A.M., (1988) "A Study of a C Function In-liner", Software-Practice and Experience, Vol 18(8), pp: 775-790.
- Davies, J., Huson, C., Macke, T., Leasure, B. and Wolfe, M. (1986) "The KAP/S-1: An Advanced Source-to-Source Vectorizer for the S-1 Mark IIa Supercomputer", Proc. of Intl. Conf on Parallel Processing, pp: 833-835.
- De Millo, R.A., Lipton, R.J. and Perlis, A.J., (1979) "Social Processes and Proofs of Theorems and Programs", Communication of the ACM 22-5, p:271-280
- Dennis, J.B. (1980) "Data Flow Supercomputers", IEEE Computer, pp: 48-56
- Dijkstra, E.W. (1976) "A Discipline of Programming", Prentice-Hall.
- Dillon, L.K. (1988) "Symbolic Execution-based Verification of ADA Tasking Programs", 3rd IEEE Int. Conf. on ADA Applic. and Env., pp: 3-13
- Dillon, L.K., (1990) "Verifying General Safety Properties of ADA Tasking Programs", IEEE Trans. on Software Eng., Vol. 16-1
- Dillon, L.K., Kemmerer, R.A. and Harrison, L.J. (1988) "An Experience with Two Symbolic Execution-Based Approaches to Formal Verification of ADA Tasking Programs", 2nd Workshop on Software Testing, Verification and Analysis, pp: 114-122.
- Duda, A. (1988) "On the Tradeoff Between Parallelism and Communication", Proc. of the 4th Int. Conf. on Modeling

Techniques and Tools for Computer Performance Evaluation, pp: 323-334.

Ebenstein, S.E. and Mcdermott, T.L. (1990) "Optimizing Techniques for Parallel Processing", *Software-Practice and Experience*, Vol 20(8), pp: 833-849.

Ehrig, H., Mahr, B., Classen, I. and Orejas, F. (1992a) "Introduction to Algebraic Specification. Part 1: Formal Methods for Software Development", *The Computer Journal*, Vol. 35, No. 5, pp: 460-467

Ehrig, H., Mahr, and Orejas, F. (1992b) "Introduction to Algebraic Specification. Part 2: From Classical View to Foundations of System Specification", *The Computer Journal*, Vol. 35, No. 5, pp: 468-477

Eigenmann, R. and Blume, W. (1991) "An Effectiveness Study of Parallelizing Compiler Techniques", CSRD Rpt. No. 1090 University of Illinois USA.

Evans, D.J. (1990) "Multitasking Strategies in Parallel Computing", *Computer Studies* 557, Loughborough University of Tech.

Evans, D.J and Williams S.A., (1978) "Analysis and Detection of Parallel Processable Code", *The Computer Journal*, Vol. 23, no. 1, pp:66 - 72.

Evans, D.J. and Mohd-Saman, M.Y. (1993) "Determination of Parallelism in Programs", In *Software for Parallel Computation* (Ed: Kowalik, J.S. and Grandinelti, L.) Springer-Verlag.

Ferrante, J., Ottenstein, K.J. and Warren, J.D. (1987) "The Program Dependence Graph and Its Use in Optimization", *ACM Trans. on Prog. Lang. and Syst.*, Vol. 9, No. 3, pp: 319-349.

- Flynn, M.J. (1972) "Some Computer Organizations and Their Effectiveness", IEEE Trans. on Computers, Vol. C-21, No. 9, pp: 948-960
- Foster, I. (1991) "Automatic Generation of Self-Scheduling Programs", IEEE Trans. on Parallel and Distributed Systems, Vol. 2-1, pp: 68-78
- Frankl, P.G. and Weyuker, E.J. (1986) "Data Flow Testing in the Presence of Unexecutable Paths" Proc. of Workshop on Software Testing, Canada, Computer Society Press, pp: 4-13
- Freeman, T.L. and Phillips, C. (1992) "Parallel Numerical Algorithms", Prentice Hall.
- Gabber, E., Averbach, A. and Amiram, Y. (1993) "Portable, Parallelizing Pascal Compiler", IEEE Software, pp: 71-81.
- Garey, M.R. and Johnson, D.S., (1979) "Computers and Interactability: A guide to the Theory of NP-Completeness", WH Freeman and Company.
- Garsden, H. and Wendelborn, A. L. (1990) "A Comparison of Microtasking Implementations of the Applicative Language SISAL", Proc. of Joint Int. Conf. on Vector and Parallel Processing, CONPAR 90-VAPP IV, Springer-Verlag, pp: 697-708.
- Gehani, N. (1984) "ADA Concurrent Programming", Prentice-Hall.
- Gehani, N.H. and Roome, W.D. (1988) "Concurrent C++: Concurrent Programming with Class(es)", Software-Practice and Experience, Vol 18(12), pp: 1157-1177.
- Girkar, M.B. and Polychronopoulos, C.D. (1988) "Partitioning Programs for Parallel Execution", Supercomputing ACM Proc. of Intl. Conf, St. Malo, France, pp: 216-229.

- Goff, G., Kennedy, K. and Tseng, C. (1991) "Practical Dependence Testing", Proc. of ACM SIGPLAN 1991 Conf. on Prog. Lang. Design and Impl. Toronto, Canada, pp: 15-29..
- Gonzalez, M.J. (1972) "Parallel Task Execution in Decentralized System", IEEE Transactions on Computer 21, no. 12, pp: 1310-1322.
- Good, D.I., Cohen, R.M. and Keeton-Williams, J. (1979) "Principles of Proving Concurrent Programs in Gypsy", 6th ACM Symp. on Principles of Programming Language, pp: 42-52.
- Gries, D. (1977) "An Exercise in Proving Parallel Programs Correct", Communication of the ACM 20-12, p:921-930
- Gries, D. (1981) "The Science of Programming", Springer-Verlag.
- Guarna, V.A., Gannon, D., Jablonowski, D., Malony, A.D. and Gaur, Y. (1989) "Faust: An Integrated Environment for parallel programming", IEEE Software, pp: 20-26.
- Guaspari, D., Marceau, C. and Polak, W. (1990) "Formal Verification of ADA Programs", IEEE Trans. on Software Eng., Vol. 16-9.
- Gurd, J.R., Kirkham, C.C. and Watson, I. (1985) "The Manchester Prototype Dataflow Computer", Communications of The ACM 28, no. 1, pp: 34-55.
- Haghighat, M.R. (1990) "Symbolic Dependence Analysis for High Performance Parallelizing Compilers", CSRD Rpt. No. 995 MSc Thesis, University of Illinois USA.
- Hall, P.A.V. (1988) "Towards Testing with Respect to Formal Specification", 2nd IEE/BCS Conf. Software Eng., pp: 159-163

- Halstead, R.H. Jr. (1985) "Multilisp : A Language for Concurrent Symbolic Computation", ACM Transactions on Programming Languages and Systems 7, no 4, pp: 501-538.
- Hansen, P.B. (1973) "Operating System Principles", Prentice Hall.
- Hansen, P.B. (1975) "The Programming Language Concurrent Pascal", IEEE Transactions on Software Engineering SE-1, no 2, pp: 199-207.
- Hantler, S.L. and King, J.C. (1976) "An Introduction to proving Correctness of programs", ACM Computing Surveys Vol 8 No. 3, pp: 331-353
- Harrison, L.J. and Kemmerer, R.A. (1988) "An Interleaving Symbolic Execution Approach for the Formal Verification of Ada Programs with Tasking", Proc. of Third IEEE Conf. on Ada: Application and Environment, pp: 15-26.
- Harrison III, W.L. and Chow, J.H. (1991) "Dynamic Control of Parallelism and Granularity in Executing Nested Parallel Loops", CSRD Report No. 1167, Center For Supercomputing Research and Development, Univ. of Illinois at Urbana-Champaign.
- Havlak, P. and Kennedy, K. (1991) "An Implementation of Interprocedural Bounded Regular Section Analysis", IEEE Transactions on Parallel and Dist. Syst., Vol. 2-3, pp: 350-360.
- Hendren, L.J. and Nicolau, A. (1989) "Interference Analysis Tools for Parallelizing Programs with Recursive Data Structures", Proc. Int. Conf. on Supercomputing, Greece, pp: 205-214.
- Hiranandani, S., Kennedy, K. and Tseng, C.W. (1992) "Compiling Fortran D for MIMD Distributed-Memory Machines", Communications of the ACM 35, no. 8, pp: 66-80.

- Hoare, C.A.R. (1969) "An axiomatic basis for computer programming", Comm. of the ACM 12-10, pp: 576-583
- Hoare, C.A.R. (1978) "Communicating Sequential Processes", Comm. of The ACM 21, no. pp: 666-677.
- Howden W.E. (1977) "Symbolic Testing and the DISSECT Symbolic Evaluation System", IEEE Trans. on Software Eng., Vol. SE-3, No. 4, pp: 266-278
- Huson, C., Macke, T., Davies, J., Wolfe, M. and Leasure, B. (1986) "The KAP/205: An Advanced Source-to-Source Vectorizer for the Cyber 205 Supersomputer", Proc. of 1986 Intl. Conf on Parallel Processing, pp: 827-832.
- Hwang, K. and Briggs, F.A. (1984) "Computer Architecture and Parallel Processing", McGraw-Hill.
- Jackson, D.T. (1985) "Data Movement on DOALL Loops", CSRD Report 524, University of Illinois.
- Kemmerer, R.A. and Eckman, S.T. (1985) "UNISEX: a UNIX-based Symbolic EXecutor for Pascal", Software-Practice and Experience, Vol. 15(5), pp: 439-458.
- Kernighan B.W. and Ritchie, D.M. (1988) "The C programming Language", Prentice-Hall.
- King, J.C., (1976) "Symbolic Execution and Program Testing", Communication of the ACM 19, 7, pp:385-394
- Kong, X., Klappholz, D. and Psarris, K. (1991) "The I Test: an Improved Dependence Test for Automatic Parallelization and Vectorization", IEEE Transactions on Parallel and Dist. Syst., Vol. 2-3, pp: 342-348.
- Krothapalli, V.P. and Sadayappan, P. (1991) "Removal of Redundant Dependences in DOACROSS Loops with Constant

Dependences", IEEE Trans. on Parallel and Distributed Systems, Vol. 2-3, pp: 281-289.

Kruatrachue, B. and Lewis, T. (1988) "Grain Size Determination for Parallel Processing", IEEE Software, pp: 23-32

Kruse, R.L. (1984) "Data Structures and Program Design", Prentice-Hall

Kuck, D.J. (1968) "ILLIAC IV: Software and Application Programming", IEEE Transactions on Computer 17, no. 8, pp: 758-770.

Kuck, D.J. (1978) "The Structure Of Computers and Computations", Volume 1, John Wiley and Sons, New York.

Kuck, D.J., Muraoka, Y., Chen, S.C. (1972) "On The Number Of Operations Simultaneously Executable in Fortran-Like Programs and their resulting Speedup", IEEE Transactions on Computer 21, no. 12, pp: 1293-1310.

Kuck, D.J., Kuhn, R.H., Padua, D.A., Leasure, B. and Wolfe, M. (1981) "Dependence Graphs and Compiler Optimizations", Proc. Conf. 8th. ACM Symp. on Principles of Prog. Lang., pp: 207-218.

Kuck, D.J., Kuhn, R.H., Leasure, B. and Wolfe, M. (1984) "The Structure of an Advanced Retargetable Vectorizer", In Tutorial Supercomputers: Design and Applications, Ed: Hwang, K., IEEE Computer Soc., pp: 163-178

Kwan, A.W., Bic, L. and Gajski, D.D (1990) "Improving Parallel Program Performance Using Critical Path Analysis", in Languages And Compilers for Parallel Computing (D. Gelernter, A. Nicolau and D Padua: Editors), Pitman, pp: 358-373

- Leasure, B. (1985) "The PARAFRASE Project's Fortran Analyser Major Module Documentation", CSRD Report 504, University of Illinois.
- Leung, B.P. (1990) "Issues on the Design of Parallelizing Compilers", CSRD Report 1012, University of Illinois.
- Leveson, N.G. (1986) "Software Safety: Why, What and How", ACM Computing Surveys, Vol. 18-2, pp: 125-163.
- Lewis T. G. and El-Rewini, H. (1992) "Introduction to Parallel Computing", Prentice Hall.
- Li, Z. (1989) "Intra-procedural and Inter-procedural Data Dependence Analysis for Parallel Computing", CSRD Rpt. No. 910, PhD thesis, University of Illinois.
- Li, Z. and Abu-Sufah, W. (1985) "A Technique for Reducing Synchronization Overhead in Large Scale Multiprocessor", Proc. 12th Int. Symp. on Computer Arch., Boston MA, pp: 284-291
- Li, Z. and Yew, P.C. (1988) "Efficient Inter-procedural Analysis for program Parallelization and Restructuring", Proc of the ACM/SIGPLAN PPEALS, pp: 85-99.
- Li, Z., Yew, P.C. and Zhu, C.Q. (1989) "Data Dependence Analysis on Multi-dimensional Array References", Proc. Int. Conf. on Supercomputing, Greece, pp: 215-224.
- Li, Z and Yew, P.C. (1990) "Some results on Exact Data Dependence Analysis", In Languages and Compilers for Parallel Computing (D Gelernter, A Nicolau and D Padua: Editors), Pitman, pp: 374-395.
- Luecke, G., Haque, W., Hoekstra, J., Jespersen, H. and Coyle, J. (1991) "Evaluation of Fortran Vector Compilers and

- Preprocessors", *Software-Practice and Experience*, Vol 21(9), pp: 891-905.
- Macke, T., Huson, C., Davies, J., Leasure, B. and Wolfe, M. (1986) "The KAP/ST-100: A Fortran Translator the ST-100 Attached Processor", *Proc. of Intl. Conf. on Parallel Processing*, pp: 171-175.
- Masterson, J.J., Ishaq, K., Patel, S., Norris, M.T. and Orr, R.A. (1988) "Intelligent Tools for Formal Specification", 2nd IEE/BCS Conf. Software Engin, pp: 149-153
- Maydan, D.E., Hennessy, J.L. and Lam, M.S. (1991) "Efficient and Exact Data Dependence Analysis", *Proc. of ACM SIGPLAN 1991 Conf. on Prog. Lang. Design and Impl. Toronto, Canada*, pp: 1-14.
- McCreary, C. and Gill, H. (1989) "Automatic Determination of Grain Size for Efficient Parallel Processing", *CACM Vol 32-9*, pp: 1073-1078.
- McParland, P. and Kilpatrick, P. (1988) "Software Tools for VDM", 2nd IEE/BCS Conf. Software Engin., pp: 154-158
- Meehan, D. (1990) "An Introduction to Fourth Generation Languages", Stanley Thornes.
- Midkiff, S.P. and Padua, D.A. (1986) "Compiler Generated Synchronization for DO Loop", *Proc. of 1986 Intl. Conf on Parallel Processing*, pp: 544-551.
- Midkiff, S.P. and Padua, D.A. (1987) "Compiler Algorithms for Synchronization", *IEEE Trans. on Computers*, Vol. C-36-12, pp: 1485-1495.
- Misra, J. and Chandy, K.M. (1981) "Proofs of Networks of Processes", *IEEE Trans. on SE*, Vol. SE-7, No. 4, pp: 417-426

- Mohd-Saman, M.Y. and Evans, D.J. (1993) "Investigation of a Set of Bernstein Tests for the Detection of Loop Parallelization", *Parallel Computing* 19, pp:197-207
- Moser, L.E., and Melliar-Smith, P.M. (1990) "Formal Verification of Safety-critical Systems", *Software-Practice and Experience*, Vol 20(8), pp: 799-821.
- Muchnick, S.S. and Jones, N.D. (1981) "Program Flow Analysis: Theory and Applications", Englewood Cliff, N.J, Prentice Hall.
- Osterhaug, A. (1987) "Guide to Parallel Programming", 2nd Edition, Sequent Computer Systems.
- Owicki, S and Gries, D. (1976) "Verifying Properties of Parallel Programs: An Axiomatic Approach", *Communication of the ACM* 19, 5, pp: 279-285
- Padua, D.A., Kuck, D.J. and Lawrie, D.H. (1980) "High-speed Multiprocessors and Compilation Techniques", *IEEE Trans. on Computers*, Vol. C-29, No. 9, pp: 763-776.
- Padua, D.A. and Wolfe, M.J. (1986) "Advanced Compiler Optimizations for Supercomputers", *CACM* Vol. 29, no 12, pp: 1184-1201.
- Perrot, R.H. (1987) "Parallel Programming", Addison-Wesley.
- Ploedereder, E. (1984) "Symbolic Evaluation as a Basis for Integrated Validation", *Software Validation*, H. L. Hausen (editor), North-Holland, pp: 167-185
- Polychronopoulos, C.D., (1988) "Parallel programming and Compilers", Kluwer Academics Publisher.
- Polychronopoulos, C.D., Girkar, M. B., Haghighat, M.R., Lee, C.L., Leung, B.P. and Schouten, D.A. (1990) "The Structure of

- Parafrase-2: an Advanced Parallelizing Compiler for C and FORTRAN", In Languages and Compilers for Parallel Computing (D Gelernter, A Nicolau and D Padua: Editors), Pitman, pp: 423-453
- Pountain, D. and May, D. (1987) "A Tutorial Introduction to OCCAM Programming", INMOS.
- Pugh, W. (1992) "A Practical Algorithm for Exact Array Dependence Analysis", Comm. of ACM, Vol. 35-8, pp: 102-114.
- Riseman, E.M. (1972) "The Inhibition of Potential Parallelism by Conditional Jumps", IEEE Transactions on Computer 21, no. 12, pp: 1405-1411.
- Sahni, S. (1984) "Scheduling Multi-pipeline and Multi-processor Computers", IEEE Trans. on Computers, Vol. C-33, No. 7, pp: 637-645
- Saltz, J.H., Mirchanhaney, R. and Crowley, K. (1989) "The DoConsider Loop", Proc. Int. Conf. on Supercomputing 1989, Greece, pp: 29-40
- Saltz, J.H, Mirchandaney, R. and Crowley, K. (1991) "Run-time Parallelization and Scheduling of Loops", IEEE Trans. on Computers, Vol. 40-5, pp: 603-611
- Sarkar, V. (1989) "Partitioning and Scheduling Parallel Programs for Multiprocessors", Pitman.
- Schouten, D.A. (1990) "An overview of Inter-procedural Analysis Techniques for high performance parallelizing compiler", CSRD Rpt. No. 1005, University of Illinois.
- Shen, Z., Li, Z. and Yew, P. (1989) "An Empirical Study on Array Subscripts and Data Dependencies", Proc. Int. Conf. on Parallel Proc., pp: II-145-152.

- Silberschatz, A. (1991) "Operating System Concepts", Addison-Wesley.
- Smith, K. and Appelbe, B. (1989) "Interactive Conversion of Sequential to Multitasking Fortran", Proc. Int. Conf. on Supercomputing, Greece, pp: 225-234.
- Sneed H.M. (1986) "Data Coverage Measurement in Program Testing" Proc. of Workshop on Software Testing, Canada, Computer Society Press, pp: 34-40
- Spyropoulos, C.D. (1978) "Analysis of Job Scheduling Algorithms for Heterogenous Multiprocessor Computing Systems", PhD Dissertation, Loughborough Univ. of Tech.
- Stroustrup, B. (1986) "The C++ Programming Language", Addison Wesley.
- Su, H.M. (1992) "On Multi-processor Synchronization and Data Transfer", CSRD Report 1176, University of Illinois.
- Tang, P., Yew, P.C. and Zhu, C.Q. (1990) "Compiler Techniques for Data Synchronization in Nested Parallel Loops", CSRD Rpt. No. 1092 University of Illinois USA, (Also in Proc. Int. Conf. on Supercomputing, Holland, Vol. 1, pp:177-186, 1990)
- Thakkar, S., Gifford, P. and Fielland, G. (1988) "The Balance Multi-processor System", IEEE Micro, pp: 57-69.
- Tjaden, G.S. (1970) "Detection and Parallel Execution of Independent Instructions", IEEE Transactions on Computer 19, no. 10 , pp: 889-895.
- Triolet, R. (1985) "Inter-procedural Analysis for Program Restructuring with PARAFRASE", CSRD Rpt. No. 538, University of Illinois.

- Triolet, R., Irigoin, F. and Feautrier, P. (1986) "Direct Parallelization of Call Statements", ACM/SIGPLAN Symposium on Compiler Construction. pp: 176-185.
- Tsuda, T. and Kunieda, Y. (1992) "V-Pascal: An Automatic Vectorizing Compiler for Pascal with No Language Extensions", in High Performance Computing: Research and Practice in Japan, Edited by R Mendez, John Wiley.
- Walmsley, S.W. and Williams, S.A. (1990) "Basically MODULA-2", Chartwell-Bratt.
- Wegman, M.N. and Zadeck, F.K. (1991) "Constant Propagation with Conditional Branches", ACM Trans. on Prog. Lang. and Syst., Vol. 13, No. 2, pp: 181-210.
- Welsh, J. and McKeag, M. (1980) "Structured System Programming", Prentice-Hall Int.
- Williams, S.A. (1978) "Approaches to the Determination for Parallelism for Computer Programs", PhD Loughborough University of Technology.
- Williams, S.A. (1990) "Programming Models for Parallel Systems", John Wiley.
- Williams, S. and Evans, D.J. (1980) "An Implicit Approach to the Determination for Parallelism", Intern. J. Computer Math, Vol. 8, pp: 51-59.
- Wolf, M. E. and Lam, M. S. (1991) "A Loop Transformation Theory and an Algorithm to Maximize Parallelism", IEEE Transactions on Parallel and Dist. Syst., Vol. 2-4, pp: 452-471
- Wolfe, M. (1986) "Advanced Loop Interchanging", Proc. of 1986 Int. Conf. on Parallel proc. pp: 536-543.

- Wolfe, M. (1988) "Multiprocessor Synchronization for Concurrent Loops", IEEE Software, pp: 34-42.
- Wolfe, M. (1989a) "Automatic Vectorization, Data Dependence and Optimizations for Parallel Computers" in Parallel Processing for Supercomputers and Artificial Intel. (Ed: K. Hwang and D Degroot), McGraw-Hill, pp: 409-440
- Wolfe, M. (1989b) "Optimizing Supercompilers for Supercomputers", Pitman London.
- Wolfe, M. (1990) "Loop Rotation", In Languages and Compilers for Parallel Computing (D Gelernter, A Nicolau and D Padua: Editors), Pitman, 1990, pp: 531-553.
- Wolfe, M and Banerjee, U. (1987) "Data dependence and its application to parallel processing", International Journal of Parallel Programming, Vol 16 No 2, pp: 137-178.
- Wordsworth, J.B. (1988) "Specifying and Refining Programs with Z", 2nd IEE/BCS Conf. Software Engin., pp: 8-16
- Young, M. and Taylor, R.N. (1986) "Combining Static Concurrency Analysis with Symbolic Execution" Proc. of Workshop on Software Testing, Canada, Computer Society Press, pp: 170-178
- Zima, H.P., Bast, H.J. and Gerndt, M. (1988) "SUPERB: A Tool for Semi-automatic MIMD/SIMD Parallelization", Parallel Computing 6, pp: 1-18, North-Holland
- Zima, H.P. and Chapman, C. (1990) "Supercompilers for Parallel and Vector Computers", ACM Press, Addison Wesley

APPENDIX A

TAG MAIN ROUTINE

TAG (Tool for Automatic Determination of Program Ganularity)

The following are the main routines:

ANALYZER - to form stanzas

DETECTOR - to perform the Bernstein Tests (BTs)

SCHEDULER - to schedule stanzas on a
shared-memory machine

MERGER - to merge stanzas

Note: Analyzer and Detector routines are similar to those described by Williams (1978).

*****/

tag()

```
{
    int i,j,k,maxpar,parent; float acc;

    analyzer(); /* scans input program & form stanzas */
    prtstanza(); /* prints stanzas */
    calset(); /* form WYZ and XYZ tables */
    findep(); /* perform the Bernstein Tests */
    maxpar = printcttab(); /* prints contemporay table */
    /* for proc=2 to max. parallel stanzas generate schedules */
    for (i=1; i<=maxpar; i++)
        acc = acctab[1][i] = genschedule(i); /* generate schedule */
    printacc(); /* prints speedup table */
    /* merging dependent stanzas */
    j = stanzacnt;
    parent = mergestanza(); /* merges stanzas */
    /* repeat scheduling & merging until no merging occurs */
    while (parent != j)
    {
        stanzacnt = parent; /* saves last count of stanzas */
        prtstanza(bst); /* prints new set of stanzas */
        calset(); /* forms new XYZ ans WYZ tables */
        findep(); /* performs BTs on new stanzas */
        maxpar = printcttab(); /* prints new contemporary table */
        /* for proc=2 to max. par. stanzas, gen. new schedules */
        for (k=1; k<=maxpar; k++)
            acc = acctab[1][k] = genschedule(k);
        printacc(); /* prints speedup table */
        j = stanzacnt;
        parent = mergestanza(); /* merge again */
    }
    printf("\n END of TAG\n");
} /* detector() */
```

APPENDIX B

THE SCHEDULER ROUTINE

```

/*
THE SCHEDULER ROUTINE

    - generating schedule for a shared-memory computer
    - take a group of contemporary stanzas
    - based on longest execution time first
        1. take a stanza with the longest exec time
        2. check its dependencies
        3. assign a cpu to it or delay it
*/

#include <stdio.h>
#include <string.h>
#include <ctype.h>
#include "parh.h"

#define UNDEF -9
#define INDEP -999

int deptime[100][20]; /* dependency table */
int cttab[100][20]; /* contemporary table */
float acctab[100][100]; /* speedup table */

struct schtab /* schedule table */
{
    int stno; /* stanza number */
    int begtime; /* begin time */
    int fintime; /* finish time */
};

struct cpustat /* cpu status info */
{
    int ctim; /* cpu latest stanza completion time */
    int cleve; /* and its level */
};

/*
    scheduler - assigning stanzas to cpus with optimum exec time
*/
float scheduler(win)
int win;
{
    int maxlev,maxcpu,maxtime,stpred,cpu;
    int stn,cgroup,st,i,j,k,l,m,level;
    int ilev,jcpu,sttim;
    int ct,stime,stpred2,jcpu2;
    int delay,gcnt,fcpu,cput,fint,ftime;

```

```

/* final schedule (level x processor) */
struct schtab schedule[1000][20];
struct cpustat cpucom[20]; /* cpu status information */
/* which cpu was st assigned - 0 level, 1 - cpu */
int cpust[20][2];
int stab[100]; /* temp area for sorting */
int contab[100]; /* list of stanzas already scheduled */
float acc;

/* Initialization */
maxcpu = 0; maxlev = 0;
for (i=0; i<=stanzacnt; i++)
{
    cpust[i][0] = UNDEF; /* cpu where st is assigned */
    contab[i] = 0; /* all stanzas marked as not scheduled yet */
    for (j=1; j<=stanzacnt+1; j++)
    {
        schedule[i][j].stno = UNDEF;
        schedule[i][j].begtime = UNDEF;
        schedule[i][j].fintime = UNDEF;
    }
}
for (i=1; i<=win; i++)
{
    cpucom[i].ctim = 0; /* cpu latest completion time */
    cpucom[i].cleve = -1; /* cpu latest completion time - level */
}

/* start with the 1st stanza in a concurrent group */
cgroup=0;
while (cgroup<=stanzacnt)
{
    k = 0;
    /* check if not scheduled yet */
    if (!contab[cgroup])
    {
        stab[k++] = cgroup;
        contab[cgroup] = 1; /* mark it as to be scheduled */
    }
    /* get the predecessor stanzas - for each, check if scheduled
    already or not */
    gcnt = cttab2[cgroup][0];
    for (i=1; i<=gcnt; i++)
        if (!contab[cttab2[cgroup][i]])
        {
            stab[k] = cttab2[cgroup][i];

```

```

    contab[stab[k++]] = 1; /* mark each one to be scheduled */
}

/* sort the out according to descending exec time */
gcnt = k-1;
for (i=0; i<gcnt; i++)
    for (j=i+1; j<=gcnt; j++)
    {
        l = stab[i]; m = stab[j];
        if (stanza[l].etime < stanza[m].etime)
        {
            for (k = j-1; k >= i; k--)
                stab[k+1] = stab[k];
            stab[i] = m;
        }
    }

/* assign stanza in stab starting with the one with
highest etime */
for (stn=0; stn<=gcnt; stn++)
{
    st = stab[stn];
    /* check if st is independent or not */
    if (deptable2[st][0] == 0)
    {
        /* st indep - can be scheduled on any available processor so
        find the lowest starting time */
        sttim = 999;
        for (i = 1; i <= win; i++)
        {
            if (cpucom[i].ctim < sttim)
            {
                sttim = cpucom[i].ctim; /* the lowest time */
                cpu = i;                /* and the proc. */
            }
        }
    }
    else
    { /* STANZA IS DEPENDENT ON OTHER STANZAS.
        st depend on stanzas in deptable2[st], so
        find the highest level st can be assigned &
        on cpu with the least load
        NOTE: stanzas in deptable2[st] must have already been
        assigned cpu
        */
        /* check if all predecessor stanzas have been assigned */

```

```

    delay = 0; /* flag to delay assignment of stanza or not */
    for (i=1; i<=deptable2[st][0]; i++)
    {
        stpred = deptable2[st][i]; /* predecessor stanza */
        if (cpust[stpred][0] == UNDEF)
        { /* predecessor not assigned yet
            so dont assign this stanza yet */
            delay = 1;
            break;
        }
    }
    if (delay) /* delay its assignment */
    {
        contab[st] = 0; /* reset as not scheduled yet */
        continue; /* go get next stanza in stab */
    } else
    { /* stanza st can be assigned find highest predecessor
        stanza finish time - ftime */
        ftime = -999; fcpu = 999;
        for (i=1; i<=deptable2[st][0]; i++)
        {
            stpred = deptable2[st][i]; /* predecessor stanza */
            ilev = cpust[stpred][0]; /* its level */
            jcpu = cpust[stpred][1]; /* its cpu */
            if (schedule[ilev][jcpu].ftime > ftime)
            { /* best time so far & its current finish time */
                ftime = schedule[ilev][jcpu].ftime;
                fcpu = cpucom[jcpu].ctim;
            } else
            if (schedule[ilev][jcpu].ftime == ftime &&
                cpucom[jcpu].ctim < fcpu)
            { /* same finish time but lower current proc. time */
                ftime = schedule[ilev][jcpu].ftime;
                fcpu = cpucom[jcpu].ctim;
            }
        }
    }

    /* finding the cpu with lowest starting time for st */
    sttim = 999;
    for (i=1; i<=deptable2[st][0]; i++)
    {
        stpred = deptable2[st][i]; /* predecessor stanza */
        jcpu = cpust[stpred][1]; /* and its cpu */
        stime = -1;
        for (j=1; j<=deptable2[st][0]; j++)
        {

```

```

        stpred2 = deptable2[st][j]; /* predecessor stanza */
        jcpu2 = cpust[stpred2][1]; /* and its cpu */
        if (jcpu == jcpu2)
            ct = cpucom[jcpu].ctim;
        else
        {
            ilev = cpust[stpred2][0]; /* and its level */
            ct = stanza[stpred2].ctime+schedule[ilev][jcpu2].fintime;
        }
        if (ct < ftime)
            ct = ftime;
        if (ct > stime)
            stime = ct;
    }
    if (stime < sttim)
    { sttim = stime; cpu = jcpu; }
}
}
}

/* finally save everything */
fint = sttim + stanza[st].etime; /* finish time */
level = cpucom[cpu].clev + 1;
schedule[level][cpu].stno = st; /* assign stanza to cpu */
schedule[level][cpu].begtime = sttim; /* its start time */
schedule[level][cpu].fintime = fint; /* and its finish time */
cpucom[cpu].ctim = fint; /* cpu latest completion time */
cpucom[cpu].clev = level;
cpust[st][0] = level; /* st is assigned at level level */
cpust[st][1] = cpu; /* st is assigned to proc cpu */
if (level > maxlev) /* maximum level for printing */
    maxlev = level;
if (cpu > maxcpu)
    maxcpu = cpu;
}
cgroup = cgroup + 1; /* get next group */
} /* while (cgroup<stanzacnt) */

/* prints schedule */
fprintf(fs, "\nSCHEDULE (no. of proc. = %d)\n", win);
/* find hingest finish time */
maxtime = 0;
for (i=1; i<=maxcpu; i++)
    if (cpucom[i].ctim > maxtime)
        maxtime = cpucom[i].ctim;
/* calculate sequantial time & save it in i */

```



```

calttime(&i,&j);
acc = (float) i/(float) maxtime; /* speedup value */
/* prints speedup value
   first, check if schedule has been printed before */
if (!maximacpu[maxcpu])
{
    maximacpu[maxcpu] = 1;
    /* print gantt chart */
    prtchart(maxcpu,maxlev,&schedule[0][0],maxtime);
    fprintf(fs,"\n==> TOTAL PAR execution time = %d",maxtime);
    fprintf(fs,"\n          SEQ execution time = %d",i);
    fprintf(fs,"\n          SPEEDUP          = %5.2f\n",acc);
}
return(acc); /* saved in a performance table - acctab */
} /* scheduler() */

/*
   prtchart - printing the gantt chart based on info in schedule[]
*/
prtchart(maxp,lev,schedule,maxt)
int lev,maxp,maxt; /* max. level & proc no. & max par time */
struct schtab schedule[1000][20];
{
    int i,j,k,l,maxft,mark,scale,last,chart[1000][30];
    int spc=-3, col=-2, row=-1, compact = 0;

    /* initialise chart to be empty */
    for (i=0; i<1000; i++)
    {
        for (j=0; j<=maxp*2; j++)
            chart[i][j] = spc;
    }
    maxft = 0; /* maximum height */
    for (i=0; i<=lev; i++)
        for (j=1; j<=maxp; j++)
            if (schedule[i][j].fintime > maxft)
                maxft = schedule[i][j].fintime;
    if (maxft > 40) /* check if too high */
    {
        compact = 1; /* yes, so set compact printing */
        for (i=1; i<=maxp; i++)
        {
            for (j=0; j<=lev; j++)
            {
                schedule[j][i].begtime = schedule[j][i].begtime / 2;
                schedule[j][i].fintime = schedule[j][i].fintime / 2;
            }
        }
    }
}

```

```

    }
    /* adjust */
    for (j=0; j<=lev; j++)
    {
        if (schedule[j][i].stno != UNDEF
            && schedule[j][i].fintime - schedule[j][i].begtime < 2)
        {
            last = schedule[j][i].fintime;
            schedule[j][i].fintime = schedule[j][i].begtime + 2;
            if (last == schedule[j+1][i].begtime)
                schedule[j+1][i].begtime = schedule[j][i].fintime;
        }
    }
}
/* for each stanza, setup its location on chart */
maxft = 0;
for (i=0; i<=lev; i++)
{
    k = 1;
    for (j=1; j<=maxp; j++)
    {
        /* put the '-' between begin and finish time */
        chart[schedule[i][j].begtime][k] = row;
        chart[schedule[i][j].fintime][k] = row;
        /* put the stanza no in chart */
        l = schedule[i][j].begtime;
        chart[l+1][k] = schedule[i][j].stno;
        if (l+1 == schedule[i][j].fintime)
            chart[schedule[i][j].fintime+1][k] = row;
        /* find the largest finish time */
        if (schedule[i][j].fintime > maxft)
            maxft = schedule[i][j].fintime;
        k = k+2;
    }
}
/* put the '-' for first and last row of chart */
for (i=0; i<=maxp*2; i++)
{
    chart[0][i] = row;
    chart[maxft][i] = row;
}
/* if (compact) maxft = maxft/2 + 1; */
/* put the 'l' for column marking */
i=0;
while (i<=maxft)

```

```

{
    for (j=0; j<=maxp*2; j=j+2)
    {
        chart[i][j] = col;
    }
    i = i+1;
}
fprintf(fs, "\nVERTICAL GANTT CHART maxft = %d (CPU = %d)\n",
        maxft, maxp);
/* printing the actual chart vertically */
fprintf(fs, "\n          ");
for (i=1; i<=maxp; i++)
    fprintf(fs, "PROC %2d  ", i);
    fprintf(fs, "\n\n");
k = 0; mark = 0;
if (compact) scale = 5;
else scale = 10;
for (i=0; i<=maxft; i++)
{
    if (compact) mark = k*2;
    else mark = k;
    if (i == maxft)
    {
        fprintf(fs, "%3d-", maxt);
        chart[i][maxp*2+1] = maxt;
    }
    else if ((k%scale) == 0 && mark < maxt)
    {
        fprintf(fs, "%3d-", mark);
        chart[i][maxp*2+1] = mark;
    } else
    {
        fprintf(fs, "%4c", ' ');
        chart[i][maxp*2+1] = -99;
    }
    k = k + 1;
    for (j=0; j<=maxp*2; j++)
    {
        if (chart[i][j] == row)
            fprintf(fs, "%s", "-----");
        else if (chart[i][j] == col)
        {
            if (chart[i][j-1] == row)
                fprintf(fs, "%s", "----|");
            else fprintf(fs, "%5c", '|');
        }
    }
}

```

```

        else if (chart[i][j] == spc)
            fprintf(fs,"%5c",' ');
        else fprintf(fs,"%5d",chart[i][j]+1);
    }
    fprintf(fs,"\n");
}
fprintf(fs,"\n\n");
fprintf(fm,"%3d\n",maxp);
for (i=0; i<=maxft; i++)
{
    for (j=0; j<=maxp*2+1; j++)
        fprintf(fm,"%5d",chart[i][j]);
    fprintf(fm,"\n");
}
return;
} /* prtchart */

```

APPENDIX C

THE MERGER ROUTINE

```

/*
MERGER ROUTINE:
STRATEGY:
    - merging 2 or more stanzas if all pred st are not merged yet
      with others
    - comm & size are main factors for consideration
*/

#include <stdio.h>
#include <string.h>
#include <ctype.h>
#include "parh.h"

int deptable2[50][20];          /* dependence table */
int gtab[maxstanza][maxstanza]; /* group to be merged */
int gtab2[maxstanza][maxstanza]; /* temp. group to be merged */
struct bernstien newst,newst2;  /* BSs after merging */

/*
MERGING MODULE: mergestanza - merging stanzas
*/
mergestanza()
{
    int i,j,gi,st,st1,st2,bm,cnt;
    int tt2,tt,bigst,bigsize;
    int mcnt,grain,ncnt;
    int mset[maxstanza];
    float gran;
    /* group of stanzas to be merge */
    int mtab[maxstanza][maxstanza];
    /* marker for stanzas not yet merged */
    int mmerg[maxstanza];
    struct bernstien *pst1;
    /* new stanza formed */
    struct bernstien nst[maxstanza];

    /* Initialise tables */
    for (i=0; i<=stanzacnt; i++)
    {
        mtab[i][0] = 0;    /* set counter */
        gtab[i][0] = 0;    /* set counter */
        mmerg[i] = 0;      /* marker - not merged yet */
    }
    st = 0;
    while (st<=stanzacnt)
    {

```

```

cnt = deptable2[st][0]; /* no. of stanzas st depend on */
bm = 0; mcnt = 0;
if (cnt>1) /* merge st with 2/more predecessor stanzas */
{ /* initialise set of stanzas to be merged as none */
for (i=1; i<=cnt+1; i++)
    mset[i] = -1;
    /* find the largest pred stanza */
    bigsize = -999; bigst = -888;
for (i=1; i<=cnt; i++)
{
    st1 = deptable2[st][i];
    if (mmerg[st1] > 0)
        bm = 1;
    if (stanza[st1].etime > bigsize)
    {
        bigsize = stanza[st1].etime;
        bigst = st1;
    }
}
    /* preds st are already merged with others or reserved
    so mark those not merged yet as reserved */
if (bm == 1)
{
    for (i=1; i<=cnt; i++)
    {
        st1 = deptable2[st][i];
        /* reserve all pred st */
        if (mmerg[st1] == 0) mmerg[st1] = 2;
    }
    if (mmerg[st] == 0) mmerg[st] = 2;
}
else
{ /* st & pred st may be merged - depends on comm + size */
for (i=1; i<=cnt; i++)
{ /* determine comm & size to find if need to merge */
    st1 = deptable2[st][i];
    if (st1 != bigst)
    {
        tt = stanza[st1].etime + stanza[st1].ctime;
        tt2 = bigsize + stanza[st1].etime;
        if (tt > tt2)
        {
            mset[++mcnt] = st1;
            bigsize = bigsize + stanza[st1].etime;
        } else mmerg[st1] = 2;
    } else mset[++mcnt] = st1;
}
}
}

```

```

    }
    mset[++mcnt] = st;
}
}
/* save mset in mtab for merging later */
if (bm == 0 && mcnt > 2)
{
    for (i=1; i<=mcnt; i++)
    {
        st2 = mset[i];
        mtab[st][i] = st2;
        mmerg[st2] = 1;
    }
    mtab[st][0] = mcnt;
    mmerg[st] = 1;
}
++st;
} /* while (st <= stanzaCnt) */

/* merging group of stanzas saved in mtab */
fprintf(fo, "\nMERGE GROUPS\n");
ncnt = -1;
st=0;
while (st<=stanzaCnt)
{
    if (mtab[st][0] > 2) /* more than 1 stanzas to be merged? */
    {
        for (i=1; i<=mtab[st][0]; i++)
        {
            st2 = mtab[st][i];
            fprintf(fo, "%d ", st2+1);
        } fprintf(fo, "\n");
        /*so merge them */
        ++ncnt;
        pst1 = &stanza[mtab[st][1]];
        /* merge all stanzas in the group */
        for (i=2; i<=mtab[st][0]; i++)
        {
            st2 = mtab[st][i];
            merge(pst1, &stanza[st2]); /* merge operation */
            pst1 = &newst2; /* result in newst */
        }
        gtab[ncnt][0] = mtab[st][0] + gtab[ncnt][0];
        gi = gtab[ncnt][0];
        if (gi == 0)
            for (j=1; j<=mtab[st][0]; j++)

```



```

        gtab[ncnt][++gi] = mtab[st][j];
    else
    for (i=1; i<=mtab[st][0]; i++)
    {
        for (j=1; j<=gtab[i][0]; j++)
            { ++gi; gtab2[ncnt][gi] = gtab[i][j]; }
    }
    /* save in nst: new stanza after merging */
    nst[ncnt].etime = newst.etime;
    nst[ncnt].ctime = newst.ctime;
    for (i=0; i<=3; i++)
    {
        nst[ncnt].bcnt[i] = newst.bcnt[i];
        for (j=0; j<=newst.bcnt[i]; j++)
            nst[ncnt].bset[i][j] = newst.bset[i][j];
    }
} else
if (mmerg[st] == 0 || mmerg[st] == 2)
{ /* st cannot be merged with any stanza ? */
    /* so copy the stanza into the new stanza */
    ++ncnt;
    if (gtab[ncnt][0] == 0)
        gtab[ncnt][1] = st;
    else for (j=1; j<=gtab[st][0]; j++)
        gtab2[ncnt][j] = gtab[st][j];
    gtab[ncnt][0] = gtab[st][0];
    nst[ncnt].etime = stanza[st].etime;
    nst[ncnt].ctime = stanza[st].ctime;
    for (i=0; i<=3; i++)
    {
        nst[ncnt].bcnt[i] = stanza[st].bcnt[i];
        for (j=0; j<=stanza[st].bcnt[i]; j++)
            nst[ncnt].bset[i][j] = stanza[st].bset[i][j];
    }
}
for (i=1; i<=stanzacnt; i++)
{
    if (i > ncnt) gtab[st][i] = 0;
    else
    {
        for (j=1; j<=gtab2[st][0]; j++)
            gtab[st][j] = gtab2[st][j];
        gtab[st][0] = gtab2[st][0];
    }
}
++st;

```

```

} /* while (st <= stanzacnt) */
fprintf(fo, "\n");
fprintf(fo, "\nNo of new stanzas: %d (Old=%d)\n", ncnt+1,
        stanzacnt+1);
/* ncnt < stanzacnt if ther were some merge op */
if (ncnt != -1 && ncnt != stanzacnt)
{ /* save back nst into stanza */
    grain = 0;
    for (st=0; st<=ncnt; st++)
    {
        grain = grain + nst[st].etime;
        stanza[st].etime = nst[st].etime;
        stanza[st].ctime = nst[st].ctime;
        for (i=0; i<=3; i++)
        {
            stanza[st].bcnt[i] = nst[st].bcnt[i];
            for (j=0; j<=nst[st].bcnt[i]; j++)
                stanza[st].bset[i][j] = nst[st].bset[i][j];
        }
    }
}
/* printing granularity size */
gran = (float) grain / (float) (stanzacnt+1);
fprintf(fo, "\nOLD Granularity size (%d/%d): %5.2f\n", grain,
        stanzacnt+1, gran);
if (ncnt != -1) stanzacnt = ncnt;
gran = (float) grain / (float) (stanzacnt+1);
fprintf(fo, "\nNEW Granularity size (%d/%d): %5.2f\n", grain,
        stanzacnt+1, gran);
return(ncnt); /* return no. of new stanzas */
} /* mergestanza */

```

```

checkbs(st, id)
struct bernstien *st;
int id;
{
    int val, i, ws;

    val = undef;
    for (ws=0; ws<=3; ws++)
    {
        for (i=0; i<=st->bcnt[ws]; i++)
            if (st->bset[ws][i] == id)
            {
                val = 200+ws;
                break;
            }
    }
}

```

```

    }
    if (val != undef)
        break;
}
return(val);
} /* checkbs */

/* merge - merging 2 stanzas */
/*      - result stanza in newst */
merge(st1,st2)
struct bernstien *st1,*st2;
{
    int i,j,k,wset,cnt,idx;

    for (i = 0; i <= 3; i++)
        newst.bcnc[i] = -1;
    if (st1 == st2)
    {
        newst.stanzatype = -1;
        newst.etime = st1->etime;
        newst.ctime = st1->ctime;
        for (i=0; i<=3; i++)
        {
            for (j=0; j<=st1->bcnc[i]; j++)
                newst.bset[i][j] = st1->bset[i][j];
            newst.bcnc[i] = st1->bcnc[i];
        }
        return;
    }
    newst.etime = st1->etime + st2->etime;
    newst.ctime = st1->ctime + st2->ctime;
    for (i=0; i<=3; i++)
    {
        for (j=0; j<=st1->bcnc[i]; j++)
        {
            idx = st1->bset[i][j];
            wset = checkbs(st2,idx);
            switch (i+200)
            {
                case setw: if (wset == setw || wset == undef)
                    k = 0;
                    else k = 2; break;
                case setx: if (wset == setx || wset == undef)
                    k = 1;
                    else k = 3; break;
                case sety: k = 2; break;
            }
        }
    }
}

```

```

        case setz: k = 3; break;
    }
    cnt = ++newst.bcmt[k];
    newst.bset[k][cnt] = idx;
}
}
/* repeat for j-th stanza to copy those vars not
   defined in i-th stanza*/
for (i=0; i<=3; i++)
{
    for (j=0; j<=st2->bcmt[i]; j++)
    {
        idx = st2->bset[i][j];
        wset = checkbs(st1,idx);
        if (wset == undef)
        {
            wset = ++newst.bcmt[i];
            newst.bset[i][wset] = idx;
        }
    }
}

/* copy new merged stanza into another one */
newst2.stanzatype = -1;
newst2.etime = newst.etime;
newst2.ctime = newst.ctime;
for (i=0; i<=3; i++)
{
    for (j=0; j<=newst.bcmt[i]; j++)
        newst2.bset[i][j] = newst.bset[i][j];
    newst2.bcmt[i] = newst.bcmt[i];
}
} /* merge */

```

APPENDIX D

THE BERNSTEIN LOOP TESTS

```

/*
DIRD.C      - Implementation of the Bernstein Loop Tests (BLTs)
              - which use the Data Reference Directions (DRDs) in
                handling the array references.
*/

#include <stdio.h>
#include <string.h>
#include <ctype.h>
#include "parh.h"
#define maxdepvar 100

/* stanzas in loops */
extern int loopstat[maxstanza][2];
extern int loopcnt; /* loop count */
extern struct deprec /* dependence record */
{
    int depst;
    char depvar[15]; /* id can causes dependence */
    int depdir[5]; /* and their directions */
} loopvar[maxdepvar];

/*
Detector    - the Bernstein Loop Tests
              - detects relationships between stanzas in LOOPS only
*/
int detector(bst)
int bst;
{
    int i,j,k,parcnt;
    float acc;

    calset(); /* Forms WYZ & XYZ sets tables */
    sloopdep(); /* find dependence in loops only */
    printf("\nEND of Bernstein Loop Tests (BLTs)\n");
} /* detector() */

/* calculate total seq. exec. time & total comm. time */
caltime(tt,ct)
int *tt, *ct;
{
    int i;

    *tt = 0; *ct = 0;
    for (i=0; i<=stanzacnt; i++)
    {

```

```

        *tt = *tt + stanza[i].etime;
        *ct = *ct + stanza[i].ctime;
    }
} /* caltime */

/*
printstanza - prints details of a stanza
*/
printstanza(bst)
int bst;
{
    int wset, max, i, j, k, ct, tt;

    /* prints all stanzas */
    fprintf(fo, "\nSTANZA # %13s%13s%13s%13s%8s%8s\n",
        "W sets", "X sets", "Y sets", "Z sets", "ETIME", "CTIME");
    for (i=1; i<=77; i++) fprintf(fo, "=");
    for (i=0; i<=stanzacnt; i++)
    {
        switch (stanza[i].stanzatype)
        {
            case 300 : fprintf(fo, "\n%2d    asgn", i+1); break;
            case 301 : fprintf(fo, "\n%2d    cond", i+1); break;
            case 302 : fprintf(fo, "\n%2d    then", i+1); break;
            case 303 : fprintf(fo, "\n%2d    else", i+1); break;
            case 304 : fprintf(fo, "\n%2d    for ", i+1); break;
            case 305 : fprintf(fo, "\n%2d    proc", i+1); break;
            case 306 : fprintf(fo, "\n%2d    read", i+1); break;
            case 307 : fprintf(fo, "\n%2d    writ", i+1); break;
            case 308 : fprintf(fo, "\n%2d    whil", i+1); break;
            case 309 : fprintf(fo, "\n%2d    rept", i+1); break;
            case 310 : fprintf(fo, "\n%2d    untl", i+1); break;
            default : fprintf(fo, "\n%2d    ", i+1); break;
        }
        max = stanza[i].bcnt[0];          /* find max. set content */
        for (j=0; j<=3; j++)
            if (stanza[i].bcnt[j] > max)
                max = stanza[i].bcnt[j];
        if (max == -1)
            fprintf(fo, "%13s%13s%13s%13s%8d%8d\n", "-", "-", "-", "-",
                stanza[i].etime, stanza[i].ctime);
        else
            for (k=0; k<=max; k++)
            {
                if (k > 0) fprintf(fo, "    ");
                for (wset=0; wset<=3; wset++)

```

```

        {
            if (k > stanza[i].bcnt[wset])
                fprintf(fo,"%13s","-");
            else fprintf(fo,"%13s",
                        symtab[stanza[i].bset[wset][k]].symname);
        }
        if (k == 0)
            fprintf(fo,"%8d%8d\n",stanza[i].etime,stanza[i].ctime);
        else fprintf(fo,"\n");
    }
}
for (i=1; i<=77; i++) fprintf(fo,"=");
fprintf(fo,"\n");
caltme(&tt,&ct); /* calculate sequential & comm times */
fprintf(fo,"%61s%8d%8d\n",
        "Total sequential time & communication time = ", tt,ct);
} /* printstanza */

/*
oropt - OR operation bet 2 sets
*/
oropt(bit1,bit2,bit3,bit4)
int bit1[], bit2[], bit3[],bit4[];
{
    int i,j,found;

    for (i=0;i<maxidcnt;i++)
        bit4[i] = bit1[i] | bit2[i] | bit3[i];
} /* oropt */

/*
getvar(id) - get the name variable for array
*/
getvar(id,idvar)
char idvar[maxkwlen];
int id;
{
    int i;

    for (i=0; (i<maxkwlen && symtab[id].symname[i] != '['); i++)
        idvar[i] = symtab[id].symname[i];
    idvar[i] = '\0';
} /* getvar */

/*

```



```

andopt - AND operation bet 2 sets
*/
andopt(bit1,bit2,bit3,dbit,dvar)
int bit1[], bit2[], bit3[];
char dbit[20][maxidcnt];
char dvar[20][maxkwlen];
{
    int i,j,k,l,found;
    char var1[maxkwlen], var2[maxkwlen];

    l = -1;
    for (i=0;i<20;i++) dbit[i][0] = -1; /* initialize results */
    for (i=0;i<maxidcnt;i++)
    {
        bit3[i] = bit1[i] & bit2[i]; /* normal AND operation */
        /* AND operation for array directions */
        if (bit1[i] == 2)
        {
            getvar(i,&var1[0]);
            for (j=0; j < maxidcnt; j++)
            {
                if (bit2[j] == 2)
                {
                    getvar(j,&var2[0]);
                    if (strcmp(var1,var2) == 0)
                    {
                        l++;
                        strcpy(dvar[l],var1);
                        for (k=0; k<5; k++)
                        {
                            switch (symtab[i].symdir[k])
                            {
                                case forward:
                                    switch (symtab[j].symdir[k])
                                    {
                                        case forward: dbit[l][k] = forward; break;
                                        case backward: dbit[l][k] = forback; break;
                                        case equal: dbit[l][k] = forward; break;
                                        case nodir: dbit[l][k] = nodir; break;
                                    } break;
                                case backward:
                                    switch (symtab[j].symdir[k])
                                    {
                                        case forward: dbit[l][k] = forback; break;
                                        case backward: dbit[l][k] = backward; break;
                                        case equal: dbit[l][k] = backward; break;

```

```

        case nodir: dbit[l][k] = nodir; break;
    } break;
case equal:
    switch (symtab[j].symdir[k])
    {
        case forward: dbit[l][k] = forward; break;
        case backward: dbit[l][k] = backward; break;
        case equal: dbit[l][k] = equal; break;
        case nodir: dbit[l][k] = nodir; break;
    } break;
    case nodir: dbit[l][k] = nodir; break;
} /* switch (symtab[i].symdir[k]) */
    }
}
}
}
}
} /* andopt */

```

```

/*
setbit - set the bit position for variables for OR opt
*/

```

```

setbit(bits,whatset,st)
int bits[], whatset, st;
{
    int i;

    for (i=0; i<maxidcnt; i++) bits[i] = 0; /* initialise */
    /* set proper bit to 1 */
    for (i=0; i<=stanza[st].bcnt[whatset-200]; i++)
        bits[stanza[st].bset[whatset-200][i]] = 1;
} /* setbit */

```

```

/*
setbit2 - set the bit position for variables for AND OPT
*/

```

```

setbit2(whatset,st,bits)
int bits[], whatset, st;
{
    int i;

    for (i=0; i<maxidcnt; i++) bits[i] = 0; /* initialise */
    /* set proper bit according to scalar or array types */
    switch (whatset)
    {

```

```

    case wyzs:
        for (i=0; i<=wyz[st].setcnt; i++)
            if (symtab[wyz[st].set[i]].symtype == scalar)
                bits[wyz[st].set[i]] = 1;
            else
                bits[wyz[st].set[i]] = 2;
        break;
    case xyzs:
        for (i=0; i<=xyz[st].setcnt; i++)
            if (symtab[xyz[st].set[i]].symtype == scalar)
                bits[xyz[st].set[i]] = 1;
            else
                bits[xyz[st].set[i]] = 2;
        break;
    }
} /* setbit2 */

/*
bitcount - counts no. of 1 in set bits
*/
int bitcount(bits)
int bits[];
{
    int i,bc;

    bc = -1;
    for (i=0; i<maxidcnt; i++)
        if (bits[i] == 1) bc++;
    return(bc);
} /* bitcount */

/*
calset - calculates Table (Xi or Yi or Zi) and (Wi or Yi or Zi)
*/
calset()
{
    int i,ibit,st,k,kk,icnt; char dchar;
    int bit1[maxidcnt], bit2[maxidcnt];
    int bit3[maxidcnt], bit4[maxidcnt];

    for (st=0; st<=stanzacnt; st++)
    { /* calculates set table (Wi or Yi or Zi) */
        setbit(bit1,setw,st);
        setbit(bit2,sety,st);
        setbit(bit3,setw,st);
        oropt(bit1,bit2,bit3,bit4); /* OR operation */
    }
}

```

```

k = 0;
wyz[st].setcnt = -1;
for (ibit=0;ibit<maxidcnt;ibit++)
    if (bit4[ibit])
    {
        wyz[st].setcnt++;
        wyz[st].set[k++] = ibit;
    }

/* calculates table (Xi or Yi or Zi) */
setbit(&bit1[0],setx,st);
oropt(bit1,bit2,bit3,&bit4[0]); /* OR operation */
k = 0;
xyz[st].setcnt = -1;
for (ibit=0;ibit<maxidcnt;ibit++)
    if (bit4[ibit])
    {
        xyz[st].setcnt++;
        xyz[st].set[k++] = ibit;
    }
}

/* prints contents of WYZ and XYZ tables */
fprintf(fo,"nCONTENTS of all WYZ and XYZ sets\n");
fprintf(fo,"nSTANZA # %24s%25s\n","WYZ sets","      XYZ sets");
for (i=1; i<=58; i++) fprintf(fo,"");
for (i=0; i<=stanzacnt; i++)
{
    fprintf(fo,"n%2d -      ",i+1);
    if (wyz[i].setcnt > xyz[i].setcnt)
        icnt = wyz[i].setcnt;
    else icnt = xyz[i].setcnt;
    if (icnt < 0)
        fprintf(fo,"%20s%22s\n","-","-");
    else
        for (k=0; k<=icnt; k++)
        {
            if (k > 0) fprintf(fo,"      ");
            if (k > wyz[i].setcnt) fprintf(fo,"%20s  ","-");
            else
            {
                fprintf(fo,"%20s ",symtab[wyz[i].set[k]].symname);
                for (kk=0; kk<5; kk++)
                    fprintf(fo,"%c",symtab[wyz[i].set[k]].symdir[kk]);
            }
            if (k > xyz[i].setcnt) fprintf(fo,"%20s\n","-");
        }
}

```

```

        else
        {
            fprintf(fo,"%20s ",syntab[xyz[i].set[k]].symname);
            for (kk=0; kk<5; kk++)
                fprintf(fo,"%c",syntab[xyz[i].set[k]].symdir[kk]);
            fprintf(fo,"\n");
        }
    }
}
for (i=1; i<=58; i++) fprintf(fo,"="); fprintf(fo,"\n");
} /* calset */

/*
sloopdep - determine dependence of loop iterations
*/
sloopdep()
{
    int st,i,j,lcnt;
    int bit1[maxident], bit2[maxident];
    int bit3[maxident];
    char dbit[20][maxident];
    char dvar[20][maxkwlen];

    /* initialization */
    for (st=0; st<=stanzacnt; st++)
    {
        loopvar[st].depst = -1; /* initialise stanza no */
        for (i=0; i<5; i++) /* initialise directions */
            loopvar[st].depdire[i] = nodir;
    }
    fprintf(fo,"\nLOOP DEPENDENCE ANALYSIS\n");
    fprintf(fo,"\n[DIRECTION symbols: < - forward, > - backward,");
    fprintf(fo," = - equal, * - <>]\n");
    /* to determine if loop are parallelizable or not */
    lcnt = 0; /* loop variables count that cause dependence */
    for (st=0; st<=loopcnt; st++)
    {
        fprintf(fo,"\n*** LOOP no. %d (stanza %d - %d) ***\n",
            st+1,loopstat[st][0]+1,loopstat[st][1]+1);
        for (i=loopstat[st][0]; i<=loopstat[st][1]; i++)
        {
            for (j=i; j<=loopstat[st][1]; j++)
            {
                fprintf(fo,"\nWYZ and XYZ for stanza %d & %d:",i+1,j+1);
                sdoandop(1,i,j,&lcnt); /* and op */
                if (j != i)

```

```

    {
        fprintf(fo, "\nWYZ and XYZ for stanza %d & %d:", j+1, i+1);
        sdoandop(2, i, j, &lcnt);
    }
    fprintf(fo, "\nXYZ and XYZ for stanza %d & %d:", i+1, j+1);
    sdoandop(3, i, j, &lcnt);
}
}
}
} /* sloopdep */

```

```

/*
AND operation for scalar and array with directions and
prints results
*/

```

```

sdoandop(tt, sti, stj, lcnt)
int tt, sti, stj, *lcnt;
{
    int i, j;
    int bit1[maxident], bit2[maxident], bit3[maxident];
    char dbit[20][maxident], dvar[20][maxkwlen];
    int pps = 1, ppa = 1;

    switch (tt) /* tt indicates which BTs test to perform */
    {
        case 1: /* WYZ(i) AND XYZ(j) */
            setbit2(wyzs, sti, &bit1[0]);
            setbit2(xyzs, stj, &bit2[0]);
            break;
        case 2: /* XYZ(i) AND WYZ(j) */
            setbit2(wyzs, stj, &bit1[0]);
            setbit2(xyzs, sti, &bit2[0]);
            break;
        case 3: /* XYZ(i) AND XYZ(j) */
            setbit2(xyzs, sti, &bit1[0]);
            setbit2(xyzs, stj, &bit2[0]);
            break;
    }
    /* normal bit operation and prints results */
    andopt(bit1, bit2, &bit3[0], &dbit[0][0], &dvar[0][0]);
    if (tt == 1 && sti == stj) pps = 1;
    if (pps && ppa)
    {
        if (bitcount(bit3) != -1)
        { /* test fails due to non-empty result set */
            fprintf(fo, "\nScalar dependence - ", sti+1);

```

```

    for (i=0; i<maxidcnt; i++)
    if (bit3[i] == 1)
    {
        fprintf(fo,"%s ",symtab[i].symname);
        loopvar[*lcnt].depst = sti;
        strcpy(loopvar[*lcnt++].depvar,symtab[i].symname);
    }
    else fprintf(fo,"\nNO scalar dependence");
}

pps = 1; ppa = 1;
if (tt == 3 && sti == stj) ppa = 0;
if (pps && ppa)
{
    fprintf(fo,"\nArray dependence : ");
    for (i=0; i<20; i++)
    {
        if (dbit[i][0] != -1)
        {
            loopvar[*lcnt].depst = sti;
            strcpy(loopvar[*lcnt++].depvar,dvar[i]);
            /* fprintf(fo,"\n%d. %s[" ,i+1,dvar[i]); */
            fprintf(fo,"%s[" ,dvar[i]);
            for (j=0; j<5; j++)
            {
                if (dbit[i][j] == forward || dbit[i][j] == backward ||
                    dbit[i][j] == equal || dbit[i][j] == forback)
                    fprintf(fo,"%c",dbit[i][j]);
                loopvar[*lcnt-1].depdir[j] = dbit[i][j];
            }
            fprintf(fo,"] ");
        } else break;
    }
    fprintf(fo,"\n");
} /* sdoandop */

```

