1991

A Theoretical Approach Involving Recurrence Resolution, Dependence Cycle Statement Ordering and Subroutine Transformation for the Exploitation of Parallelism in Sequential Code.

Chih-ping Chu
Louisiana State University and Agricultural & Mechanical College

Order Number 9207498

A theoretical approach involving recurrence resolution, dependence cycle statement ordering and subroutine transformation for the exploitation of parallelism in sequential code

Chu, Chih-Ping, Ph.D.

The Louisiana State University and Agricultural and Mechanical Col., 1991

U·M·I
300 N. Zeeb Rd.
Ann Arbor, MI 48106

A THEORETICAL APPROACH INVOLVING RECURRENCE RESOLUTION, DEPENDENCE CYCLE STATEMENT ORDERING, AND SUBROUTINE TRANSFORMATION FOR THE EXPLOITATION OF PARALLELISM IN SEQUENTIAL CODE

A Dissertation

Submitted to the Graduate Faculty of the
Louisiana State University and
Agricultural and Mechanical College
in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy

in

The Department of Computer Science

by

Chih-Ping Chu
B.S., National Chung Hsing University, Taiwan, R.O.C., 1971
M.S. in Computer Science, University of California at Riverside, 1987
August 1991

Acknowledgements

I would like to sincerely express my heartfelt thanks to Dr. Doris L. Carver for her untiring advice, patience, and encouragement in all stages of my dissertation research. I am also very appreciative to the members of my committee: Drs. Charles Harlow, Donald Kraft, Bush Jones, and A. Hoppe for serving as members and for the contributions they made to my research.

I am grateful to my parents, Chen-Chung Chu and Gin-Nin Pei, for their constant love, support and encouragement throughout my life. I also wish to express my appreciation to my wife, Mei Lin, who has endured much during my study towards a Ph.D. degree, yet has taken good care of our children. Without her sacrifice, I would not have been able to complete this dissertation. I thank my daughter, Yin-Je Chu, and my son, Chen-Lwen Chu, too. Their lovely smiles always make me forget the frustration encountered in my study.

Finally, I sincerely thank my friends in the Computer Science Department at Louisiana State University: Fenglien Lee for supporting me with some valuable research references and constructive discussions, and Ching-Liang Tseng, Sheng-Shyong Her, Cheng-Mei Wang, Abbas Dehkhoda, Linda Sherrell, Weibin Lu, Jigang Liu, Sangbum Lee, Maung Maung Htay, Kyung Hee Kwon, Shaolong Huang and Youwen Ouyang for their unfailing encouragement.

Table of Contents

List of Figures

Abstract

To exploit parallelism in Fortran code, this dissertation consists of a study of the following three issues: (1) recurrence resolution in Do-loops for vector processing, (2) dependence cycle statement ordering in Do-loops for parallel processing, and (3) subroutine parallelization.

For recurrence resolution, the major findings include: (1) the node splitting algorithm cannot be used directly to break an essential antidependence link, of which the source variable that results in antidependence is itself the sink variable of another true dependence so a correction method is proposed, (2) a *sink variable renaming* technique is capable of breaking an antidependence and/or output-dependence link, (3) for recurrences formed by only true dependences, a *dynamic dependence* concept and the derived technique are powerful, and (4) by integrating related techniques, an algorithm for resolving a general multistatement recurrence is developed.

The performance of a parallel loop is determined by the level of parallelism and the time delay due to interprocessor communication and synchronization. For a dependence cycle of a single parallel loop executed in a general synchronization mode, the parallelism exposed varies with the alignment of statements. Statements are reordered on the basis of execution-time of the loop as estimated at compile-time. An improved timing formula and a derived statement ordering algorithm are proposed. Further extension of this algorithm to multiple perfectly nested Do-loops with *simple global dependence cycle* is also presented.

The subroutine is a potential source for parallel processing. Several problems must be solved for subroutine parallelization: (1) the precedence of parallel executions

of subroutines, (2) identification of the optimum execution mode for each subroutine and (3) the restructuring of a serial program. A five-step approach to parallelize called subroutines for a calling subroutine is proposed: (1) computation of control dependence, (2) approximation of the global effects of subroutines, (3) analysis of data dependence, (4) identification of execution mode, and (5) restructuring of calling and called subroutines. Application of these five steps in a recursive manner to different levels of calling subroutines in a program addresses the parallelization of subroutines.

Chapter 1

Introduction

1.1 Vector- and Multi-processing: Past, Current and Future

The demand of high speed computation in engineering and science has been one of the principal driving forces in the development of powerful computers. Many important applications such as simulation of distributed parameter systems, signal processing, real-time simulation of dynamic systems, and artificial intelligence are infeasible if the data processing speed cannot meet the actual requirement. For example, image processing of radar signals involves the digitizing of analog signals and the processing of the resulting data. This processing entails the application of mathematical transformations, such as the fast Fourier transform, in order to facilitate auto- and cross-correlations. The input signals arrive at very high rates, and the output must be generated very rapidly in order to be useful [28].

To overcome the limitation of speed of conventional von Neumann computers, two major innovations in the past 20 years in computer engineering have resulted in significant improvement in computing speed: (1) the advances of hardware technology and (2) pipelining and parallelism design. Computers of the third-generation (1964-1969) such as the IBM S/360 series and the CDC 6600 has a clock speed of about 100 nanoseconds. The Cray 2 supercomputer in 1985 has a clock speed of about 4 nanoseconds. Thus, device speed has increased 25 times in the past 20 years. New materials, such as gallium arsenide devices are expected to increase clock speed by a factor of 5 in the next 5 to 10 years [28,47,51].

Pipelining and parallelism design in computer architecture have led to even greater contributions to computer performance. During the 1970s, the development of the supercomputer focused on the pipeline architecture: instruction pipelining, data pipelining, and pipelined functional units. Instruction pipelining is the use of pipelining to allow more than one instruction to execute at different stages at the same time. Instruction pipelining is achieved by dividing the execution of each instruction into a number of phases, such as fetch, decode, operand fetch, and instruction execute. The data pipeline allows vectors to be streamed from memory into the CPU where they are manipulated by pipelined functional units. In pipelined functional units, the stages of instruction execution are overlapped to reduce execution time for a given operation [27,47]. The first pipelined supercomputer to win widespread acceptance was the Cray-1, which contains both vector and scalar processors. Other contemporary supercomputers such as CDC Cyber-200 series, Fujitsu VP-200, and Hitachi S-810 have a similar approach to high-speed computation. These supercomputers can provide peak performance of several hundred million floating-point arithmetic operations per second (Mflops). However, the pipelined architecture has its limitations. To gain the speedup, the start-up delay of vector pipelines has to be offset via sufficiently long vector operations. The vector break-even length (the minimum vector length for the vector mode of a vector processor surpassing the equivalent scalar mode) for some supercomputers of this pattern is about one hundred [28,51]. But multiple pipelined vector processors like the IBM 3090/VF and the Cray X-MP have much shorter break-even lengths of approximately 10 and 2, respectively [15].

Instruction and data pipelining achieve parallelism at the inter-datum and inter-instruction levels. To further enhance the speedup, parallelism at the job or process level needs to be exploited. Multiprocessing is a method to achieve such a goal. This concept has led to the development of multiprocessor supercomputers. Multiprocessor systems manifest a wide variety of architectural approaches and innovative concepts. Based on the forms of information exchange between processing elements, multiprocessor supercomputers can be classified as shared-memory systems, message-passing systems, and hybrid shared-memory and message-passing systems.

Shared-memory systems are tightly coupled MIMD (multiple-instruction stream and multiple data stream) machines using shared memory among multiple processors. The Cray X-MP, which is extended from the Cray-1 and consists of two processors, is a full scale typical shared-memory multiprocessor supercomputer system. It adopts a direct connect between distinct processors. Another full scale shared-memory systems in this configuration is the Cray-2, which has 4 processors with peak performance of 2 billion floating-point operations per second (Gflops). The full-scale shared-memory multiprocessor systems are very expensive. To reduce the cost, some other shared-memory near super and minisuper computer systems have been constructed. These systems, adopting direct or bus connection between distinct processors, have relatively slower computation speed, but may have higher performance/cost ratio. Representative systems include the IBM 3090/400 VF (near-super computer), Alliant FX/8 and Encore/Multimax (minisuper computer) [28].

Message-passing multiprocessor systems correspond to loosely coupled MIMD systems with distributed local memories attached to multiple processor nodes. There

exist no shared memory units in the system. Data to be exchanged are transmitted as messages between two processors. The interconnection methods for the multiprocessors include the hypercube, ring, butterfly, mesh, pyramid, hypertrees, and hypernets. Currently there does not exist a full-scale message-passing multiprocessor supercomputer system. The existing message-passing systems are classified as near super or minisuper computers. Representative systems include the CDC Cyberplus (near-super) which has 64 processors connected via ring network with peak performance of 65 Mflops per processor; the NCUBE/10 (minisuper) which has 1024 processors connected via hypercube network with peak performance of 500 Mflops; and the FPS T-Series (minisuper) which has 4096 processors connected via hypercube network with peak performance of 16 Mflops per processor [28,47].

The hybrid shared-memory and message-passing multiprocessor systems include several experimental supercomputers, such as the the Cedar, IBM RP3, and ETA-10 systems. All of these systems, currently under development, emphasize a high degree of parallelism. ETA-10 is a full-scale supercomputer with expected peak performance of 10 Glops [28,51].

At present, Cray Research is developing a shared-memory multiprocessor Cray-3, which has 16 processors with expected peak performance of 16 Gflops. Some supercomputer research projects are challenging with a system with hundreds of processors, each of which is a 32-bit microprocessor equipped with vector accelerators [28].

1.2 Software Features for Vector- and Multi-processor Systems

In contrast to hardware progress, parallel software development is far behind. It is relatively easier to build a high-performance computer architecture than it is to develop software that effectively exploits the parallelism. So far, programming for vector and multi-processor systems is a rather unfamiliar process and a subject that is currently under study. Most of the programs run on today's supercomputers achieve less than 20% of their peak performance [57].

The slow progress of parallel software development on vector- and multi-processor systems has lead to the need for special environments for parallel software development. The primary features of parallel software development environments are parallel algorithms, parallel languages, parallel compilers, and parallel programming environments.

1.2.1 Parallel Algorithms

There are at least two ways to develop a parallel algorithm: (1) to detect and exploit the inherent parallelism in an existing sequential algorithm, and (2) to develop a new parallel algorithm. Humans tend to solve problems via a sequential procedure rather than a concurrent approach. Developing a sequential algorithm and then further exploiting its potential parallelism is a straightforward way to design a parallel algorithm. However, as some sequential algorithms have no potential parallelization, the parallel algorithm derived is obviously inefficient. In that case, the development of a new parallel algorithm is required [47].

In multiprocessor systems, parallel algorithms are highly machine-dependent.

On shared memory systems, algorithms are relatively easy to design. One simply puts the data in memory as if programming on a uniprocessor. The algorithm on a message passing system is quite different from that used on a shared memory system for two reasons. First, data must be explicitly moved from the memory of one processor to the memory of another. Second, there is often no master processor to spawn tasks. This second difference is due in part to the distributed memory. Thus, algorithm design is hard because the data must be distributed so that communications traffic is minimized. Hybrid systems are programmed like shared memory systems, but have data access delays like message passing systems. Algorithm design on hybrid systems is the most difficult [30]. In any event, programmers of parallel programs need to be highly educated in the development of parallel algorithms.

1.2.2 Parallel Languages

Programming languages have a substantial impact on the development and complexity of algorithms. The parallel language also determines the grain size of parallelism exposed in the programs. For example, in vector languages, such as Fortran 8X, the vector statements support statement-level parallelism. In parallel languages, such as Parallel Fortran, the **parallel loop** construct supports task (i.e., iteration) level parallelism. Currently, most parallel languages are created by extending sequential languages with parallel constructs. Examples are the vector and parallel dialects of Fortran and Pascal [45].

The features of parallel languages for multiprocessor systems are related to the embedded machine's architecture. In the shared memory systems, extensive language

modifications are needed. That is, there is a need to distinguish explicitly the privacy or sharing of data, to express parallel statements, to manage concurrent processes in a variety of modes, and to control the mechanism of synchronization. These features cause the shared memory systems to allow an entirely different style of programming. In message-passing systems, only two basic functions, **send** and **receive**, need to be added to the standard language support for communication among processes [30]. So far, there is not a general consensus as to the characteristics that a parallel language should possess.

1.2.3 Parallel Compilers

A parallel compiler is software that translates programs in a high-level sequential or parallel language into machine code for parallel processing. In general, a parallel compiler does the following tasks, based on the language constructs [1,46,51]:

- checks the correctness of syntax and semantics
- establishes global data dependence relations between statements
- generates sequential instructions for sequential constructs
- detects parallelism and generates vector instructions and/or partitions into concurrent processes for vectorizable (parallelizable) constructs
- generates communication and synchronization instructions for concurrent processes
- generates code for creation and termination of concurrent processes
- generates code for processor scheduling

At present, parallel compilers are capable of parallelizing and/or vectorizing the serial construct to some extent. In general, the compilers transform eligible loops into parallel and/or vector code. Parallelization of other parallel constructs such as parallel case, parallel loop, and parallel tasks is also available for some parallel compilers [29].

1.2.4 Parallel Programming Environments

A parallel programming environment includes programming tools to assist programmers in developing efficient and reliable programs for vector and multiprocessor systems. In general, coding based on parallel algorithms is an unnatural, tedious, and error-prone process. Certain bugs, such as race conditions, deadlock, and unreproducibility, are easily introduced. In addition, parallel program testing, debugging and validating are not trivial. For example, in terms of debugging, programs developed on shared memory systems are hard to debug because it is difficult to identify when an error occurred. This situation implies that in addition to vectorizing and parallelizing compilers, tools for program testing, debugging and validating are also indispensable for parallel software development.

1.3 Problem and Motivation

The high-speed computer is primarily developed for solving engineering and science problems. In reality, this new generation of computers also can be used for the advancement of the performance of existing software systems. For the past 20 years, a large number of scientific programs has been developed. It is unrealistic to expect programmers to redesign those programs for vector- or multi-processing. Programmers

may not even understand the source code because it was written by someone else in an age without a sound engineering approach to program development. Software engineers of the new generation of computers thus face a challenge: how to exploit the parallelism in existing code without rewriting and redesigning the sequential program. A direct approach is to develop automatic techniques, translating serial code into vectorized or parallelized code. Another method is to develop semi-automatic interactive tools which help the programmers tune the programs such that potential parallelism can be fully exploited. In fact, both approaches are valuable.

Semi-automated tools favor the thorough exploitation of parallelism from serial code [5,7]. Users of semi-automatic tools can be highly trained to tune their code and thus learn the special programming style for vector/parallel coding. But, the semiautomatic nature introduces another problem, the intervention of manual coding. This can make transformation of code time-consuming and the generated code unreliable. Moreover, parallel programs are notoriously difficult to debug and validate. These factors can severely limit the use of semi-automated tools.

On the other hand, automated tools may not be able to exploit fully the parallelism in the code because some parallelism depends on the programmer's assertions which, in general, cannot be determined currently by any techniques. In addition, automated tools isolate programmers from learning the coding style for vector/parallel processing; however, programmers for vector and multiprocessor should not totally escape thinking about parallelism exploitation [3]. However, the automatic approach saves considerable time and generates reliable code.

In the future, we believe software engineers should develop powerful automated techniques to transform serial code for vector/parallel processing. That is, the intervention of manual coding should be reduced to a minimum. This approach is not only required for dealing with existing code, but also needed for the development of vector/parallel software. Programming for parallel machines is very complicated. A way to simplify programming for vector/parallel machines is to free the programmer from having to understand the complications and let automatic software take over the difficult tasks. Programmers then simply need to use high-level parallel languages to implement the algorithms in a straightforward manner.

Considerable efforts have been devoted to the theories and techniques of parallelism detection and exploitation [2,6,9,11-14,16,18,22,23,32,33,39,40,42-45,51,53]. To date, many automatic sequential-code to parallel-code translators and compilers, and semi-automatic parallel-code transformation tools are available. For example, commercialized vectorizing and parallelizing compilers include Alliant FX/8, BBN Butterfly, Cray X-MP, Encore Multimax, IBM VS Fortran, and Parallel Fortran; semi-automatic parallel programming tools include PTOOL, ParaScope, Pat, and Start/Pat; automatic source to source translators include PFC, KAP/S-1, KAP/205; some research parallelizing compilers include Parafrase, PTRAN, and PFC. These translators, compilers or parallel programming tools are able to identify vectorizable and/or parallelizable iteration constructs such as Fortran Do-loops and to partially exploit the parallelism. Some compilers are even capable of determining the optimum execution mode [2,4,5,7,13,17,19,24,26,46,50]. However, the general low ratio (20%) of the peak performance achieved by today's parallel application software implies that

there is still much to be achieved.

Fortran is a language characterized by its execution efficiency. It has been used effectively in engineering and science fields. In the past years, industry has invested a tremendous amount of resources in Fortran programmers, tools, and libraries. The Fortran code is unlikely to be ignored in attempts to deal with automatic parallelization of existing serial code. In fact, most research on parallelism exploitation of serial code uses Fortran or Fortran dialects as a template. In this dissertation, Fortran 77 is used as a sequential language to study techniques of parallelism exploitation.

1.4 Proposed Research and Organization of The Dissertation

In this dissertation we study three problems of sequential code transformation for vector- and multi-processing: **recurrence resolution, dependence cycle statement ordering,** and **subroutine parallelization.**

Dependence recurrences are a type of data dependence relation that occurs in statements of repetitive language constructs such as Fortran Do-loops. The performance of vectorization and parallelization of language constructs with such a relation is reduced or even totally inhibited. Existing techniques to resolve recurrence relation include node splitting, thresholding, cycle shrinking, and library calls to a linear recurrence solver, depending on the types of dependences [6,37,45]. A systematic approach to handle this problem is not observed. In Chapter 2 we study the resolution of recurrences in Fortran Do-loops with the aim of developing a systematic strategy to handle the vectorization of statements. Some techniques developed for breaking the dependence cycles are also applicable to code transformation for concurrent process-

ing.

Iteration language constructs are suitable either for vector processing or for parallel processing. In fact, parallel computing combined with vector processing is the best way to enhance performance. On shared memory multiprocessor systems, the performance of a parallel loop depends primarily on (1) the level of parallelism and (2) the process scheduling strategy. Obviously, a given loop with maximum exposed parallelism associated with an optimum loop-iteration scheduling policy leads to the best performance in parallel processing.

To exploit the parallelism, the statements in the loop should be aligned properly (without modifying the semantics) such that potential parallelism is explicitly exposed. For a dependence cycle, finding an alignment of statements with maximum parallelism is not trivial. Chapter 3 studies statement ordering algorithms for parallel loops on the basis of estimation of execution time at compile-time.

In addition to statement-level parallelism, to fully utilize the embedded parallel hardware, code transformation for coarse-grain parallelism, such as subroutine parallelization, is also needed. Most research in this area focuses on the approach of asynchronous parallelization of subroutines. To speedup the parallel processing of a program, subroutines executable in synchronous mode also need to be exploited. In Chapter 4, we discuss the parallelization approach of subroutines executed in asynchronous and synchronous modes under shared memory systems.

The results obtained from these research can be applied to develop efficient vector/parallel compiler and source to source translators. Chapter 5 contains a summary of this research, concluding remarks, and a discussion of future research.

Chapter 2

Recurrence Resolution in Do-loops for Vector Processing

2.1 Introduction

In the execution-time profile of a scientific or engineering Fortran program, the Do-loop construct in general occupies a significant portion of time. In vector processing, the Do-loop is the primary construct. Thus, to improve the performance of a program executed in a vector processor system, a practical approach is to speed up the processing of Do-loops.

The speedup of a Do-loop in vector processing is determined by two factors: (1) the length of the loop bound, and (2) the level of parallelism in the statements. To gain the speedup, the length of the loop bound must be longer than the break-even length (the minimum vector length for the vector mode of a vector processor surpassing the equivalent scalar mode) of the underlying machine [15]. That is, only when the length of the loop bound is over the break-even length of the embedded machine will the loop be eligible for vector processing.

The level of parallelism in a Do-loop depends on data dependence in statements. To exploit the parallelism we thus need to develop (1) an accurate data dependence analysis technology, and (2) a systematic approach to remove undesired data dependence. Some optimization techniques which favor the development of (1) or (2) are also required.

For the past twenty years, many algorithms regarding data dependence testing,

undesired data dependence elimination, and Do-loop optimization have been developed [6,9,12,17,36,37,39,40]. A majority of the dependence testing algorithms operate on array variables with linear subscript expressions; some are especially for array variables with partially linear and nonlinear subscript expressions; several others are for array variables with coupled subscripts (i.e., the loop index appears in more than one dimension). The systematic approaches to remove undesired dependence include *scalar renaming, scalar expansion, scalar forward-substitution, node splitting,* and *dead code elimination.* Other optimization techniques which simplify the dependence test involve *loop normalization, induction variable substitution, loop rerolling, expression forward substitution,* and *constant folding* [6,17,36,37,49]. To date, many vectorizing compilers (or translators) are capable of exploiting the parallelism of a general Do-loop to a certain extent.

In the process of vector compilation, recurrences are generally considered as not suitable for vector processing. A recurrence is a type of dependence relation consisting of strongly connected statements extracted at the back-end phase *loop distribution* [6,17,37]. A linear recurrence may be processed via a library call to a linear recurrence solver [17,37]. Some techniques such as *node splitting, cycle shrinking,* and *thresholding* are proposed to break recurrences or partially vectorize the statements involved in the recurrences [6,37,45].

Undoubtedly, recurrence breaking is one approach for increasing the level of parallelism. However, the dependence patterns in recurrences are possibly complex, and therefore effective transformations of a general recurrence for vectorization is not trivial. A systematic study to this problem, which involves maximum parallelism

exposure, efficient resolving techniques, and a general algorithm is needed.

In this chapter, we analyze the recurrences from the breakability of the dependence links, derive three patterns of links, and identify each pattern's breaking strategy and its theoretical evidences. An algorithm to resolve a general recurrence is proposed. In Section 2.2, we establish the concepts of data dependence and the relationships with vectorizability. Section 2.3 provides an analysis of the formation of recurrences and derives three dependence links on the breaking-strategy basis. For each link pattern, its features, resolving technique, and application details are introduced. In Section 2.4 we develop an algorithm for the resolution of a general multistatement recurrence. Finally, Section 2.5 contains a summary.

2.2 Data Dependences and Vectorizability

Data dependences, which are generated from coding practices, may exist either interstatement or intrastatement. Kuck and others [35], based on Bernstein's conditions [2], have defined three types of dependence, i.e., true dependence, antidependence, and output dependence, that exist in statement(s). If a statement S_j uses the variable defined first by another statement S_i, then S_j is true-dependent on S_i. If S_j can define a variable only after that variable is first used by S_i, then S_j is antidependent on S_i. If S_j can redefine a variable only after that variable is first defined by S_i, then S_j is output-dependent on S_i. Another dependence, control dependence, which arises due to control statements, is not addressed in this chapter.

Each of these three types of dependence refers to an ordered access to a common memory location. The true dependence is a dependence of values of variables, which

cannot be replaced by any existing technique. The other two types of dependence are in fact storage-dependent. They can be eliminated without changing the semantics by supplying additional temporary variables [22]. But, these three basic types of dependence are equally important in terms of semantics preservation. In general, when modifying or reordering the statements inside a Do-loop for vectorization or parallelization, the original relationship of serial data dependences must be preserved.

These data dependences may result in loop-independent and loop-carried data dependence [6] when they exist in statement(s) with indexed variables in a Do-loop. Loop-independent dependence refers to the dependence confined within each single iteration, while loop-carried dependence implies the dependence occurring across the iteration boundaries. The loop-carried data dependence can be further distinguished as consistently loop-carried data dependence and inconsistently loop-carried data dependence, depending on whether the dependence is consistent across the loop. Since vector quantity types are the main data types in vector processing, scalar data may be expanded to vector type in Do-loops before operation [16,35,36]. In the remainder of this chapter, our discussion of dependence implicitly refers to the relation existing in statement(s) with indexed variables such as array variables inside a Do-loop. More formally, we introduce the following definitions.

Definition *Loop-independent dependence* (denoted Δ) includes loop-independent true dependence (denoted δ), loop-independent antidependence (denoted $\bar{\delta}$), and loop-independent output dependence (denoted δ^o). These relations are represented by the set (denoted Δ), i.e., $\Delta = \{\delta, \bar{\delta}, \delta^o\}$.

Definition *Consistently loop-carried dependence* (denoted [Δ]) includes consistently loop-carried true dependence (denoted [δ]), consistently loop-carried antidependence (denoted [$\bar{\delta}$]), and consistently loop-carried output dependence (denoted [δ^o]). These relations are represented by the set [Δ] = {[δ], [$\bar{\delta}$], [δ^o]}.

Definition For two statements S_i and S_j in a Do-loop, where $S_i \neq S_j$, S_i $\delta / \bar{\delta} / \delta^o$ S_j (S_j is loop-independent true/anti/output dependent on S_i) implies that for an instance of dependence there exist n index values $x_1, x_2, \ldots, x_n$, where $1 \leq n \leq N$ (loop upper bound) and $1 \leq x_i \leq N$, such that the statement S_j of the x_ith iteration of the loop is true/anti/output dependent on the statement S_i of the x_ith iteration of the loop. For all other iterations S_i and S_j are independent of each other.

Definition For two statements S_i and S_j in a Do-loop, where S_i and S_j are not necessarily different, S_i [δ]/[$\bar{\delta}$]/[δ^o] S_j (S_j consistently loop-carried true/anti/output dependent on S_i) implies that for an instance of dependence there exist k pair(s) of index values x_i and y_i, where $1 \leq k \leq N-1$ (N is loop upper bound) and $1 \leq x_i < y_i \leq N$, such that the statement S_j of y_ith iteration of the loop is true/anti/output dependent on the statement S_i of x_ith iteration of the loop. For all other iterations S_i and S_j are independent of each other.

Definition For two statements S_i and S_j in a Do-loop, where S_i and S_j are not necessarily different, if S_i and S_j are *inconsistently loop-carried dependent* on each other, then for this particular instance of dependence there exist multiple inconsistent dependence relations between S_i and S_j. That is, there exist at least two pairs of index values x_i and y_i, where $1 \leq i \leq 2$ and $1 \leq x_i \leq y_i \leq N$, such that statement S_j of y_ith iteration and the statement S_i of x_ith iteration are dependent each other but the two dependence

relations are inconsistent.

Figure 2.1 presents an example showing the two statements S_1 and S_2 are inconsistently loop-carried dependent on each other. S_2 at the 3rd iteration is true-dependent on S_1 of the 2nd iteration; but S_1 at the 6th iteration is antidependent on S_2 of the 5th iteration.

```
Do  10  I = 1, 10
S1  A(I+3) = C(I) + 1
S2  B(I) = A(2*I-1) + 3
10  Continue
```

Fig. 2.1 Inconsistently loop-carried dependence

In terms of loop-independent dependence, due to the execution order of the instructions, a single statement may be anti-dependent on itself but can never be true-dependent or output-dependent on itself. Thus we have the following definition:

Definition For statement S_i in a Do-loop, $S_i \bar{\delta} S_i$ (S_i is loop-independent anti-dependent on itself) implies that for an instance of dependence there exist n index values $x_1, x_2, \ldots, x_n$, where $1 \le n \le N$ (loop upper bound) and $1 \le x_i \le N$, such that the statement S_i of x_ith iteration of the loop is anti-dependent on itself at the same iteration of the loop. For all other iterations S_i does not depend on itself.

The intrastatement dependence relations inside a Do-loop thus can be classified as:

(1) no dependence,

(2) loop-independent anti-dependence,

(3) consistently loop-carried true-dependence,

(4) consistently loop-carried anti-dependence,

(5) consistently loop-carried output-dependence, and

(6) inconsistently loop-carried dependence.

Similarly, the interstatement dependence relations can be classified as:

(1) no dependence,

(2) loop-independent true-dependence,

(3) loop-independent anti-dependence,

(4) loop-independent output-dependence,

(5) consistently loop-carried true-dependence,

(6) consistently loop-carried anti-dependence,

(7) consistently loop-carried output-dependence, and

(8) inconsistently loop-carried dependence.

Therefore, a section of serial code in a Do-loop can be vectorized or parallelized as long as these types of dependence present in statement(s) can be preserved. Some dependence relations may prevent immediate vectorization or parallelization of code; other relations may even totally inhibit vectorization or parallelization. However, it is possible to remove some redundant dependences or reposition certain undesired dependence arcs for vectorization without changing the semantics [37]. Usually, a variable with a multiple purpose results in redundant dependences. Such dependences can be eliminated via introducing more variables to unify each variable's purpose. Techniques like scalar renaming and scalar expansion are based on this concept. Whereas, a node splitting method is used for repositioning the undesired dependence arcs [17,36,37].

To study the data dependence and vectorizability for a Do-loop, we first describe briefly the features of the execution of vector mode, and then analyze the possible

dependence relations existing in statements of code. For each dependence relation, we examine its characteristics regarding vectorization. The vectorizability for statements with inconsistently loop-carried dependence depends on the component dependences, which in fact are consistently loop-carried and loop-independent dependence. In this chapter we do not address inconsistently loop-carried dependence.

Definition For an interstatement or intrastatement dependence, the *source variable* of the dependence implies the instance of the indexed variable to be accessed first; the *sink variable* of the dependence implies the instance of the indexed variable to be accessed later.

Definition The relation of an interstatement or intrastatement dependence can be represented graphically by a dependence arc. The direction is from the source variable to the sink variable.

Definition Let S_i and S_j be two statements that appear in a Do-loop. We say that $S_i < S_j$ if S_i appears first in the loop and $S_i \neq S_j$ [6].

Definition A normalized Do-loop is a Do-loop of which the value of its index I ranges between 1 and N, and the increment r is a constant by which I is incremented each time it is updated.

Suppose there is a single normalized Do-loop with index (I) values from 1 to N. This loop contains a sequence of m assignment statements $S_1, S_2, ..., S_m$, where $S_1 < S_2 < ... < S_m$. The variables of each statement are of indexed types. The execution mode of typical vector processors [5] for this loop is shown below.

1. Execute statement S_1 for I = 1, I = 2, ... , I = N simultaneously. That is, for each

single vector instruction (e.g., **load, add,** and **store**) of S_1 all instances of its operand(s) (each instance represents one specific index value) are operated concurrently in an overlapped manner. The instructions are executed in pipelined and/or overlapped way.

2. Execute S_2, S_3, ... , S_m in order. For each statement the execution mode is the same as that of S_1.

Based on the execution mode, in general, the vectorizability of statments in a single Do-loop is primarily concerned with the directions of dependence arcs. A Do-loop is vectorizable if the directions of dependence arcs are consistent with that of instruction flow within a statement or the lexical order of statements. For intrastatement dependence, only consistently loop-carried true-dependence inhibits direct vectorization. Consistently loop-carried antidependence or output dependence can be ignored for vectorization [6]. Formally the following theorem is derived.

Theorem 2.1 A single Do-loop without inconsistently dependent statement(s) can be fully vectorized directly if the following two conditions hold.

1. There does not exist a single statement S_i such that S_i [δ] S_i.
2. There does not exist a pair of statements S_i and S_j, where $S_i < S_j$, such that $S_j \phi S_i$, where $\phi \in$ [Δ].

<Proof>

Assume there exists a Do-loop of which the above two conditions hold but it cannot be vectorized. In other words, there exists at least one statement S_p such that the results obtained after vectorization are not consistent with the results obtained

by execution in serial mode. Based on the execution manner of the vector mode, there are two possible causes for S_p.

Case I : There exist x and y, where $1 \leq x < y \leq N$ (upper bound of loop index). S_p uses/defines/redefines a variable at the yth iteration which was defined/used/defined by statement S_q at the xth iteration and $S_p < S_q$. That is, $S_q \, \phi \, S_p$, where $\phi \in [\Delta]$. This is a contradiction to our assumption.

Case II : There exist x and y, where $1 \leq x < y \leq N$ (upper bound of loop index). S_p uses a variable at yth iteration which was defined by itself at xth iteration. That is, $S_p \, [\delta] \, S_p$. This again contradicts our assumption.

Since neither case holds, the assumption is false. Thus, the theorem holds. □

For example, in Figure 2.2(a), $S_1 \, [\bar{\delta}] \, S_1$, which meets the conditions of the theorem, so that the loop can be vectorized immediately. In Figure 2.2(b) where $S_1 \, [\delta] \, S_1$, which violates the conditions of the theorem, the loop cannot be fully vectorized.

```
Do  10  I = 1, 100          Do  10  I = 1, 100
S1      A(I) = A(I+1) + 3   S1      A(I+1) = A(I) + 3
10  Continue                10  Continue
```

(a) A vectorizable loop (b) An unvectorizable loop

Fig. 2.2

A similar example for a loop with two statements is given in Figure 2.3. In Figure 2.3(a), $S_1 \, [\delta] \, S_2$ meets the conditions of the theorem, so the code is vectorizable. However, in Figure 2.3(b) where $S_2 \, [\bar{\delta}] \, S_1$, which violates the conditions of the theorem, the code cannot be vectorized directly.

```
      Do 10 I = 1, 100                    Do 10 I = 1, 100
S1    A(I+1) = B(I)               S1      A(I) = B(I)
S2    C(I) = A(I)                 S2      C(I) = A(I+1)
   10 Continue                         10 Continue
```

(a) A vectorizable loop (b) An unvectorizable loop

Fig. 2.3

Theorem 2.1 introduces a dependence relation for which direct vectorization of the code is available. Theorem 2.2 presents another dependence relation which can be vectorized only after suitable adjustment to the order of the statements.

Definition *Statement Reordering* is an approach which requires the rearrangement of the order of consistently loop-carried dependent statements such that the direction of the dependence arc for the reordered statements is consistent with the lexical order of statements.

Definition For statements S_i and S_j, $S_i \phi^+ S_j$ implies that one or more than one instance of $S_i \phi S_j$ exists, where $\phi \in \Delta \cup [\Delta]$.

Theorem 2.2 A single Do-loop with two statements S_i and S_j, where $S_i < S_j$ and $S_j \phi S_i$, $\phi \in [\Delta]$, is the only dependence relation in the statements, can be vectorized via the statement reordering technique, i.e., reorder S_i and S_j to become $S_j < S_i$.

<Proof>

By definition of consistently loop-carried dependence, such dependence is concerned with the index expressions and relative positions (left-hand side or right-hand side of assignment operator) of the dependent indexed variable(s) in the statements but is not related to the lexical order of statements. After we reorder S_i

and S_j to become $S_j < S_i$, the index expressions and relative positions of the dependent variable(s) are not changed. So the original dependence relation $S_j \phi S_i$, where $\phi \in [\Delta]$, is preserved. By Theorem 2.1, the reordered code is vectorizable. □

An example is shown in Figure 2.4.

```
      Do  10  I = 1, 100                    Do  10  I = 1, 100
S1    B(I) = A(I) + 2                 S2    A(I+1) = C(I+1) - 3
S2    A(I+1) = C(I+1) - 3             S1    B(I) = A(I) + 2
 10   Continue                         10   Continue
```

(a) S_2 [δ] S_1 (b) Reordering to become $S_2 < S_1$

Fig. 2.4

(Corollary) A single Do-loop with n ($n > 1$) assignment statements $S_1, S_2, S_3, \ldots, S_n$ of which $S_1 < S_2 < S_3 \cdots < S_n$ and $S_3 \psi^+ S_2 \phi^+ S_4 \ \phi^+ \ldots \phi^+ S_n \ \psi^+ S_1$, where $\phi \in \Delta \cup [\Delta]$ and $\psi \in [\Delta]$, can be vectorized after we reorder the statements to become $S_3 < S_2 < S_4 \ldots < S_n < S_1$.

Theorem 2.2 suggests no matter how complicated the dependence relations in statements are, the statements can always be vectorized via statement reordering if the dependences do not form a cycle. In practice, most vector compilers use a sorting algorithm to reorder the statements before the generation of vector operations [6,16,36].

If the dependence relations in the statements form a cycle, or more typically, a recurrence, the vectorization of statements is inhibited. Statements involved in a recurrence are strongly connected by various dependences. There exists at least one

path between any pairs of statements in the dependence graph. The vectorizability of a recurrence depends on whether the dependence cycle(s) can be broken or on the level of dependence which can be avoided.

2.3 Resolution of Recurrences

In general, a recurrence is vectorizable as long as one of the existing dependence links is breakable. If we break one dependence link, then by Theorem 2.2, the statements involved in the recurrence can be vectorized via statement reordering. So, to deal with the vectorizability of a recurrence, we only need to consider the breakability of its dependence links.

The breakability of a dependence link is primarily dependent on its dependence types. As most dependences at this stage do not originate from the variables in statements with multiple purpose, our strategy on the link-breaking concentrates on the *repositioning* of the dependence link, but not on the elimination of the dependence relation, such that the resulted dependences do not form a cycle. The breakability of a dependence link implies the possibility of repositioning such a link.

Since true-dependence is value-dependent, and existing techniques cannot reposition this type of link [45], if a recurrence is formed only by true-dependence then such a dependence cycle is unbreakable. This pattern of recurrence can only be partially vectorized via dependence-avoidance strategies. This pattern will be discussed later.

Theoretically, a storage-dependent link is breakable [22]. That is, there must exist some techniques which can reposition such a type of dependence. In order to

find a suitable technique for breaking each type of storage-related dependences in a recurrence, we first need to analyze the link patterns, and then further develop the breaking strategies.

For a statement S_j that depends on a statement S_i, where $S_j \neq S_i$, there exist seven possible dependence links: (1) true-dependence, (2) antidependence, (3) output-dependence, (4) true- and anti-dependences, (5) true- and output-dependences, (6) anti- and output-dependences, and (7) true-, anti-, and output-dependences. The dependence here refers to the general loop-independent or loop-carried dependence, regardless of the direction of the dependence arcs. In terms of breakability and breaking strategy in a recurrence, these seven dependence links can be classified as three patterns. In each pattern the dependence links are shown below.

Pattern I : antidependence link

Pattern II : output-dependence link, and anti- and output-dependences link

Pattern III : true-dependence link, and all other possible dependences link

Of these three patterns, links of Pattern I and II can be broken while those of Pattern III are unbreakable.

In order to address the breaking strategies for the links of Pattern I and II, it is practical to employ a general recurrence of which at least one dependence link is of Pattern I or II. Suppose there exist n (> 1) statements S_0, S_1, ... , S_{n-1}, where $S_0 < S_1 < \ldots < S_{n-1}$, in a Do-loop. These n statements are strongly connected via various dependences. For a pair of statements S_p and S_q, where $0 \leq p, q \leq n-1$, and $p \neq q$, and the dependence link from S_p to S_q is one of the first two link patterns, the breaking technique and application details are described below.

2.3.1 Link Pattern I - Antidependence

For a recurrence with an antidependence link, Kuck, et al. have developed the following node splitting algorithm to break such a dependence [37]:

1. Add an extra assignment statement S_t, where $S_t < S_p$ and $S_t < S_q$.
2. S_t *uses* the indexed variable, that is the source variable of antidependence relation, and assigns its values to a new indexed variable.
3. Substitute the source variable of the antidependence relation with the new indexed variable.

Two examples are shown in Figure 2.5. The directions of antidependence link in these two examples are different.

```
      Do 10 I = 1, 100
S1    B(I) = A(I+2)-1
S2    C(I) = B(I+1)+B(I-1)
      10 Continue
```

(a) A recurrence with an antidependence link

```
      Do 10 I = 1, 100
St    T(I+1) = B(I+1)
S1    B(I) = A(I+2)-1
S2    C(I) = T(I+1)+B(I-1)
      10 Continue
```

(b) Node splitting (no recurrence)

```
      Do 10 I = 1, 100
S1    C(I) = B(I+1)+B(I-1)
S2    B(I) = A(I+2)-1
      10 Continue
```

(c) A recurrence with an antidependence link

```
      Do 10 I = 1, 100
St    T(I+1) = B(I+1)
S1    C(I) = T(I+1)+B(I-1)
S2    B(I) = A(I+2)-1
      10 Continue
```

(d) Node splitting (no recurrence)

Fig. 2.5

In general, the node splitting technique is effective for breaking an antidependence link. We now present theoretical support for this technique.

Theorem 2.3 An antidependence link in a general recurrence is breakable via *node splitting* algorithm.

<Proof>

To prove the link is breakable via node splitting algorithm, we need to show:

(1) All of the dependence relations are preserved via node splitting.

(2) The original dependence cycle does not exist and no new cycle forms.

As S_q is antidependent on S_p, by definitions, there exist some common (indexed) variable(s) in S_p and S_q which are fetched first by S_p and then defined by S_q. This relation is determined by the index expressions and relative position (left-hand side or right-hand side of the assignment operator) of the variables existing in statements, and also related with the lexical order of statements if it is a loop-independent antidependence.

As $S_t < S_p$ and $S_t < S_q$, the precedent fetch of the common variable(s) by S_p is replaced by S_t. In other words, S_q is antidependent on S_t after node splitting. Since S_t assigns all of the values of the indexed variable into new (indexed) variable and the new variable also substitute the original source variable in S_p, S_p is true-dependent on S_t in terms of the new variable. The true dependence promises S_p still prefetches the values of the common variables, indicating the antidependence relation is semantically unchanged. Other dependences are obviously not changed via node splitting, so (1) holds.

The proof of (2) is straightforward. Because there does not exist a dependence link from S_p to S_q or from S_p to S_t, the original dependence cycle does not exist

and no new cycle forms. Both (1) and (2) hold so the theorem holds. □

Definition The *dependence distance* for a particular dependence relation is the number of iterations in a Do-loop over which the data dependence occurs.

If the antidependence link involves more than one instance of antidependence relations, theoretically every instance should be transformed alone via node splitting. In practice, if those instances have constant dependence distances, for the sake of saving storage, we can split only the instance of which the source variable of the antidependence has a *minimum* dependence distance, and modify the source variables of other instances on the basis of that particular source variable. The extra new assignment statement is executed in a separate loop with loop index length $(N+d)$, where N is the original loop index length and d is the difference of dependence distance between the instances with minimum and maximum dependence distances. This technique is shown in Figure 2.6.

```
      Do 10 I = 1, 100
S1    B(I) = A(I+2)-1
S2    C(I) = B(I+1)+B(I+5)+B(I-1)
   10 Continue
```

(a) A recurrence with multiple antidependence links

```
      Do 5 I = 1, 104    (* 100 + 4 *)
    5 T(I+1) = B(I+1)
      Do 10 I = 1, 100
S1    B(I) = A(I+2)-1
S2    C(I) = T(I+1)+T(I+5)+B(I-1)
   10 Continue
```

(b) Node splitting (no recurrence)

Fig. 2.6

The node splitting technique does not universally preserve the original dependences. In the case where the source variable that results in antidependence is itself the sink variable of another *loop-independent* true-dependence, the node splitting transformations are incorrect. This is because that node splitting under this situation

will result in the change of that particular loop-independent true-dependence. Consider the following example:

```
      Do 10 I = 1, 100                    Do 10 I = 1, 100
S1    B(I) = A(I+2)-1              St     T(I+3) = B(I+3)
S2    B(I+3) = B(I-1)-1            S1     B(I) = A(I+2)-1
S3    C(I) = B(I+3)+2              S2     B(I+3) = B(I-1)-1
 10   Continue                     S3     C(I) = T(I+3)+2
                                    10   Continue
```

(a) A recurrence with an antidependence link (b) Node splitting technique is invalid

Fig. 2.7

In this example, S_1 [δ] S_2 δ S_3 [$\bar{\delta}$] S_1 and S_2 [δ^o] S_1. Clearly, node splitting will convert the original true-dependence of S_3 on S_2 to the antidependence of S_2 on S_3. Figure 2.7(b) is an incorrect transformation.

In addition, the node splitting technique cannot eliminate a recurrence through repositioning an essential antidependence link, to which the source variable that results in antidependence is itself the sink variable of another *loop-carried* true-dependence. Since the values of the source variable of antidependence are loop-carriedly true-dependent on other variable, repositioning the antidependence link will always accompany with repositioning the true-dependence link, resulting in the persistence of a recurrence. Consider the example in Figure 2.8. In this example, S_1 [δ] S_2 [δ] S_3 [$\bar{\delta}$] S_1 and S_2 [δ^o] S_1. Node splitting transformation results in S_t [$\bar{\delta}$] S_1 [δ] S_2 [δ] S_t, S_t δ S_3 and S_2 [δ^o] S_1. A new recurrence involving an antidependence link occurs. Further repositioning for the newly generated antidependence link will again lead to a new recurrence with a new antidependence link.

```
      Do 10 I = 1, 100
S1    B(I) = A(I+2)-1
S2    B(I+3) = B(I-1)-1
S3    C(I) = B(I+1)+2
 10   Continue
```

(a) A recurrence with an antidependence link

```
      Do 10 I = 1, 100
St    T(I+1) = B(I+1)
S1    B(I) = A(I+2)-1
S2    B(I+3) = B(I-1)-1
S3    C(I) = T(I+1)+2
 10   Continue
```

(b) Node splitting (a recurrence persists)

Fig. 2.8

Therefore, the node splitting technique is reliable for breaking an antidependence link only when the source variable which results in an antidependence relation is itself *not* the sink variable of another true-dependence. We thus derive the following corollary:

(Corollary) The node splitting technique is valid for breaking an antidependence link in a recurrence of which the source variable which results in the antidependence relation is itself not the sink variable of another true-dependence relation.

To break an antidependence link with such a feature, we need to preserve the true-dependence for the source variable after it is substituted in node splitting. This can be achieved via renaming technique. One additional statement is required. Meanwhile, we simply need to preserve those values of the source variable of antidependence which are free of the true-dependence. Figure 2.9(b) shows the correct transformations of the example in Figure 2.7(a). Note, the antidependence link S_3 [$\bar{\delta}$] S_1 disappears.

```
      Do 10 I = 1, 100
S1    B(I) = A(I+2)-1
S2    B(I+3) = B(I-1)-1
S3    C(I) = B(I+1)+2
 10   Continue
```

(a) A recurrence with an antidependence link

```
      Do 1 I = 1, 2
 1    St   T(I+1) = B(I+1)
      Do 10 I = 1, 100
S1    B(I) = A(I+2)-1
S2    T(I+3) = B(I-1)-1
St'   B(I+3) = T(I+3)
S3    C(I) = T(I+1)+2
 10   Continue
```

(b) Modification after node splitting

Fig. 2.9

Therefore, if the source variable of the antidependence is itself the sink variable of another true-dependence, the node splitting algorithm should be modified as:

1. Add an extra assignment statement S_t, where $S_t < S_p$ and $S_t < S_q$.

2. S_t uses the source variable of the antidependence relation and assigns its values to a new output variable. S_t is executed in an independent loop either in vector mode or in serial mode *before* other statements. The loop index length is the dependence distance of the true dependence.

3. Substitute the source variable of the antidependence relation by the new output variable of S_t.

4. For the true-dependence relation, there exists a statement S_i, where S_i $\delta/[\delta]$ S_p (S_i and S_p are not necessarily different). Change the array name of the output variable of S_i to the same array name of the output variable of S_t. Next, add an extra assignment statement S_t' right after S_i. S_t' carries the renamed output variable of S_i as the input variable and assigns the values to the original output variable of S_i.

5. If there exist other variables which are true-dependent on the original output

variable of S_i, change their array names to the same name as the output variable of S_t. Meanwhile, the loop index length in step 2 should be modified to be the maximum dependence distance among all of these true-dependences.

The node splitting technique repositions the antidependence link via renaming the *source* variable of the antidependence relation. We can also reposition the antidependence arc via renaming the *sink* variable, which also results in the breaking of the dependence cycle. The algorithm for this technique is described as follows, assuming no dead statements exist.

1. Change the array name of the sink variable of the antidependence relation.
2. Change the array name of the variables, if any, which are true-dependent on the sink variable of the antidependence to the same array name as the sink variable.
3. If the dependence distance for the true-dependence relation of step 2 is greater than zero, then use an extra loop with index length that is the dependence distance to preserve those variables which are free of dependence. If more than one instance occurs, only preserve the variable with maximum dependence distance. The loop can be executed in vector mode before other statements.
4. Add an extra assignment statement S_t, where $S_t > S_p$, and $S_t > S_q$.
5. S_t carries the renamed output variable of S_q as the input variable and assigns the values to the original output variable of S_q.

Figure 2.10 shows two examples of such a transformation. The original example is from Figure 2.5.

```
  Do 10 I = 1, 100
S1  B(I) = A(I+2)-1
S2  C(I) = B(I+1)+B(I-1)
 10 Continue
```

(a) A recurrence with an antidependence link

```
  Do 1 I = 1, 1
1 St' T(I-1) = B(I-1)
  Do 10 I = 1, 100
S1  T(I) = A(I+2)-1
S2  C(I) = B(I+1)+T(I-1)
St  B(I) = T(I)
 10 Continue
```

(b) Sink variable renaming

```
  Do 10 I = 1, 100
S1  C(I) = B(I+1)+B(I-1)
S2  B(I) = A(I+2)-1
 10 Continue
```

(c) A recurrence with an antidependence link

```
  Do 1 I = 1, 1
1 St' T(I-1) = B(I-1)
  Do 10 I = 1, 100
S1  C(I) = B(I+1)+T(I-1)
S2  T(I) = A(I+2)-1
St  B(I) = T(I)
 10 Continue
```

(d) Sink variable renaming

Fig. 2.10

We thus can derive the following theorem.

Theorem 2.4 An antidependence link in a general recurrence is breakable via *sink variable renaming* technique.

<Proof>

To prove the link is breakable we need to show:

(1) All of the dependence relations are preserved via renaming.

(2) The original cycle does not exist and no new cycle forms.

If S_q is antidependent on S_p, then by definition, there exists some common (indexed) variable(s) in S_p and S_q which are used first by S_p and then defined by S_q. This relation is determined by the index expressions and relative position (left-hand side or right-hand side of the assignment operator) of the variables

existing in these two statements. It is also related to the lexical order of statements if it is a loop-independent antidependence.

If there does not exist any variables which are true dependent on the output variable of S_q, then since $S_t > S_p$, $S_t > S_q$, and S_t uses the new output variable of S_q to define the original variable of S_q, the order of defining the common variable(s) by S_p and S_q is actually unchanged. Moreover, in terms of the new output variable of S_q, the true dependence relation, between S_q and S_t also ensures that S_q uses the same values to define the common variable(s). So, the renaming technique preserves the original antidependence.

If there exist any variables which are true dependent on the output variable of S_q, then the renaming technique in steps 2 and 3 of the algorithm also preserves such a relationship. In case those variables are themselves also the source variables of other antidependences, renaming semantically does not change the antidependences. Since there do not exist any dead statements, we do not need to consider the case that there is a variable which is loop-independently output-dependent on the output variable of S_q. Other dependences are obviously not changed via renaming. Therefore, (1) holds.

The proof of (2) is straightforward. As there does not exist a dependence arc from S_t to S_q, the original dependence cycle does not exist and no new cycle forms. Both (1) and (2) hold, thus the link is breakable. □

The node splitting transformation codes more neatly and performs more efficiently than the sink variable renaming, if in the recurrence the sink variable of

antidependence is itself the source variable of another *loop-carried* true-dependence. But, in some cases the sink variable renaming is better. For example, if the source variable of antidependence is itself the sink variable of another true-dependence and the sink variable of antidependence is itself the source of another *loop-independent* true-dependence, the sink variable renaming technique is neater and performs more efficiently. Consider this example:

```
Do 10 I = 1, 12
S1  A(I) = B(I)+2
S2  A(I+3) = A(I)-3
S3  C(I) = C(I)+A(I+1)
10  Continue
```

(a) A recurrence

```
Do 10 I = 1, 12
S1  T(I) = B(I)+2
S2  A(I+3) = T(I)-3
S3  C(I) = C(I)+A(I+1)
St  A(I) = T(I)
10  Continue
```

(b) Sink variable renaming

```
Do 10 I = 1, 2
10  St'  T(I+1) = A(I+1)
Do 20 I = 1, 12
S1  A(I) = B(I)+2
S2  T(I+3) = A(I)-3
St  A(I+3) = T(I+3)
S3  C(I) = C(I)+T(I+1)
20  Continue
```

(c) Node splitting

Fig. 2.11

In this case, $S_1\ \delta\ S_2\ [\delta]\ S_3\ [\bar{\delta}]\ S_1$, and $S_2\ [\delta^o]\ S_1$. Thus, the sink variable renaming technique offers a neater, more efficient approach for breaking the antidependence link.

In practice, node splitting transformation and sink variable renaming can be used in a complementary manner to break the antidependence link in a recurrence.

2.3.2 Link Pattern II - Output-Dependence or Anti- and Output-Dependences

This pattern of link in a general recurrence can be broken via a *sink variable renaming* approach. The algorithm for this technique is described as follows, assuming no dead statements exist.

1. Change the array name of the output variable of S_q.

2. Change the array name of the variables, if any, which are true-dependent on the output variable of S_q to the same array name of the output variable of S_q.

3. If the dependence distance for the true-dependence relation is greater than zero, then use an extra loop with length to be the dependence distance to store those variables which are free of dependence. If more than one instance occurs, only preserve the variable with maximum dependence distance. The loop is executed in vector mode or serial mode before other statements.

4. Add an extra assignment statement S_t, where $S_t > S_p$, $S_t > S_q$.

5. S_t carries the renamed output variable of S_q as the input variable and assigns the values into the original output variable of S_q.

Two examples are shown in Figure 2.12.

```
  Do 10 I = 1, 100
S1  B(I+1) = A(I+1)+A(I-1)
S2  B(I) = B(I+2)-1
 10 Continue
```

(a) a recurrence with an output-dependence link

```
  Do 10 I = 1, 100
S1  B(I+1) = A(I+1)+A(I-1)
S2  T(I) = B(I+2)-1
St  B(I) = T(I)
 10 Continue
```

(b) sink variable renaming

```
  Do 10 I = 1, 100
S1  B(I+2) = A(I+2)-1
S2  B(I+3) = B(I)+B(I+4)
 10 Continue
```

(c) A recurrence with an anti- and output-dependences link

```
  Do 1 I = 1, 2
St' T(I) = B(I)
  1 Continue
  Do 10 I = 1, 100
S1  T(I+2) = A(I+2)-1
S2  B(I+3) = T(I)+B(I+4)
St  B(I+2) = T(I+2)
 10 Continue
```

(d) Sink variable renaming

Fig. 2.12

We thus can derive the following theorem.

Theorem 2.5 An output dependence or an anti- and output-dependences link in a general recurrence is breakable via *sink variable renaming* technique.

<Proof>

Similarly, to prove the link is breakable we need to show:

(1) All of the dependence relations are preserved via renaming.

(2) The original cycle does not exist and no new cycle forms.

The output-dependence relation is proved here. For the anti- and output-dependences relation the proof is similar.

If S_q is output-dependent on S_p, then by definitions, there exist some common (indexed) variable(s) in S_p and S_q which are defined first by S_p and then defined by S_q. This relation is determined by the index expressions and relative position (left-hand side or right-hand side of the assignment operator) of the variables existing in these two statements, and is also related to the lexical order of statements if it is a loop-independent output dependence.

In case there does not exist any variables which are true dependent on the output variable of S_q, then since $S_t > S_p$, $S_t > S_q$, and S_t uses the new output variable of S_q to define the original variable of S_q, the order of defining the common variable(s) by S_p and S_q is actually unchanged. Moreover, in terms of new output variable of S_q, the true dependence relation, between S_q and S_t also ensures that S_q uses the same values to define the common variable(s). So, the renaming technique preserves the original output dependence.

If there exist any variables which is true dependent on the output variable of S_q, then renaming technique in algorithm steps 2 and 3 also preserves such a relationship. In case those variables are themselves also the source variables of other

antidependences, renaming semantically preserves the antidependences. Since there do not exist any dead statements, we do not need to consider the case that there is a variable which is loop-independently output-dependent on the output variable of S_q. Other dependences are obviously not changed via renaming, so (1) holds.

To prove (2) is simple, because there does not exist a dependence arc from S_t to S_q, the original dependence cycle does not exist and no new cycle forms. Both (1) and (2) hold so the link is breakable. □

Renaming the sink variable seems to be the only approach to break the output dependence link in a recurrence. This output-dependence link is unbreakable via renaming the source variable, because after repositioning the output-dependence arc, the resulting dependences always form a new dependence cycle.

An anti- and output-dependence link is also breakable via employing the node splitting and sink variable renaming techniques, respectively. This two-step approach is obviously less-efficient compared to the single-step sink variable renaming technique, manifesting the power of sink variable renaming.

Sink variable renaming technique is not limited to break the anti- and output-dependences coexisting in a link. It is also able to reposition the antidependence and output-dependence existing in different links. That is, if antidependence and output-dependence exist respectively in two distinct links and their sink variable of the antidependence and the output-dependence is common, then sink variable renaming can simultaneously break these two links. Figure 2.13 shows an example.

This feature of sink variable renaming can reduce the times to break the depen-

dence cycles in a recurrence. It also can decrease the additional statements required when the links are broken via individual strategies.

```
      Do 10 I = 1, 100
S1    B(I) = A(I+2)-1
S2    B(I+3) = B(I-1)-1
S3    C(I) = B(I+1)+2
10    Continue
```

(a) A recurrence of which S_3 [$\bar{\delta}$] S_1 and S_2 [δ^o] S_1

```
      Do 1 I = 1, 1
1     St   T(I-1) = B(I-1)
      Do 10 I = 1, 100
S1    T(I) = A(I+2)-1
S2    B(I+3) = T(I-1)-1
S3    C(I) = B(I+1)+2
St'   B(I) = T(I)
10    Continue
```

(b) The anti- and output-dependences links are broken via sink variable renaming

Fig. 2.13

Similarly, sink variable renaming also can reposition several distinct antidependence links altogether if all of those antidependences share a common sink variable. Therefore, one advantage of sink variable renaming over the node splitting is that it may simultaneously remove other output-dependence (or antidependence) links, improving the efficiency of recurrence resolution.

2.3.3 Link Pattern III - True-Dependence or Other Dependences

When a recurrence is formed only by link of pattern III, this recurrence can only be executed in partial vector mode. We first discuss the recurrences formed only by true dependence, and then further examine the results obtained for the recurrence involving other types of link pattern III.

Although the dependence graph for this recurrence indicates that all statements are strongly connected, there may be instances of each statement that are not involved in a dependence. We thus can create a two-layer nested loop and put these

dependence-free instances of the statements into the inner loop. The inner loop then can be executed either in parallel or in vector mode [6,45]. Figure 2.14 shows two examples.

Allen et al. [6] state that an indirect dependence does not exist whenever one of the direct dependences in the dependence path does not exist. The dependence distance for a particular dependence path is thus the *maximum* of the direct dependences in the path.

```
Do 10 I = 1, 100
  B(I+10) = B(I)-1
10 Continue
```

(a) A recurrence formed by true-dependence

```
Do 10 I = 1, 10      { scalar }
  Do 10 J = 1, 10    { vector }
    B(I*10+J) = B(I*10+J-10)-1
10 Continue
```

(b) Partial vectorization

```
Do 10 I = 1, 100
  B(I+5) = A(I)-1
  A(I+5) = B(I)/2
10 Continue
```

(c) A recurrence formed by true-dependences

```
Do 10 I = 1, 20      { scalar }
  Do 10 J = 1, 5     { vector }
    B(I*5+J) = A(I*5+J-5)-1
    A(I*5+J) = B(I*5+J-5)-1
10 Continue
```

(d) Partial vectorization

Fig. 2.14

Since there may exist multiple paths between two statements and the dependence does not occur only when all the indirect dependences do not exist, then the dependence distance of a composite dependence is the *minimum* of the dependence of individual paths.

Recurrences in a Do-loop are a type of dependence relations in which each statement is indirectly dependent on itself. Therefore, there must exists at least a dependence path between any statement and itself in a recurrence. In general, there may

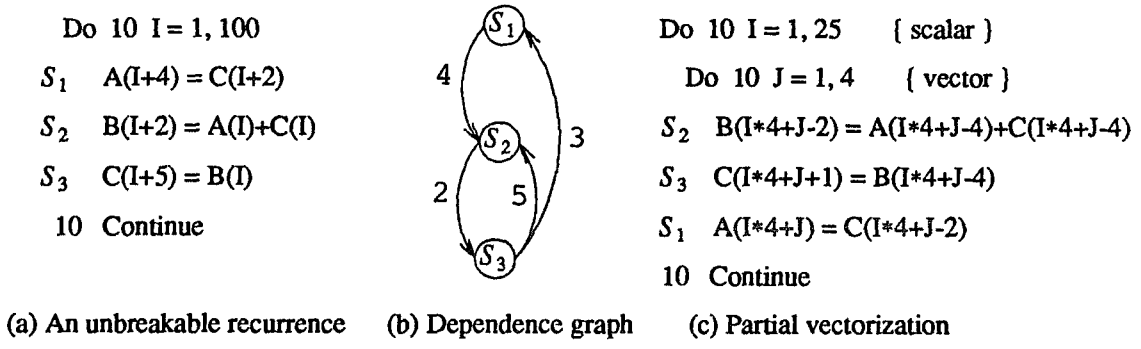
exist multiple paths between a statement and itself. Since the dependence in a recurrence will disappear only when all the statements do not depend on themselves indirectly, the dependence distance for a recurrence is the *minimum* of the composite dependence distance of the individual statements. On the other hand, in recurrences, by Theorem 2.1, there cannot exist a pair of statements S_i and S_j, where $S_i < S_j$, S_j [δ] S_i, and the dependence distance for S_i on S_j is less than that of the recurrences. Otherwise, the dependence of S_i on S_j makes the code unable to be partially vectorized. That is, the statements may need to be reordered to avoid such a situation. Based on these concepts, an algorithm to partially vectorize a general recurrence formed by true dependences follows:

1. Find the composite dependence distances for each statement on itself in the recurrence.
2. Find the dependence distance of the recurrence, i.e., the minimum composite dependence among the statements in the recurrence. Assume it is λ_p.
3. Reorder the statements (preserving the original dependences) properly such that in the reordered code there does not exist a pair of statements S_i and S_j, where $S_i < S_j$, S_j [δ] S_i, and the dependence distance for S_i on S_j is less than λ_p. If this reordering is impossible, then simply reorder the statements to a particular sequence such that the minimum of the dependence distance among every pair of S_i and S_j, where $S_i < S_j$ and S_j [δ] S_i, is the greatest. Let it be λ_p.
4. Put the reordered statements into an inner loop with loop index length λ_p.
5. Set the index length of the outer loop to be $\lceil N/\lambda_p \rceil$ (N is upper bound of original loop).

6. Modify the index expression for each indexed variable in statements to be equal to the original index expression.

7. Execute the inner loop in vector mode, and the outer loop in serial mode.

Figure 2.15 presents an example that uses this algorithm.



(a) An unbreakable recurrence (b) Dependence graph (c) Partial vectorization

Fig. 2.15

For this example, S_1 [δ] S_2 [δ] S_3 [δ] S_1 and S_3 [δ] S_2. Figure 2.15(b) shows the dependence graph. The dependence arcs are labeled with direct dependence distances for each individual dependence. Obviously, there exist two dependence paths between S_2 and itself, and between S_3 and itself. The composite dependence distances for S_1, S_2, and S_3 are 4, 5, 5, respectively. Therefore, the dependence distance for the recurrence is 4. Meanwhile, the statement order must be adjusted to avoid the existence of dependence for statement S_1 at this dependence distance. The resulted code is shown in Figure 2.15(c).

This algorithm is not limited to dealing with only the recurrence formed by true-dependence. It can be applied to a more general recurrence formed by any patterns of links.

As we know, the larger the dependence distance λ_p, the greater the speedup. This

algorithm indeed provides a general approach to exploit the parallelism in the recurrence. But, what Allen et al. stated is in fact a concept of *static dependence distance*. Partial vectorization based on static dependence distance is somewhat conservative in terms of parallelism exploitation in a recurrence. To more fully expose the parallelism we need to further incorporate the concept of *dynamic dependence distance*. We first introduce the definition more formally.

Definition *Static dependence distance* in statements refers to the dependence distance found at compile-time; *dynamic dependence distance* in statements refers to the dependence distance found at run-time.

Let us use a n-statement simple recurrence, i.e. a recurrence which does not contain any sub-recurrences, to state the concept of dynamic dependence distance. Suppose we have n (> 1) statements $S_0, S_1, \ldots, S_{n-1}$ in a Do-loop, $S_0 < S_1 < \ldots < S_{n-1}$ and S_0 $\phi^+ S_1 \phi^+ \ldots \phi^+ S_{n-1} \phi^+ S_0$, where $\phi \in \Delta \cup [\Delta]$. The static dependence distance for $S_{(i+1) \bmod n}$ on S_i is λ_i for all i, where $0 \le i \le n - 1$. Since the static dependence distance in the Do-loop for S_0 on S_{n-1} is λ_{n-1}, S_0 is free of dependence for λ_{n-1} iterations. So, S_0 can be executed in vector mode for λ_{n-1} iterations at run-time. After S_0 has been executed for λ_{n-1} iterations, the dynamic dependence distance for S_1 on S_0 will be $\lambda_{n-1} + \lambda_0$, where λ_0 is the static dependence distance for S_1 on S_0. That is, S_1 is free of dependence for $\lambda_{n-1} + \lambda_0$ iterations at run-time instead of λ_0 iterations at compile-time. Therefore, in this manner at the first cycle of execution for the recurrence the dynamic dependence distance for S_i will be $\sum_{k=0}^{i-1} \lambda_k + \lambda_{n-1}$, where $0 \le i \le n$ -1. Since the dynamic indirect dependence distance for each statement on itself in the recurrence is $\sum_{k=0}^{n-1} \lambda_k$,

after the first cycle of execution, every statement can be executed in vector mode for $\sum_{k=0}^{n-1} \lambda_k$ iterations sequentially until the upper bound of the loop. These observations are summarized in the following definition.

Definition In a simple recurrence, the dynamic indirect dependence distance for each statement depending on itself is the sum of all static direct dependence distances between each successive pair of statements in the path. In the first execution, the dynamic direct dependence distance for each statement is the sum of all static direct dependence distances between each successive pair of statements lexically in front of the particular statement.

Based on the description, in order to partially vectorize the statements in a simple recurrence via dynamic dependence strategy, the statements must be aligned in such a sequence that except for the first statement, each statement only depends on the statement lexically in front of it.

This dynamic dependence concept provides an approach to improve the efficiency of partial vectorization for an unbreakable recurrence. It is suitably used to deal with each individual recurrence pattern, but it is not straightforward to generalize the technique to a general recurrence. However, some basic principles are still available:

1. Like static dependence, the composite dynamic dependence distance between a statement and itself in a recurrence is the minimum one of individual paths, provided that more than one paths exist between this statement and itself. Thus, the dynamic dependence distance of a general recurrence is the minimum of the

composite dynamic dependence distance of individual statements.

2. The statements in the recurrence must be reordered properly to allow the implementation of dynamic dependence technique.

3. In case reordering the statements for implementing dynamic dependence technique is unavailable, search the second-largest possible dynamic dependence distance for the recurrence. That is, the original dynamic dependence distance needs to be modified properly to meet the constraint of dependence of other statements. Generally, the worst-case for the dynamic dependence distance is the static dependence distance.

The following algorithm is based on dynamic dependence concept to partially vectorize a simple recurrence. It is a variation of the cycle shrinking technique [14].

1. Reorder the statements to a sequence such that there exists only one pair of statements (the first and the last one) S_i and S_j, where $S_i < S_j$ and S_j [δ] S_i.

2. Align the statements in a single loop with index length to be $\sum_{k=0}^{n-1} \lambda_k$.

3. Adjust execution times for each statement based on its dynamic indirect dependence distance. Use control statements to do this.

4. Align the statements in an inner loop with loop index length to be $\sum_{k=0}^{n-1} \lambda_k$.

5. Set the index length of the outer loop to be $\lceil N / \sum_{k=0}^{n-1} \lambda_k \rceil - 1$. (N is upper bound of original loop).

6. Modify the index expression for each indexed variable in statements to get the same values as the original index expression.

7. Align the statements except the last one in a single loop with loop index

length to be $\sum_{k=0}^{n-1} \lambda_k - \lambda_{n-1}$.

8. Adjust iteration times for each statement based on its remaining iterations. Using control statements to do this.

9. Execute the first single loop in vector mode. Then, for the nested loop, executes the inner loop in vector mode, and the outer loop in serial mode. Finally, execute the second single loop in vector mode.

An example that uses this algorithm is presented in Figure 2.16.

```
Do 10 I = 1, 100
  B(I+5) = C(I)-1
  A(I+2) = B(I)/2
  C(I+3) = A(I)+Y
10 Continue
```

(a) An unbreakable recurrence

```
Do 1 I = 1, 10      { vector }
   if (I < 4) B(I+5) = C(I)-1
   if (I < 9) A(I+2) = B(I)/2
   C(I+3) = A(I)+Y
1  Continue
Do 10 I = 1, 9      { scalar }
   Do 10 J = 1, 10  { vector }
      B(I*10+J-2) = C(I*10+J-7)-1
      A(I*10+J) = B(I*10+J-2)/2
      C(I*10+J+3) = A(I*10+J)+Y
10 Continue
Do 5 I = 1, 7       { vector }
   B(I+98) = C(I+93)-1
   if (I < 3) A(I+100) = B(I+98)/2
5  Continue
```

(b) Partial vectorization via dynamic dependence

Fig. 2.16

Compared with the former algorithm, this approach may induce some difficulty on the modification of code. But in terms of speedup, this technique is significantly superor. In addition, the statement arrangement and index expression modification strategies can be further studied to find an optimization approach.

As stated at the beginning, we assumed that the dependence type for this recurrence is a true-dependence. We need to generalize the resolving strategy to include other types of link pattern III.

As we know, a direct dependence with multiple paths ceases to exist whenever all of the direct dependences in each individual paths cease to exist. Hence, the dependence distance for a particular direct dependence with multiple paths is simply the *minimum* of the dependence existing in the paths. Therefore, the dependence distance for a link with true-dependence and other types of dependence is the minimum of each individual dependence distance. This result can be included in the algorithms of partial vectorization. In addition, we also can use the breaking strategies to break antidependence and/or output-dependence relations first before we apply the partial vectorization strategy to deal such a recurrence; however, the cost needs to be weighted against the direct strategy for partial vectorization.

2.4 An Algorithm for Resolution of A General Recurrence

All of the above results are aimed at developing an efficient algorithm to deal with a general recurrence. In the algorithm presented below, the term π-block is used. A π-block is a node of the dependence graph derived by using loop distribution algorithm [37] to general dependence graphs. The node is either a single statement or a set of statement(s) cyclically connected by data dependence.

Suppose there exist a general recurrence consisting of n statements. A recursive algorithm used to resolve it follows.

1. Starting from the first statement, examine one link connecting it. If it is a

pattern-I link, follow the strategy mentioned in Section 3.1. If it is a pattern-II link, use the sink variable renaming technique to break it. For pattern-III link, ignore it temporarily and examine another link connecting it. If there does not exist any breakable link examine the succeeding statement and take same actions. If again no breakable link is found, do the same thing for the succeeding link. Once a link is broken go to step 2. If examine the last link of the last statement and still do not find a breakable link, go to step 6.

2. After breaking a link, a transformed code with new relation is generated. Use a global dependence testing algorithm to determine the new dependence relations.

3. Use Tarjan's depth-first search technique [52] to find π-blocks.

4. Use a topological sorting algorithm [34] to reorder the sequences of π-blocks.

5. Examine the reordered π-blocks in order and do the following steps.

(1) If it is a vectorizable π-block, output the code and go back to step 5.

(2) If it is an unvectorizable π-block, go to step 1, then go back to step 5.

(3) If no other π-blocks are left, go to step 7.

6. Use the partial vectorization algorithm to transform the code. For a simple recurrence, use a variation of cycle shrinking technique. For other patterns of recurrence, use the general partial vectorization strategy. After code transformation, output the code.

7. Stop.

This algorithm may need to involve a code optimization phase and a performance evaluation phase at the end stage. The former is used to remove redundant code in the transformed code. For example, at the time of breaking antidependences

(or output dependences), extra independent loops may be generated. Those loops may be able to be combined to make the resulted code cleaner and more efficient. The performance evaluation phase is used to evaluate the performance of the transformed code in vector mode, and to decide whether the original code merits vectorization.

2.5 Summary of Results

To speedup the execution of Do-loops in vector processing, a systematic approach to exploit the parallelism in recurrence relations is needed. The breakability of a recurrence is dependent on the dependence links (relations) of the recurrence. For a statement S_j that depends on a statement S_i, where $S_i \neq S_j$, there exist seven possible dependence links. In terms of the breakability and the breaking strategy in a recurrence, these seven dependence links can be classified as three patterns (sets): (I) antidependence link, (II) output-dependence link, and anti- and output-dependence link, and (III) true-dependence link and all other possible dependence links. A recurrence with dependence link(s) of pattern I or II is breakable.

To break an antidependence link, the node splitting technique is available. But, in case the source variable for the antidependence link is itself the sink variable of another true-dependence link, the node splitting technique needs to be modified. In this chapter, we have presented the theoretical foundation, application range, and correction method for node splitting technique. An output-dependence link or an anti- and output- dependence link is breakable via sink variable renaming technique. This technique is also available to break an antidependence link. The algorithm, theoretical evidence, and implementation details for the sink variable renaming have also been

stated.

In practice, node splitting and sink variable renaming should be employed in a complementary manner to deal with the breaking of antidependence links. A recurrence formed by pattern III links can only be partially vectorized. Based on the static dependence concept, a general, simple, but less-efficient algorithm is presented. To improve the efficiency, a modified, obviously powerful algorithm, which is based on dynamic dependence concept, is introduced. Currently this modified algorithm is restricted to deal with a simple recurrence.

The developed dependence-link breaking strategies and partial vectorization algorithms can be integrated with the existing dependence testing techniques, depth-first search algorithm, and topological sorting. A general algorithm to exploit the parallelism in recurrence relations in repetitive language constructs such as Do-loop for vector processing has been proposed.

Chapter 3

Dependence Cycle Statement Ordering on the Performance of Parallel Loops

3.1 Introduction

In addition to vector processing, multiprocessing offers a promising and powerful alternative approach for improving computing performance. A multiprocessor system can be used either for multiprogramming to increase the throughput or for large-job parallel processing to reduce the turnaround time. In the latter case, multiple processors, dedicated to a single job, execute in parallel.

To speed up the execution of a single-job program by parallel processing, the program should be partitioned into as many parallel tasks as possible. These parallel tasks do not necessarily have to be data independent if the data dependences between them are enforced by interprocessor communication and synchronization. But the fewer data dependences that exist in them, the better the processors can be utilized [51]. Thus, one approach for improving the turnaround time of a program is to fully exploit parallelism at the algorithm and code levels.

The parallel tasks of a program are assigned to distinct processors for parallel processing. Due to several factors, including interprocessor communication and synchronization, the speedup of a multiprocessed single-job program may not be as great as expected. For a given number of processors, the scheme of scheduling these processors to different tasks is highly related to the time delay of interprocessor communication and/or synchronization. The scheduling policy clearly affects the performance of

a program.

In scientific or engineering programs, most computations are concentrated in the Do-loop structure. To improve the performance of such a program in parallel processing, speeding up the execution of Do-loops is required. Research on parallel Do-loops focuses primarily on: (1) the processor scheduling for the iterations of Do-loops, and (2) the exploitation of statement-level parallelism in Do-loops. These two areas are not independent, however. An efficient scheduling scheme will result in the increment of parallelism, and vice versa.

The statement-level parallelism, i.e., the number of overlapped statements during concurrent execution, depends on the data dependence. In parallel processing, three classes of dependence relationships in statements of a Do-loop are related to the performance: (1) lexical forward dependence, (2) lexical loop-carried backward dependence (denoted LBD), and (3) dependence cycle. A lexical forward dependence is a dependency from a statement S_p to a statement S_q that occurs lexically later in the loop, whereas a lexical loop-carried backward dependence is a dependency to an earlier statement [56]. A dependence cycle implies that both a lexical forward dependence and a lexical backward dependence coexist in the statements, forming a recurrence of dependence. Due to the lexical order of the interdependent statements, synchronization-wait inherently occurs in the processors executing the statements with lexical backward dependence, significantly inhibiting the parallelism level. But, the lexical loop-carried backward dependence may be transformed to lexical loop-carried forward dependence, which, in the best case, will not reduce the parallelism in the loop except for the overhead of synchronizing [28].

To exploit the parallelism that exists in dependence cycles of Do-loops, research has been conducted in terms of program partitioning, cycle shrinking and statement reordering [21,43,45]. Program partitioning is a method to partition a loop into many independent tasks such that no synchronization is required for concurrent execution. Techniques such as GCD (greatest common divisor) and minimum distance of the dependences have been proposed [42,43]. The cycle shrinking strategy deals with the extraction of dependence-free portions in the dependence cycle, which are partially parallelized via barrier synchronization mode. Statement reordering refers to the alignment of statements in dependence cycles to enhance the parallelism.

In this chapter we study the exploitation of statement-level parallelism of parallel Do-loops executed in a *random* (i.e., general) synchronization scheme [56], with emphasis on the strategy of statement ordering in dependence cycles. In Section 3.2, we address some related concepts regarding the parallel processing of Do-loops. Section 3.3 provides an analysis of statement ordering in dependence cycles in relation to the parallelism level of a single loop. An effective and general algorithm for statement reordering, together with its theoretical evidence are proposed. In Section 3.4 we discuss the impact of statement ordering in global dependence cycles on the performance of multiple perfectly nested loops. The joint effects of statement ordering with the dynamic shortest-delay self-scheduling scheme are also discussed. Section 3.5 contains a summary.

3.2 Some Related Concepts

3.2.1 Model of the Multiprocessor

The machine model used is a *shared memory multiprocessor* system. It consists of a fixed number of identical processors connected through the interconnection network to a common set of memory modules, which are equally accessible by the processors. Each processor has its own local memory which can be used for local storage, and each processor can be a scalar processor or a vector processor. The interprocessor synchronization or communication is implemented via special memory locations or special storage. Commercial systems such as the Cray X-MP and IBM 3090 fit this model [54].

3.2.2 Parallel Processing of Do-Loops

A Do-loop in a program can be declared as a parallel loop for a multiprocessor system. Depending on whether there are loop-carried data dependences, there can be two types of parallel loops: Doall loop and Doacross loop [6]. The Doall loop is a loop of which no dependence or only loop-independent dependence exists in the statement(s). If the number of processors is unlimited, a Doall loop can be executed in asynchronous parallel mode without interprocessor communication and synchronization. The Doacross loop refers to a loop with loop-carried data dependence in statement(s), such that the processors always need to be synchronized or communicate with each other to ensure the correct semantics. For a parallel Do-loop, the tasks (iterations) can be allocated to the processors on the basis of a single iteration or a chunk of iterations, statically or dynamically, depending on the scheduling schemes.

The Doacross loops can be executed in a variety of synchronization schemes [56]. In barrier synchronization strategy, the loop body is divided into segment(s) and all executing processors must complete execution of a segment before any processor starts to execute the next segment. The random synchronization scheme is a simple and flexible strategy of which synchronization primitives are placed for every data dependence relation. This scheme can expose the potential parallelism of the loop, but may result in difficulty of management of synchronization points and the overhead of synchronization instructions. Existing techniques can eliminate redundant synchronizations [38,41].

3.3 Dependence Cycle Statement Ordering on the Performance of Single Loops

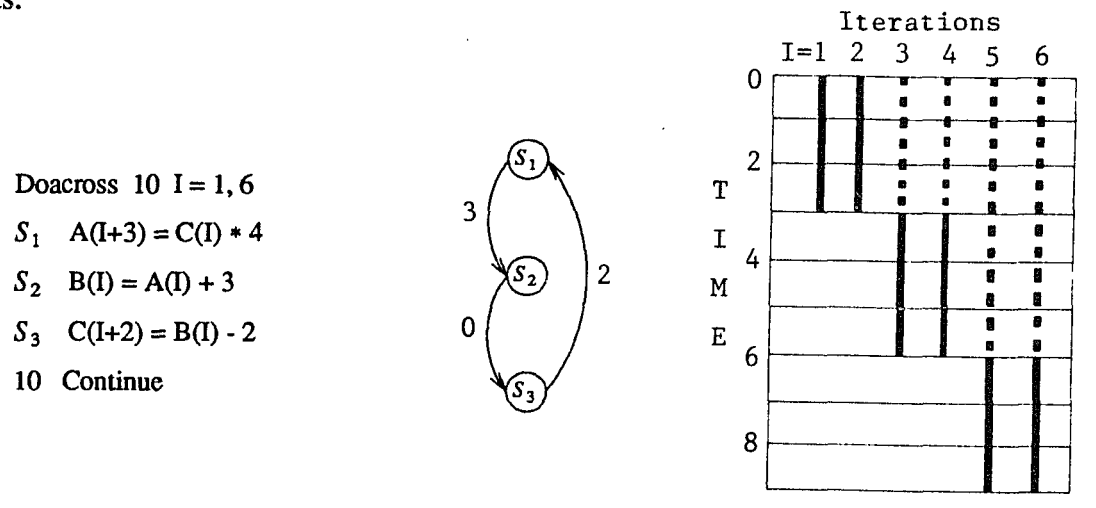
Formally, a dependence cycle is a dependence relation that exists for statements that are strongly connected via various dependence arcs (relations). A dependence cycle is breakable if one of the essential links is antidependent or output-dependent [16]. For a breakable cycle, the links of antidependence or output-dependence can be repositioned such that the cycle disappears. The resulting code can be transformed to remove lexical backward dependences. Due to the avoidance of parallelism-inhibiting dependences, the performance of the code may be improved.

In case all the links of a dependence cycle are true-dependence, the dependence cycle is unbreakable with any existing techniques. As a result, the lexical backward dependences in the cycle inevitably inhibits the parallelism exposure of the parallel loop. Since the level of inhibition varies with the backward dependences, and reordering the statements (without changing the semantics) in the cycle forms different back-

ward dependences, we may reorder the statements to improve the level of parallelism.

3.3.1 An Example of Statement Ordering on Performance

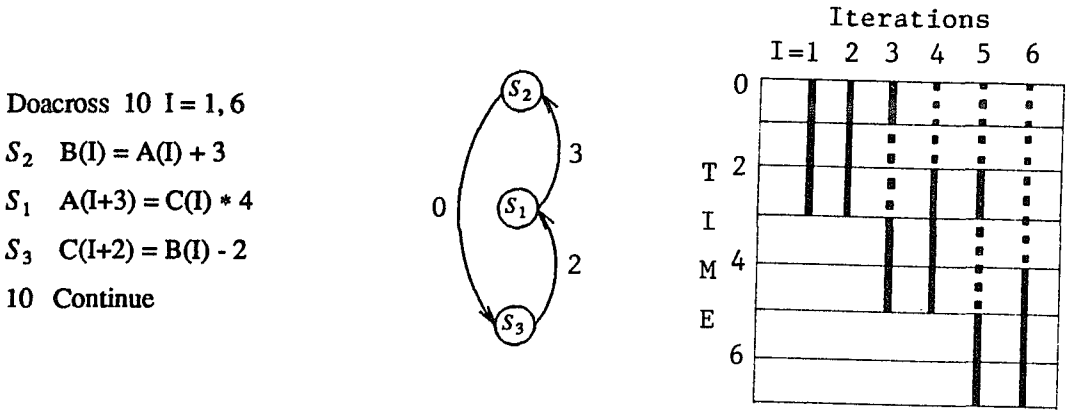
Figure 3.1(a) shows a Doacross loop with a dependence cycle in the statements. The data dependence graph for the statements is shown in Figure 3.1(b). Each circle represents a statement and each directed arc denotes a data dependence. The arcs are labeled with the corresponding dependence distances. Suppose the number of processors in the system is unlimited and each iteration of the loop is assigned to a distinct processor, the execution profile of the loop is shown in Figure 3.1(c), assuming each statement takes one unit of time and the time costs of synchronization instructions and communication are ignored. The solid lines denote the execution time, while the dashed lines denote the idle time of the processors waiting for the data synchronization. Due to the synchronization-wait caused by S_3 [δ] S_1 with dependence distance 2, the level of parallelism of the loop is reduced. The total execution time is 9 time units.



(a) The loop (b) The dependence graph (c) The execution profile

Fig. 3.1 A Doacross loop with a dependence cycle

If we reorder the statements of the dependence cycle in Figure 3.1 without modifying the semantics, the level of parallelism exposed and the total execution time required are different. Figure 3.2 shows another legal (dependences preserved) alignment of the statements. Notice that two lexical backward dependences, S_1 [δ] S_2 and S_3 [δ] S_1, exist in the cycle. In Figure 3.2(c) the number of overlapped statements increases and the total execution time is reduced to 7 time units.



(a) The loop (b) The dependence graph (c) The execution profile

Fig. 3.2 A Doacross loop with a dependence cycle

Figures 3.1 and 3.2 demonstrate that for a Doacross loop with a dependence cycle, the level of parallelism varies with the alignment of statements, which reflects on the execution time. The effect of statement ordering on the parallelism level is described in detail in the following sections.

3.3.2 Estimating the Execution Time of A Doacross Loop

As the level of parallelism in a parallel loop affects the execution time, we thus can use the execution time of a parallel loop to derive the level of parallelism. Let m denote the number of statements, t denote the execution time of each statement (assume same execution time for each statement), and N denote the number of

iterations. The level of parallelism P of a parallel loop with m statements, N iterations, t time units for each statement and S units of execution time is

$$P = \left\lceil 1 - \frac{S - mt}{Nmt - mt} \right\rceil \tag{3.1}$$

If S = mt the parallelism is 100%. If S = Nmt the parallelism is 0%. For example, the loop in Figure 3.1 has t = 1, m = 3, N = 6, and S = 9. The level of parallelism is 1 - 6/15 = 60%. After reordering the statements in Figure 3.2, S = 7, the level of parallelism becomes 1 - 4/15 = 73.3%.

To study an algorithm of statement reordering in a dependence cycle for maximum parallelism, we first need to develop a general formula to estimate the execution time of a Doacross loop. For a Doacross loop there are three parameters dominating the level of its parallelism exposure (in other words, the execution time): (1) the lexical distances of the lexical backward dependences, (2) the dependence distances of the lexical backward dependences, and (3) the number of iterations. The lexical distance of a lexical backward dependence is the number of statements between the source and the sink statements, inclusively, of the dependence. It determines the time that the processors executing the sink statements need to wait for data synchronization. The dependence distances and the number of iterations are related to the occurrence times of the backward dependences, indirectly deciding the execution time. Since the synchronization-wait for backward dependences occurs across whole iterations, there exists a linear recurrence relation on the waiting time for the processors. But, in general, the recurrence relations are not homogeneous across the iterations. Thus, deriving a single formula to precisely estimate the execution time of a general Doacross

loop is difficult.

Cytron [21] states that for a single Doacross loop with dependence cycles, in addition to the first iteration, each iteration delays an average of x time units more than its preceding iteration. Since all delays arising from each distinct lexical backward dependence can be satisfied by the maximum delay, x is the *maximum* delay per iteration among all existing lexical backward dependences. The average delay per iteration for each single lexical backward dependence S_p [δ] S_q is $t(S_p,S_q)/d_{pq}$, where $t(S_p,S_q)$ is the sequential execution time between S_p and S_q, inclusively, and d_{pq} is the dependence distance. Therefore, in a single loop with N iterations, the last iteration will accumulate $(N-1)x$ delay before executing the loop body. Based on this concept, he develops a near-optimal formula to estimate the execution time of a general Doacross loop with dependence cycles:

$$T(loop_l) = \sum_{i=1}^{l} ((N_i - 1)\, x_i + T(B_i)) \tag{3.2}$$

where $T(loop_l)$: the execution time of the loop at nest level l

N_i : upper bound of loop index at nest level i

x_i : the maximum delay at nest level i

$T(B_i)$: the sequential execution time of loop body at nest level i

In general, (3.2) offers a simple, accurate and efficient approach to estimate the execution time of a Doacross loop. Finding the optimum alignment of statements with minimum delay on the basis of (3.2) belongs to the domain of linear programming [21]. But, (3.2) reveals some guiding principles for statement ordering:

1. All statements which are not strongly connected in the dependence cycle need

to be moved away to reduce the sequential execution time between the lexical backward-dependent statements. Current parallel compilers are capable of optimizing the loop for this objective [56].

2. Since the shortest execution time comes from the alignment with minimum delay, we need first to check all possible average delays per iteration in the alignments from the existing dependence distances and the execution time of each statement. Then, starting from the minimum delay, validate in order whether there exists a corresponding alignment with the particular tested delay. The first valid alignment found is the one with minimum execution time.

Note, $t(S_p,S_q)$ for (3.2) equals to the lexical distance of S_p [δ] S_q, if each statement takes one time unit. In the remainder of this chapter, we assume that the statements in the dependence cycles are pretransformed into a three-address code such that each statement takes roughly the same amount of execution time. Unless specified, we use the lexical distance of a loop-carried dependence, S_p [δ] S_q, to represent the sequential execution time between the statements S_p and S_q, inclusively.

Formula (3.2) is established on the assumption that each iteration suffers the same delay as its preceding iteration. This assumption is reasonable and required for deriving a generalized timing formula, but it may not fit perfectly to some cases.

Consider a single loop with a lexical backward dependence LBD_i and loop upper bound N. Let the lexical distance and dependence distance of LBD_i be l and d. By definition, starting from the first iteration, synchronization-wait for l time units occurs in every d iterations subsequently across N iterations. That is, the $(nd+1)$th iteration will delay for nl time units, where $n \geq 1$. In between $(nd+1)$th and $(nd+d)$th iteration,

inclusively, no data synchronization is required, resulting in *identical* execution time for these d iterations. However, (3.2) assumes each iteration takes an average of l/d delay, which contributes an (positive) error of 0 to $(l/d)(d-1)$ units execution time to each iteration. The error of execution time for N iterations thus is

$$\frac{l}{d}((N-1) \bmod d) \tag{3.3}$$

time units. This fact implies that for some loops (3.2) is inaccurate, and the magnitude of error depends on the maximum delay per iteration and the loop upper bound.

Formula (3.2) also implies that the execution time for two distinct alignments of a single loop is equivalent if their maximum delay is the same. Suppose there exists a single loop with index N. The maximum delays for two alignments, $alignment_1$ and $alignment_2$, are l_1/d_1 and l_2/d_2. If $l_1/d_1 = l_2/d_2 = k$ and $N = nd_1 + d_1 = md_2 + 1$, where $n, m \geq 0$, by (3.3), the error of execution time for $alignment_1$ is $(d_1 - 1)k$ but zero for $alignment_2$. Thus the difference of execution time between these two alignments is $(d_1 - 1)k$ rather than zero. In general, the possible difference of execution time, x, for any two alignments among a set of alignments with equal maximum delay (i.e., $l_i/d_i = k$ for all $alignment_i$) is

$$0 \leq x \leq k(max\{d_i, \text{for } all\ i\} - 1) \tag{3.4}$$

Another implication from (3.2), i.e., an alignment of statements with *lesser* delay will result in less execution time, is true simply in terms of a general case. Consider two alignments of statements, $alignment_1$ and $alignment_2$, of a single loop with index value N. Suppose the maximum delays for these two alignments are l_1/d_1 and l_2/d_2, respectively, where $l_1/d_1 < l_2/d_2$. By (3.2) the execution time for $alignment_1$ is less than

alignment$_2$. However, in case there is an error of l_2/d_2 $((N-1)\ mod\ d_2)$ execution time for *alignment*$_2$, then it requires

$$\frac{l_2/d_2\,((N-1)\ mod\ d_2)}{l_2/d_2 - l_1/d_1} \tag{3.5}$$

iterations for *alignment*$_1$ to make up these errors, because the delay in one iteration for *alignment*$_1$ is less than *alignment*$_2$ by $l_2/d_2 - l_1/d_1$ time units. Hence, only when N is greater than the value of (3.5) will *alignment*$_1$ result in less execution time than *alignment*$_2$. This result suggests that in addition to the maximum delay, other parameters such as the error of execution time and loop upper bound also need to be considered when deriving a statement order on the basis of (3.2).

The limitation in (3.2) results in at least two disadvantages:

1. Although for a single loop the inaccuracy shown in (3.3) may be negligible, the errors summed in multiple nested loops can be considerable. By (3.2) and (3.3) the accumulated error for a p-layer *perfectly* nested loop (i.e., a multiple nested loop of which the statements in the loop are identical in the scope of each surrounding loop) is $\sum_{i=1}^{p} \frac{l_i}{d_i}\,((N_i - 1)\ mod\ d_i)$. Similarly, for a p-layer *perfectly* nested loop, the accumulated difference for the alignments with equal maximum delay k_i at each surrounding loop i is $\sum_{i=1}^{p} k_i\,(max\,\{d_i\} - m_i)$, where $1 \le m_i \le \max\{d_i\}$. These differences are further aggravated if the loop is executed repeatedly. For a p-layer nested loop executed n times in a program, the maximum difference of execution time can be $np(l/d)(d-1) \approx npl$, assuming equal delay existing in each layer of the loop. Such a difference can degenerate the performance of the program significantly.

2. As the optimum quantity of processors needed for a parallel loop varies with statements alignments, inadequate ordering of statements based on (3.2) may cause unsuitable allocation of processors. Such an ordering is clearly detrimental to the performance of the loop.

To revise (3.2), we first introduce a timing formula for a single loop of which a single lexical backward dependence exists in its body. Each statement is assumed to take one time unit.

Lemma 3.1 The execution time of a single loop with a single LBD in its body is

$$T(loop) = m + (\lceil N/d \rceil - 1)\, l \tag{3.6}$$

where m, N, d and l denote the number of statements, loop upper bound, dependence distance and lexical distance of the LBD, respectively.

<Proof>

In addition to the first d iterations, the remaining $N - d$ iterations are data-dependent on the earlier iterations. As the dependence distance is d, there exist $\lceil (N-d)/d \rceil$ successive data-dependences across N iterations. Each dependence delays for l time units. The accumulated delay time (denoted ADT) will be $\lceil (N-d)/d \rceil l$. The execution time for m statements is m. So the total execution time of the loop is as shown in (3.6). □

For example, for the loop in Figure 3.1(a), m = 3, N = 6, d = 2, and l = 3, the execution time is 3 + (6/2 – 1) * 3 = 9, which is same as that obtained in Figure 3.1(c).

For a single loop with more than one LBDs, it is observed that in some cases the delay is determined primarily by a particular LBD. Formally, we introduce the

following definition.

Definition For a single loop with k ($\geq$ 1) LBDs, the execution time of the loop is *dominated* by a particular LBD_p only if the delays of all other existing LBDs are satisfied by the delay of LBD_p.

Consider a loop consisting of k (> 1) lexical backward dependences, i.e., LBD_1, LBD_2, ... , LBD_k. For each LBD_i, where $1 \leq i \leq k$, d_i and l_i represent the dependence distance and lexical distance. Some observations regarding the identification of the dominated LBD in the loop are shown in the following lemmas.

Lemma 3.2 If $d_1 = d_2 = ... = d_k$ and $l_p = \max\{l_i, 1 \leq i \leq k\}$ then the execution time of the loop is dominated by LBD_p.

<Proof>

In terms of a LBD_i, it causes a delay for l_i units at each successive dependence in the loop. As the dependence distances are the same for all LBDs, the dependences will occur at the same iterations for all distinct LBDs. Since all of the delay times l_i, that arise from each distinct LBD can be satisfied by a maximum delay, the LBD with longest lexical distance, which will result in the longest delay, dominates the execution time of the loop. □

Lemma 3.3 If $l_1 = l_2 = ... = l_k$ and $d_p = \min\{d_i, 1 \leq i \leq k\}$ then the execution time of the loop is dominated by LBD_p.

<Proof>

In terms of a LBD_i, the delay time is l_i for every other d_i iterations. The lexical

distance for all LBDs is the same, implying the delay times for each LBD are equivalent. As the dependence for a LBD with shorter dependence distance will occur at earlier iterations, all of the delay at each successive dependence for each LBD can be satisfied by a LBD with minimum dependence distance. So the LBD with minimum dependence distance dominates the execution time of the loop. □

Lemma 3.4 If $l_p = \max\{l_i, 1 \le i \le k\}$ and $d_p = \min\{d_i, 1 \le i \le k\}$ then the execution time of the loop is dominated by LBD_p.

<Proof>

Based on Lemmas 3.2 and 3.3, the proof is straightforward. □

Lemma 3.5 If $d_i = d_j$ and $l_i = l_j$ for all i and j, where $i \neq j$ and $1 \le i, j \le k$, then the execution time of the loop is dominated by any LBD_i, where $1 \le i \le k$.

<Proof>

The proof is obvious. □

Lemma 3.6 If there exists a pair of lexical backward dependences LBD_s and LBD_t, where s ≠ t, $l_t > l_s$, and $d_t > d_s$, then the execution time of the loop is dominated by multiple lexical backward dependences.

<Proof>

Based on Lemmas 3.2 and 3.3, there does not exist a single dominating LBD, implying that multiple dominated LBDs exist in the loop. □

The assumption of (3.2), that each iteration suffers equal delay to its preceding iteration, is reasonable to deal with the timing of a loop with multiple dominated LBDs, because the delay at each iteration is implicitly determined by some LBD. Formula (3.2) thus fits the calculation of the execution time for this type of loop.

Lemmas 3.2 to 3.6 imply that if a single dominated LBD can be identified in a loop, formula (3.6) can be used to compute the execution time of the loop. For the timing of a multiple loop, (3.6) can be generalized as

$$T(loop_l) = \sum_{i=1}^{l} ((\lceil N_i/d_i \rceil - 1)\, l_i + m_i), \tag{3.7}$$

if there exists a single dominated LBD at each surrounding loop.

To get an improved timing formula without the aforementioned disadvantages, (3.2) and (3.7) can be incorporated to form:

$$T(loop_l) = \sum_{i=1}^{l} \begin{cases} ((\lceil N_i/d_i \rceil - 1)\, l_i + m_i) & \text{if } a \textit{ single dominated } LBD_i \textit{ exists} \\ ((N_i - 1) \max\{l_i/d_i\} + m_i) & \text{if } \textit{multiple dominated } LBD_i \textit{ exist} \end{cases} \tag{3.8}$$

3.3.3 An Algorithm of Statement Reordering

Based on the above discussion, to minimize the execution time of a loop with dependence cycles, the statement ordering needs to be adjusted globally such that the resulting alignment has minimum delay. We first introduce an algorithm for a simple dependence cycle and then generalize it to fit any dependence cycles.

3.3.3.1 Simple Dependence Cycle

Consider a simple dependence cycle consisting of m (> 1) statements, i.e., S_0 $\delta_1/[\delta_1]$ S_1 $\delta_2/[\delta_2]$ ... $\delta_{m-1}/[\delta_{m-1}]$ S_{m-1} $\delta_0/[\delta_0]$ S_0, where S_i does not necessarily appear earlier

than $S_{(i+1) \bmod m}$ for all i $(0 \le i \le m-1)$. Let l_i and d_i be the lexical distance and dependence distance of $\delta_i/[\delta_i]$. The algorithm of statement ordering is shown below:

1. For the given alignment find the maximum delay per iteration, i.e., $\max(l_k/d_k)$ among existing LBD_ks. Let it be p.

2. Sort all d_is (> 0) in the cycle. Suppose they are $\lambda_1 > \lambda_2 > \ldots > \lambda_k$, where $1 \le k \le m$.

3. For an LBD in the cycle, the minimum and maximum possible lexical distance is 2 (because $m > 1$) and m. Let $2/\lambda_1$ and $\min(m/\lambda_1, p)$ be the lower and upper bounds of delay per iteration in this loop.

4. Collect all of the *possible* delays between the lower and upper bounds of delay per iteration, inclusively, in the following order: $2/\lambda_1$, $2/\lambda_2$, ... , $2/\lambda_k$, $3/\lambda_1$, ... , $(m-1)/\lambda_k$, m/λ_1. Ignore those values which are greater than the upper bound of delay.

5. For each delay time obtained, use formulas (3.2) and (3.6) to calculate two distinct *accumulated delay times* (ADTs) respectively.

6. Sort the ADTs in ascending order. Discard the ADTs which are greater than or equal to upper bound of ADT, i.e., min(ADT derived from m/λ_1, ADT derived from p). Merge the ADTs that are equal in value and in lexical and dependent distances of the maximum delay per iteration, but are different in the timing formula used to compute ADT.

7. For a specific delay $n/lambda_i$ obtained in step 4, it is always feasible to order a pair of statements to form an initial partial alignment (two statements) with LBD of lexical distance n and dependence distance λ_i. Recognizing this, for the sorted

ADTs, use *binary search* approach to do the following steps until either an alignment with minimum ADT is obtained or the upper bound of ADT is reached.

8. For a given ADT, there exists a corresponding delay n/λ_i (ADT is computed from n/λ_i). Find the set of all possible alignments involving only two statements with LBD of lexical distance n and dependence distance λ_i.

9. Use the algorithm in Figure 3.3, which is a *branch and bound* technique [46], to extend the partial solution. If the ADT is derived from formula (3.2), extend the partial alignment in a manner such that *multiple* dominated LBDs with maximum delay n/λ_i will result. If the ADT is derived from formula (3.6), order the statements in a way that will lead to an alignment with a *single* dominated LBD and delay n/λ_i. If the ADT is a merged one, i.e., it is derived from formulas (3.2) and (3.6), order the statements in a way without regard as to whether a single or multiple dominated LBDs will be finally derived.

10. If a full alignment is achieved go back to step 8 and test the existence of alignment of next smaller ADT (in binary serach approach). Otherwise, test the alignment for next larger ADT in the same manner.

11. For the delay $m/max\{d_i\}$ the corresponding alignment always exists. Beginning with the first statement, that is: $S_{p \bmod m}\ \delta_{(p+1) \bmod m}/[\delta_{(p+1) \bmod m}]\ S_{(p+1) \bmod m}\ \delta_{(p+2) \bmod m}/[\delta_{(p+2) \bmod m}] \cdots\ \delta_{(p+m-1) \bmod m}/[\delta_{(p+m-1) \bmod m}]\ S_{(p+m-1) \bmod m}\ \delta_p/[\delta_p]\ S_{p \bmod m}$ $(0 \leq p \leq m-1)$, assuming $d_p = \max\{d_i\}$. This algorithm either preserves the original alignment or generates a near-optimal statement order.

For example, in Figure 3.4(a) $S_1\ [\delta_2]\ S_2\ [\delta_3]\ S_3\ [\delta_1]\ S_1$, where $d_1 = 2$, $d_2 = 3$, and $d_3 = 2$. The approach to derive a near-optimal statement order is shown below.

Input

1. lexical distance n, dependence distance λ_i, maximum delay per iteration n/λ_i, the corresponding ADT X, and the timing formula(s), formula (2), (6) or both, used to get X
2. set of all possible initial alignments $A_1 = \{S_x^p\ S_y^q\}$
   where S_x^p is statement S_x at pth lexical position of the full alignment
   S_y^q is statement S_y at qth lexical position of the full alignment
   $x \neq y,\ p \neq q$
   LBD between S_x^p and S_y^q has lexical distance n and dependence distance λ_i
3. full_alignment ← false
4. alignment (set of statements) ← ϕ

Output

a full alignment is achieved or full_alignment = false

Algorithm

```
k ← 1
while (k > 0) and (not full_alignment) do
    while A_k ≠ φ do
        get a_k = {S_x^p, S_y^q}, where S_x^p S_y^q ∈ A_k
        A_k ← A_k − {S_x^p S_y^q}
        alignment ← a_k + alignment
        if alignment is a full alignment then
            full_alignment ← true
            A_k ← φ
        else
            k ← k + 1
            compute A_k = {S_l^r S_m^s}
            where S_l^r ∈ alignment or S_m^s ∈ alignment
                  LBD (if any) between S_l^r and S_m^s has delay ≤ n/λ_i
                  Single (X from formula (6)) or multiple (X from formula
                  (2)) dominated LBDs in alignment + {S_l^r, S_m^s}
        endif
    endwhile
    if not full_alignment then
        k ← k − 1
        alignment ← alignment − a_k
    endif
endwhile
```

Fig. 3.3 A branch and bound algorithm

(1) For the given alignment the maximum delay per iteration p is l_1/d_1 = 3/2, where 3 is the lexical distance l_1.

(2) The dependence distances in the cycle are sorted as 3 (λ_1) > 2 (λ_2).

(3) The lower and upper bounds of maximum delay per iteration are 2/3 and $\min(m/\lambda_1, p)$ = 3/3, where m = 3, respectively.

(4) The possible maximum delays per iteration between the lower and upper bounds of delay per iteration, inclusively, are 2/3, 2/2, 3/3.

(5) The accumulated delay times (ADTs) computed from formulas (3.2) and (3.6) for each of the possible maximum delays per iteration in (3.4) are 9 (from delay 2/3), 12 (from delay 2/2), 12 (from delay 3/3), and 9 (from delay 2/3), 11 (from delay 2/2), 12 (from delay 3/3), respectively.

(6) The upper bound of ADT is min(ADT derived from m/λ_1 (3/3) by formula (3.6), ADT derived from p (3/2) by formula (3.8)) = 12. Thus the sorted ADTs in ascending order are: 9 (from delay 2/3, formulas (3.2) and (3.6)), 11 (from delay 2/2, formula (3.6)), 12 (from delay 3/3, formula (3.6)).

(7) The first alignment to be tested for existence is the one with corresponding ADT 11 (from delay 2/2, formula (3.6)). Such an alignment can be derived by the algorithm in Figure 3.3. But the next tested alignment with corresponding ADT 9 (from delay 2/3, formulas (3.2) and (3.6)) cannot be derived. Thus the desired near-optimal alignment is the one with ADT 11 (from delay 2/2, formula (3.6)), which is shown in Figure 3.4(b).

Doacross 10 I = 1, 10	Doacross 10 I = 1, 10
S_1 A(I+3) = C(I) * 4	S_1 A(I+3) = C(I) * 4
S_2 B(I+2) = A(I) + 3	S_3 C(I+2) = B(I) – 2
S_3 C(I+2) = B(I) – 2	S_2 B(I+2) = A(I) + 3
10 Continue	10 Continue
(a) The original loop	(b) After statement reordering

Fig. 3.4

The branch and bound technique discards all alignments that place statements in a partial alignment where the resulting delay exceeds the assigned delay, or the undesired single (multiple) dominated LBD (LBDs) is (are) derived, or when the resulting alignment violates the semantics. This approach can be expensive. In practice, step 9 can be implemented in a flexible manner to obtain a tradeoff between the maximum parallelism and cost.

3.3.3.2 General Dependence Cycles

Multiple dependence cycles can occur in the statements. We first address the strategy of statement reordering for a loop with two dependence cycles. Thereafter, a statement-ordering algorithm for general dependence cycles is presented.

All the LBDs in each cycle contribute to the delay of a loop with multiple dependence cycles. For a dependence cycle with m (> 1) statements and n loop-carried dependences, each with a dependence distance d_i, the possible delay per iteration for arbitrary alignment of statements will range from $2/\max(d_i)$ (lower bound) to $m/\min(d_i)$ (upper bound). Consider a loop with two dependence cycles, $cycle_i$ and $cycle_j$. The relationships of the delay per iteration for these two cycles are shown in Figure 3.5, where l_p and u_p denote the lower bound and upper bound of possible delay in $cycle_p$.

Figures 3.5A and 3.5B show that the delay per iteration due to $cycle_j$ is greater than or equal to that of $cycle_i$. In terms of this delay relation, if these two cycles are separated from each other, or $cycle_j$ is totally involved in $cycle_i$, the delay due to the LBDs in $cycle_j$ is definitely longer than that of $cycle_i$. Therefore, the execution time of the loop is dominated by the ordering of the statements in $cycle_j$. For statement ordering of the loop, only the statements involved in $cycle_j$ need to be considered. However, for any other relations between these two cycles, i.e., connected, overlapped, etc., the statement ordering in $cycle_j$ inevitably affects that of $cycle_i$. That is, in Figure 3.5, A and B only reflect a static delay relation, i.e., an independent possible delay between

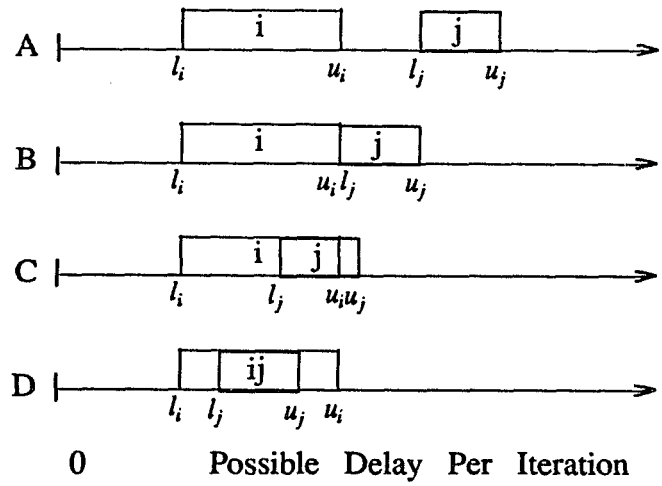


Fig. 3.5 The relations of delay between two dependence cycles

these cycles. The actual delay relations can be like that of Figure 3.5C or 3.5D. This relation implies that an optimum alignment cannot account for the statement order in any single cycle. It is necessary to find a minimum delay which can satisfy both cycles. In implementation, statement reordering begins with $cycle_j$. Further extension to fulfill the alignment of $cycle_i$ can be done only when the delay in $cycle_j$ is satisfied.

Figures 3.5C and 3.5D present a partially or totally overlapped relation between the delays in each cycle. Regardless of the relative positions between these two cycles, the delay of the loop is nondeterministically determined by the LBDs in either cycle. As the lower bound of delay for $cycle_j$ is greater than that of $cycle_i$, theoretically, the minimum delay of the loop is dominated by $cycle_j$. Similarly, statement reordering begins with $cycle_j$ and then extends to $cycle_i$. A full optimum alignment is the one in which the delay in $cycle_j$ is the minimum one among all those which satisfies both $cycle_i$ and $cycle_j$.

The concepts mentioned above facilitate the generalization of the statement ordering:

1. Trim the dependence distances by removing redundant dependences. That is, if a pair of interdependent statements with two different dependence distances exists, then by Lemma 3.3, the one with longer distance can be ignored.
2. Find the (maximum) delay for the given alignment. Let it be q.
3. Compute the lower bound of possible delay for all dependence cycles.
4. Identify the dependence cycle with maximum lower bound of possible delay. Assume it is $cycle_p$.
5. Order the statements in $cycle_p$ by the simple cycle algorithm, using q as the upper bound of possible delay. Keep the order of other statements *unchanged.*
6. When a valid partial alignment is achieved for $cycle_p$, check whether the delay for $cycle_p$, say, t, can satisfy the delays of other cycles. If it does, the full alignment is obtained. Otherwise, do the next step.
7. Preserve the partial optimum alignment for $cycle_p$. Do the following steps for

each of the remaining cycles:

(a) Sort the loop-carried dependence distances d_i in the cycle.

(b) Let $2/\max\{d_i\}$ ($1/\max\{d_i\}$ for self-dependence cycle) and t be the lower and upper bound of delay. Use the approach mentioned in the simple cycle algorithm to derive all possible delays.

(c) Sort the possible delays in ascending order.

(d) Employ the algorithm in Figure 3.3 to align the statements in the cycle; using the possible delays in (c) as the bounds subsequently.

(e) If a valid alignment for the cycle is obtained, leave the statements involved in the cycle unchanged. Do step 7(a) for next cycle. Otherwise, go back to step 4 to test the succeeding ADT.

8. If all the partial alignments for each cycle are obtained, the full alignment is achieved.

For example, in Figure 3.6(a) S_1 $[\delta_2]$ S_2 $[\delta_3]$ S_3 $[\delta_1]$ S_1 which forms a dependence cycle. This dependence cycle also involves another sub-cycle, i.e., S_2 $[\delta_3]$ S_3 $[\delta_{2'}]$ S_2. By using the aforementioned algorithm, the step-by-step approach to derive a near-optimal alignment is shown below.

(1) For the given alignment the maximum delay per iteration is 3/3 = 1. By Lemma 3.4 there exists a single dominated LBD. The ADT derived is 102.

(2) The lower bounds of possible delays for the major and sub- dependence cycles are 2/4 and 2/5 respectively. So the major dependence cycle is with higher lower bound.

(3) Order the statements in the major cycle by the simple cycle algorithm. This

first leads the alignment S_3 S_2 S_1 (in lexical order) to be achieved. To this alignment the maximum delay per iteration is 2/2 and the ADT is 101. Meanwhile, this maximum delay also can satisfy the maximum delay of the sub-cycle S_3 [$\delta_{2'}$] S_2 [δ_3] S_3, which is also 2/2. Further search for the near-optimal alignment leads the final alignment, shown in Figure 3.6(b), is achieved. In this alignment the maximum delay per iteration is 3/4 and the corresponding ADT is 75.

```
      Do 10 I = 1, 100                 Do 10 I = 1, 100
S1    A(I+4) = C(I+2)          S2    B(I+2) = A(I)+C(I)
S2    B(I+2) = A(I)+C(I)       S3    C(I+5) = B(I)
S3    C(I+5) = B(I)            S1    A(I+4) = C(I+2)
      10 Continue                      10 Continue
```

(a) The original loop (b) After statement ordering

Fig. 3.6

3.3.3.3 A General Local Statement-Ordering Approach

As the lexical distance of an LBD affects parallelism of a given loop, reducing and/or adjusting the lexical distances locally for all existing LBDs may result in improvement of performance. A general local statement-ordering technique is introduced below.

Let $Sc(LBD_i)$ and $Sk(LBD_i)$ represent the lexical order of the source and sink statements of LBD_i in an alignment. By definition, $Sc(LBD_i) > Sk(LBD_i)$ (ignore the case $Sc(LBD_i) = Sk(LBD_i)$). For two distinct lexical backward dependences, LBD_p and LBD_q, there are four possible relations between them.

Case 1: LBD_p and LBD_q are separated, i.e., $Sc(LBD_p) > Sk(LBD_p) > Sc(LBD_q) > Sk(LBD_q)$.

Case 2: LBD_p and LBD_q are connected, i.e., $Sc(LBD_p) > Sk(LBD_p) = Sc(LBD_q) > Sk(LBD_q)$.

Case 3: LBD_p and LBD_q are partially involved, i.e., $Sc(LBD_p) > Sc(LBD_q) > Sk(LBD_p) > Sk(LBD_q)$.

Case 4: LBD_p and LBD_q are totally involved, i.e., $Sc(LBD_p) \geq Sc(LBD_q)$ and $Sk(LBD_p) \leq Sk(LBD_q)$.

In addition, a set of LBD_is are *perfectly* involved in an LBD_j if these LBD_is are connected to each other and $Sc(LBD_j) = \max(Sc(LBD_i))$ and $Sk(LBD_j) = min(Sk(LBD_i))$.

Figure 3.7 presents an example for these relations.

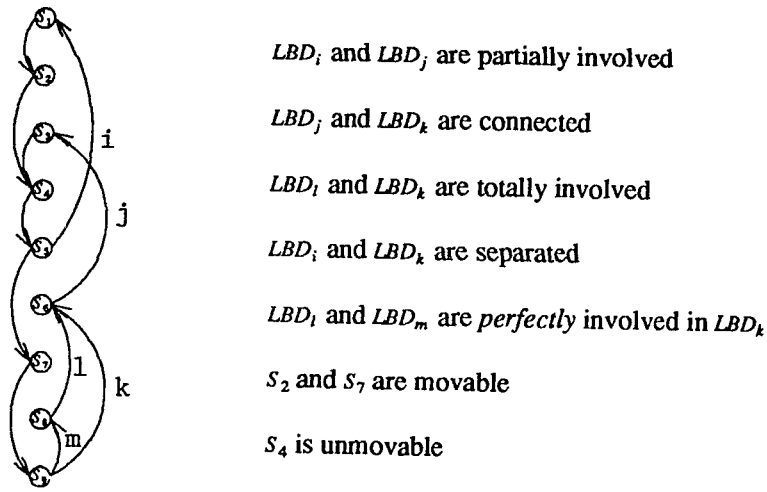


Fig. 3.7 The relationships between various LBDs

Two strategies are concerned with the decrease of lexical distance(s): *statement moving* and *distance reallocation*. *Statement moving* deals with the identification and moving of statements located in the domain of an LBD that are movable. *Distance reallocation* refers to the allocation of the lexical distances on the basis of dependence

distances for a set of consecutive interrelated LBDs. Distance reallocation is available when movable statements are in the domains of the LBDs, but, due to dependences, they cannot be moved out of the entire scopes of these LBDs. We will define that type of statement shortly.

Let $\gamma(S_i)$ be the lexical position of the statement S_i and $p(a,b)$ be a path from the statement with lexical order a to the statement with lexical order b in a dependence graph. Then, $p(a,b)$ is a *direct path* if there does not exist a node (statement) S_p on the path where $\gamma(S_p)$ is not lexically between a and b. Let $P(a,b)$ be a set of direct paths $p(a,b)$.

Since the statements to be dealt with are strongly connected via true dependences, a statement S_j which can be moved away from the domain of some specific LBD_k must satisfy these two conditions:

1. S_j is neither the source nor the sink statement of any LBD.
2. S_j is not on the path of any path in $P(Sk(LBD_k),Sc(LBD_k))$.

In Figure 3.7, the dependence graph shows some examples of movable statements.

A movable statement has limitation as to the locations to which it can be moved. For a movable statement S_j, the allowable range of movement is between $\gamma(S_j)$ and $\gamma(S_p)$, exclusively, where S_p must meet three conditions: (1) S_j $\delta/[\delta]^+$ S_p (S_j $\delta/[\delta]$ S_p or S_j $\delta/[\delta]$ ... $\delta/[\delta]$ S_p), (2) S_p is the source statement of some LBD, and (3) S_p is the first (in lexical order) statement which meets the conditions (1) and (2). But, the original dependences of S_j with other statements also limits its moving. That is, if S_j $\delta/[\delta]$ S_q and S_q is also movable, then the lexical order for S_j always needs to be kept less than

S_q irrespective of where it is moved.

The distance reallocation technique is available only for consecutive connected and/or partially involved LBDs. We first present the theoretical foundation for optimum reallocation of the lexical distances.

Lemma 3.7 The *minimum delay* for m consecutive connected and/or partially involved LBDs, LBD_1, LBD_2, ... , LBD_m in a loop, each with dependence distances d_1, d_2, ... , d_m and lexical distances l_1, l_2, ... l_m, will result, in general, if $l_1/d_1 = l_2/d_2 = \cdots = l_m/d_m$, assuming no movable statements in the overlapped area of the LBDs.

<Proof>

Let $k = l_1/d_1 = l_2/d_2 = \cdots = l_m/d_m$. By formula (2), the delay per iteration due to these LBDs is k.

Assume k is not the minimum delay for this particular alignment. Then, there exists another allocation of the lexical distances, l_1', l_2', ... , l_m', to which the delay, i.e., $\max\{l_1'/d_1, l_2'/d_2, ..., l_m'/d_m\}$, is less than k.

Suppose $\max\{l_1'/d_1, l_2'/d_2, ..., l_m'/d_m\} = l_p'/d_p$. Then $k = l_p/d_p > l_p'/d_p$. This results in $l_p > l_p'$. Let $l_p - l_p' = s$. Because there are no movable statements in the overlapped area,

$$l_1 + l_2 + \cdots + l_m = l_1' + l_2' + \cdots + l_m' \tag{3.9}$$

If l_p' is less than l_p by s, by (3.9) there must exists a l_q', where $1 \le (l_q' - l_q) \le s$. Therefore, $l_q'/d_q > l_q/d_q$ because $d_q = d_q$. That is, $l_q'/d_q > l_q/d_q = k > l_p'/d_p$.

Therefore, $l_p'/d_p \neq \max\{l_1'/d_1, l_2'/d_2, ..., l_m'/d_m\}$. This contridicts the assumption. So

the minimum delay is k. □

Lemma 3.7 implies an important fact: for a set of connected and/or partially involved LBDs, reallocation of lexical distance may be valuable if there exist movable statements and the delays for each LBD are not equal. Allocation of the lexical distance is carried out in a manner to equalize the delay in each LBD.

If reallocating a set of interrelated LBDs is required, we need to find the optimum lexical distance for each LBD. Assume there exist m connected and/or partially involved LBDs, LBD_i, where $1 \le i \le m$. Let l_i and d_i denote the lexical and dependence distances of LBD_i. So there exist at least two distinct LBDs, LBD_p and LBD_q, where $l_p/d_p \neq l_q/d_q$. Suppose the optimum lexical distance for each LBD_i is l_i', for which the minimum delay results. Then $l_i'/d_i = k$ for all i. As $\sum_{i=1}^{m} l_i = k \sum_{i=1}^{m} d_i$, we get $k = \sum_{i=1}^{m} l_i / \sum_{i=1}^{m} d_i$. Thus the reallocated lexical distance l_i' for LBD_i is

$$l_i' = (\sum_{i=1}^{m} l_i / \sum_{i=1}^{m} d_i)\, d_i$$

When the proper distance for each LBD is known, we can adjust the distance for them based on the existing movable statements in each LBD, and the locations to which they can be moved. Note, the value of l_i' may not be an integer. It is necessary to truncate or round it to get an integer allocation of lexical distance. As the delay l_i'/d_i for shorter d_i is changed sharply with the minute variation of l_i, the value of l_i' for smaller d_i needs to be truncated if it is not an integer. On the contrary, l_i' needs to be rounded for large d_i. In simplicity, d_i can be differentiated as large (small) if it is greater (less) than the mean value of all d_i. In addition, the movable statements may not scatter in

each LBD evenly, for each LBD we simply adjust its lexical distance to be the closest l_i' value. That is, this technique may not achieve the best parallelism to a specific alignment.

Based on these principles, a general algorithm to locally adjust the lexical distances is proposed:

1. Find all LBDs. Get Sc(LBD_i) and Sk(LBD_i) for each LBD_i.
2. Identify the movable statements in each LBD and their movable locations.
3. For each LBD_i do the following:
   - If LBD_i is connected or partially overlapped with other LBD(s), move the statements out of the entire domains of the group of LBDs, if possible, and out of the overlapped areas.
   - Otherwise, move the movable statements, if any, out of the domain of LBD_i but not into other *isolated* LBD domains.
4. For each group of connected and/or partially overlapped LBDs do the following.
   - Ignore the LBD, if any, which perfectly involves a set of connected LBDs but the delay for it is less than that of the connected LBDs; otherwise, those connected LBDs can be ignored.
   - If no movable statements are inside the domain of any component LBD, take no action.
   - If the delays for each component LBD are equal, take no action.
   - Otherwise, use the distance reallocation technique derived from Lemma

3.8 to find the optimum distance for each component LBD. Then adjust the distance locally.

Figure 3.8 shows the application of this algorithm to the dependence cycles. The resulting delay is less than that of the original alignment.

(a) Before local statement ordering (b) After local statement ordering

Fig. 3.8 Local Statement Ordering

3.3.4 Quantity of Processors Required for a Doacross Loop

The statement reordering algorithms in Section 3.3.3 are developed under the assumption that the number of processors in the system is unlimited. Clearly, finding an optimum quantity of processors for a given loop is required in order to achieve the same parallelism level.

Lemma 3.8 For a parallel loop with m statements involved in dependence cycles, the quantity of processors required will be $(\lceil m/l \rceil)\, d$, where d and l are the dependence distance and lexical distance of the single *dominated* LBD, if it exists.

<proof>

By definition, starting from the first iteration, synchronization-delay for l time units occurs in every d iterations *regularly* across the loop. That is, the $(nd + 1)$th

iteration will delay for nl time units, where $n \geq 1$. If $nl \geq m$ (the execution time of m statements), the iterations from $(nd + 1)$th to $2nd$th iteration can be processed by the processors executing the iterations from 1st to ndth iteration without losing parallelism. Similar situations happen for $(2nd + 1)$th to $3nd$th iteration, $(3nd + 1)$th to $4nd$th iteration, and so on. If $nl \geq m$ then $n = \lceil m/l \rceil$. So the processors required will be $(\lceil m/l \rceil)d$. □

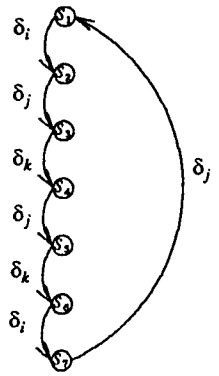
For a parallel loop with dependence cycles for which multiple dominated LBDs exist, Cytron [20] has derived the optimum quantity of processors required.

3.4 Statement Ordering on the parallelism of Multiple Nested Loop

The statement reordering concept can be extended to a multiple perfectly nested loop with global dependence cycles [6]. A global dependence cycle refers to a dependency in which the dependence links (relations) originate from different surrounding loops, forming a recurrence. This dependence occurs generally for statements with variables of multi-dimensional arrays. Figure 3.9 shows the model.

```
Do 10 I = 1, N
  Do 10 J = 1, N
    Do 10 K = 1, N
S1      C(I+2) = B(4,J)
S2      B(J+1,3) = C(I) - 1
S3      A(K+1) = B(J,3) * 3
S4      E(J+1) = A(K)/4
S5      D(I+2,J,K+3) = E(J) + 1
S6      C(I+3) = D(I+2,J,K)
S7      B(4,J+1) = C(I+1)
10 Continue
```

(a) A multiple nested loop



(b) The dependence graph

Fig. 3.9 A Multiple Doacross loop with a global dependence cycle

3.4.1 Simple Global Dependence Cycle

To minimize the delay of a multiple loop with global dependence cycles, based on (3.8), the statements need to be reordered in a way such that the delay that arises from each surrounding dependent loop is minimized. As the *upper bound* of the loop index also accounts for the accumulated delay for each loop, this parameter also needs to be considered for optimum ordering of statements.

Suppose there exists a p-layer perfectly nested loop, $loop_1$, $loop_2$, ... , $loop_p$, where $p > 1$. In the loop body there are m statements, S_1, S_2, ... , S_m, for which the dependences form a simple global dependence cycle, i.e., S_1 $\delta_i/[\delta_i]$ S_2 $\delta_j/[\delta_j]$ ... $\delta_k/[\delta_k]$ S_m $\delta_l/[\delta_l]$ S_1, where $1 \le i, j, k, l \le p$ ($\delta_i/[\delta_i]$ denotes the true dependence due to $loop_i$). Theoretically, the algorithm of statement ordering for maximum parallelism is given below.

1. For the given alignment, find the maximum delay for each related loop.
2. Compute the overall accumulated delay by formula (3.8) for the multiple loop, suppose it is p.
3. Sort the loop-carried dependence distances for each individual loop in which dependences occur in the statements. Suppose the order for ith level of loop is: $d_{i1} > d_{i2} > ... > d_{ik}$, where d_{ik} denotes a dependence distance d_k for $loop_i$.
4. For each surrounding dependent loop let $2/d_{i1}$ be the lower bound and m/d_{i1} be the upper bound of delay. Find all possible delay per iteration between the lower and upper bounds of delay based on the existing dependence distances.
5. For each delay use formulas (3.2) and (3.7) respectively to estimate the accumulated delay in each loop. Thus for each accumulated delay time there exists a

corresponding delay per iteration.

6. Sort the accumulated delay time for each loop separately.

7. In terms of a dependent loop i it is always feasible to align the statements such that the only delay is m/d_{i1} if $d_{i1} > 0$. The accumulated delay times ADT_{i1} can also be computed. Find the minimum accumulated delay time among those surrounding dependent loops, i.e., $\min\{ADT_{i1} \mid 1 \le i \le p\}$. Let it be q.

8. Let the minimum accumulated delay time in step 5 be the lower bound and min{p,q} be the upper bound. Eliminate the accumulated delay in step 5 which is greater than the upper bound for each loop. Find all possible *combined accumulated delay times* which are between the lower and upper bounds of accumulated delay times. For example, let x and y be the accumulated delay for $loop_i$ and $loop_j$. If $x + y \le \min\{p,q\}$ then $x + y$ is a valid combined delay time. Sort all of these accumulated delay times in ascending order.

9. For each accumulated delay time, there exists a corresponding delay for $loop_i$ (or combined delay time for multiple loops). Do the following steps in order until one corresponding alignment is obtained.

- if the accumulated delay time is for $loop_i$ and calculated from (3.2), use the algorithm in Figure 3.3 to order the statements with $loop_i$-carried dependences, such that the resulted alignment with multiple dominated $loop_i$-carried LBDs and the maximum delay is the tested delay.
- if the accumulated delay time is for $loop_i$ and calculated from (3.7), use the algorithm in Figure 3.3 to order the statements with $loop_i$-carried dependences, such that the resulted alignment with single dominated

$loop_i$-carried LBDs and the maximum delay is the tested delay.

- if the accumulated delay time is a combined multiple loops' delay, say, ith and jth loops, use the algorithm in Figure 3.3 to order the statements such that the resulted alignment containing single and/or multiple dominated LBDs, with maximum delay time equals to the tested combined delay time for ith and jth loops, respectively.

In practice, this algorithm can be simplified to meet individual needs. For example, the simplest approach for a simple global statement ordering is to compare the overall accumulated delays p and q computed at steps 2 and 7. If $q < p$, reorder the statements such that m/d_{i1}, where $1 \le i \le p$, is the only delay in the multiple loop and $q = \min\{ADT_{i1} \mid 1 \le i \le p\}$ (as stated in step 7, this alignment is always feasible if $d_{i1} > 0$). Otherwise, keep the original alignment unchanged.

3.4.2 Joint Effects of Statement Reordering and Processors Scheduling

Although the parallelism level of a multiple nested Doacross loop is inherently associated with the alignment of statements, the processor scheduling scheme for a loop has significant impact on the exposure of the inherent parallelism. A shortest-delay self-scheduling (SDSS) strategy is claimed to give near-optimal performance in most cases [51].

The SDSS approach assigns the iterations of a multiple perfectly nested loop to limited processors based on the order of the theoretical execution-starting time of each iteration. That is, due to data synchronization, each iteration in the loop suffers a delay time before it can be executed. The delay time clearly varies with iterations. The earli-

est allowable starting time of a iteration is the theoretical execution-starting time. The SDSS approach modifies the loop index values properly to satisfy the need for theoretical execution order, resulting in a near-optimal performance. To a given alignment of statements in a loop, the SDSS scheme exposes fully the inherent parallelism in the alignment.

The SDSS approach simply exposes the existing parallelism for a given alignment but does not exploit the potential parallelism in the loop. However, the statement ordering algorithm is capable of maximizing the exploitation of the potential parallelism in the statements of a loop. Obviously these two techniques need to be incorporated to maximize parallelism for a loop.

3.5 Summary of Results

In parallel processing of Do-loops, in addition to processor-scheduling, the exploitation of statement-level parallelism is another approach to improve performance. Due to data dependence, the level of parallelism is inherently associated with an alignment of statements. Data dependence can be removed or alleviated to some level by statement reordering technique.

In general, the lexical loop-carried backward dependences in statements explicitly inhibit the exposure of parallelism. In an unbreakable dependence cycle of a loop, the lexical loop-carried backward dependences inevitably inhibit the parallelism exposure of the statements. Optimum reordering of the statements can form lexical backward dependences which results in maximum parallelism.

The level of parallelism reflects the execution time of the loop. Cytron [21]

developed a general timing formula to estimate the execution time at compile-time. Some limitations in this formula were observed. For a p-layer multiple nested loop, the maximum absolute accumulated error from the formula is $\sum_{i=1}^{p} (l_i/d_i)(d_i - 1)$, where l_i and d_i is the lexical distance and dependence distance of a lexical backward dependence that results in the maximum delay of the ith-level loop. The impact of the limitation of this formula depends on its application domain. Statement reordering or loop scheduling based on this formula may lead to inefficiency, depending on program structure. An improved timing formula was developed.

Based on the improved formula, a statement ordering algorithm for a simple dependence cycle in a loop is described in detail. This algorithm is further extended to cover general dependence cycles. To meet practical use, a local adjustment technique of the lexical distances for all lexical backward dependences is proposed. The technique moves the movable statements in the domains of LBDs and reallocates the lexical distances for a group of interrelated LBDs. In addition, to achieve the maximum parallelism of a loop with some types of dependence, a formula used to compute the minimum required quantities of processors is derived.

The statement reordering concept on global dependence cycles of multiple nested loops is also available. An algorithm for simple global dependence cycle to achieve near-maximum parallelism is proposed.

The effect of jointly using this statement reordering strategy and the processor dynamic shortest-delay self-scheduling on the performance of multiple nested loops is obviously superior to the effect of their individual use.

Chapter 4

Subroutine Parallelization

4.1 Introduction

When multiprocessing a Fortran program, the subroutine is a construct that potentially can be executed in parallel. In practice, one way to access the multitasking facilities of the available multiprocessors (Cray X-MP, IBM-3090, BBN Butterfly, ELXSI 6400, Flex/32, etc.) is to use parallel **call** statements, which create parallel tasks to execute the called subroutines. A parallel compiler (or translator) should thus be able to exploit the parallelization of subroutines.

A subroutine can be computed in one of the three modes: (1) serial, (2) asynchronous parallel, and (3) synchronous parallel, depending on inter- and intra-procedural data dependence. For example, consider two successive calls for subroutines A and B in a program, where the call to A appears earlier than the call to B. Suppose the intraprocedural statement-execution order is unchangeable for both A and B. If there exists no interprocedural data dependence between A and B, they are available for asynchronous parallel execution. If there exists interprocedural data dependence, but the source of dependence is not the last statement of A and the sink of dependence is not the first statement of B then they can be processed in synchronous parallel. Otherwise, A and B are only suitable for sequential computing.

Generally, three problems are encountered for subroutine parallelization: (1) the precedence of parallel execution of subroutines, (2) the execution mode for each subroutine, and (3) the restructuring of the program to cope with parallel processing.

Researchers in this area are interested primarily in the parallelization of asynchronous subroutines, with special concern on the estimation of the global effects of the called procedures and the computation of interprocedural dependence [12,13,53]. Restructuring for a program with asynchronizable subroutines is trivial, thus is little discussed.

However, to gain more parallelism, subroutines may need to be processed in synchronous parallel if the synchronization overhead is negligible. The process of identification of synchronous subroutines includes locating statements which cause interprocedural dependence, in addition to other information required for asynchronous parallel execution. Restructuring of programs with synchronizable subroutines is not trivial. Moreover, as the parameters passed to a subroutine may vary with the call sites, the inter- and intra-procedural data dependences of a subroutine may not be constant across the program, implying the possible complexity of restructuring a program with multiply called subroutines.

This chapter presents an approach to parallelize the serial subroutines of a Fortran program. Subroutines in this chapter refer to both subroutines and functions. Section 4.2 describes the machine model and the parallel execution mode. Section 4.3 states the principle to parallelize the called subroutines for a calling subroutine. An algorithm for parallelizing the subroutines in a *program* is given in Section 4.4. This algorithm does not consider synchronization overhead. Finally Section 4.5 contains a summary.

4.2 Machine Model and Parallel Execution Mode

We first describe the machine model available for the processing of *parallelized*

subroutines as given in this chapter. We then introduce the parallel execution mode for the output code of our parallelization approach.

4.2.1 Model of the Multiprocessor

The machine model used is a *shared memory multiprocessor* system, which is same as that described in 3.2.1.

4.2.2 Target Parallel Execution Model

The input of this approach is a general sequential Fortran program with structured code. For output, we use the parallel call statement to indicate the parallel execution. This statement allows the creation and termination of a parallel task executing the called subroutine. We also use library subroutines for synchronization management. Such features are found in parallel languages such as Parallel Fortran [29].

The parallel model includes the ordinary sequential programming constructs prevalent in the Fortran language. Such constructs imply sequential execution. For simplicity, we only describe the features that enable the execution of parallel tasks and operations of parallel events.

The **parallel call** statement invokes a subroutine for asynchronous execution in the environment of a previously originated task. The calling subroutine continues to execute under its own task. The **send** subroutine signals the specified event for the calling task; while the **wait** subroutine causes the invoking task to wait until the specified event occurs. The argument for **send** and **wait** is the exclusively accessed variable. By using a parallel event, explicit tasks may synchronize their execution.

Multiple levels of parallel tasks can be created. The **parallel call** statement can be used at any level in a parallelized program. A task created by the **parallel call** is owned by the task that executed the **parallel call** statement. When multiple levels of tasks are created, a hierarchical task parallel-execution structure is established. The root task of the program is the task executing the main program. This feature is found on the IBM-3090 for Parallel Fortran language [8]. Figure 4.1 presents an example of the parallel mode.

In Figure 4.1(a) an input sequential mode Fortran program is given. The output subroutine-parallelized program is shown in Figure 4.1(b). In Figure 4.1(b), the main program is executed in sequential mode for statements S_1, S_2, ... , until statement S_n, where a subroutine A is invoked. The main program and subroutine A are then executed in asynchronous mode. When statement S_m is encountered, another subroutine B is invoked. The main program and subroutine B are executed in synchronous parallel. That is, the main task cannot continue to execute the statement S_p until the task for subroutine B finishes the execution of its statement S_q. The task that executes subroutine A also invokes subroutines C and D. Subroutine A can be executed in asynchronous mode with subroutines C and D. However, subroutines C and D must be executed in synchronous mode. The task of subroutine A finally terminates its execution when subroutines C and D are finished. Similarly, the main task terminates when the tasks for subroutines A and B have completed their jobs.

program	MAIN
dimension	K(100),L(100)
common	/S/ M1
S_1	x = (a+b)/2
S_2	y = (a-b)/2
S_3	w = 3
.	.
S_n	call A (M1,y)
.	do 5 j = 1, 100
.	K(j) = L(j)+3
.	5 continue
.	.
S_m	call B (w)
.	M1 = 25
.	.
S_p	w = (w+y)/2
.	t = (x-y)/2
.	.
.	stop
end	MAIN
subroutine	A (L,M)
common	/S/ C
.	.
call	C (C,M)
call	D (L)
.	.
end	A
subroutine	B (h)
S_1	ss = h + 2
.	j2 = k - 3
S_q	h = j2 + ss
.	.
end	B
subroutine	C (W,X)
S_1	s = W + X
.	.
.	.
end	C
subroutine	D (S)
S_1	S = S + 2
.	.
.	.
end	D

(a) An input serial program

program	MAIN
dimension	K(100),L(100)
common	/S/ M1
S_1	x = (a+b)/2
S_2	y = (a-b)/2
S_3	w = 3
.	.
S_n	**parallel call** A (M1,y)
.	do 5 j = 1, 100
.	K(j) = L(j)+3
.	5 continue
.	.
S_m	**parallel call** B (w)
.	M1 = 25
.	.
.	**wait** (w)
S_p	w = (w+y)/2
.	t = (x-y)/2
.	.
end	MAIN
subroutine	A (L,M)
common	/S/ C
.	.
parallel call	C (C,M)
parallel call	D (L)
.	.
end	A
subroutine	B (h)
S_1	ss = h + 2
.	j2 = k - 3
S_q	h = j2 + ss
.	**send** (h)
end	B
subroutine	C (W,X)
S_1	s = W + X
.	**send** (W)
.	.
end	C
subroutine	D (S)
.	**wait** (S)
S_1	S = S + 2
.	.
end	D

(b) A parallelized program

Fig. 4.1

4.3 The Principle of Subroutine Parallelization

To parallelize the called subroutines of a calling subroutine, there are five major steps: (1) computation of control dependence, (2) approximation of the global effects of subroutine calls, (3) analysis of data dependence, (4) identification of subroutine execution mode, and (5) restructuring of calling and called subroutines. Each of these steps is described below.

4.3.1 Computation of Control Dependence

During the processing of a sequential program, the execution of some statements often depend on the results of other conditional statements, such as If-, Do-, and Goto-statements. These control dependence relations must be respected whatever the execution mode of the program. By examining control dependence we can identify unnecessary parallelizing of subroutines and simplify the implementation of parallelization. That is, only the subroutine calls which are under *identical control dependences* and not lexically isolated by statements of different control dependence are candidates for execution in parallel. Meanwhile, the task which spawns subtasks executing the parallelized subroutines must wait the subtasks finishing their jobs before it execute statements of different control dependence. This strategy may be conservative and may constrain concurrency but it is general, uncomplicated, and reliable. Improper parallelization for subroutines of different control dependences can result in execution failure. Consider the example in Figure 4.2 where subroutine P1 and subroutine P2 are under different control dependences. Suppose they are processed in synchronous mode. Due to unexpected results of conditional statement S_1 at run-time,

subroutine P1 is possibly skipped. This may cause the task executing the subroutine P2 to wait forever for the occurrence of a parallel event.

S_1	if	A > B
S_2	then	**parallel call P1**
S_3	else	P = A - B
S_4		if P < 0
S_5		then J = M - P
S_6		else J = M + P
S_7		endif
S_8	endif	
S_9	K = J/3	
S_{10}	**parallel**	**call P2**

Fig. 4.2 An example of improper subroutine parallelization

Definition A control flow graph is a directed graph augmented with a unique entry node **entry** and a unique exit node **stop** where the nodes can be a statement (basic block), maximal basic blocks, or any straight-line sequence [23,25].

Definition A node S in a control flow graph is post-dominated by a node T if every directed path from S to the exit node contains T, S ≠ T and T ≠ **stop** [25].

Definition A post-dominator tree is a post-dominator graph derived from the control flow graph.

Computing the post-dominators in the control flow graph is equivalent to computing the dominators in the reverse control flow graph. There exists an efficient algorithm for computing the dominators [25].

Definition A control dependence graph is a directed graph with the same nodes as its control flow graph, where the edge from a node X to a node Y is defined if:

1. there exists a non-empty, directed path P in the corresponding control flow

graph from X to Y with all nodes in P (except X and Y) post-dominated by Y, and

2. X is not post-dominated by Y [25].

Definition A control flow graph is *reducible* if and only if the edges can be partitioned into disjoint groups called the *forward* edges and *back* edges, with the following two properties:

1. The forward edges form an acyclic graph in which every node can be reached from the initial node of the control flow graph.

2. The back edges consist only of edges whose head dominate their tails [1].

The control dependence of a calling subroutine can be computed from its control flow graph and the derived post-dominator tree [23,25]. Since we simply attempt to expose the called subroutines with the same control dependence, the nodes of the control flow graph in this step can be maximal basic blocks (statements), subroutine call statements, or conditional statements. This representation may simplify the computation of control dependence.

A control dependence graph must be acyclic to allow the identification of identical control-dependent subroutines. A *structured* calling subroutine, i.e., a subroutine with no overlapped branches in the code, will always have a corresponding acyclic control dependence graph. A structured subroutine derives a reducible control flow graph, and a reducible control flow graph derives an acyclic control dependence graph [1,23]. In other words, only structured code can be processed in this phase.

4.3.2 Approximation of the Global Effects of Subroutine Calls

The global effect of a subroutine call refers to the summary results of the global variables (in terms of the calling subroutine) modified and/or used after the execution of the called subroutine. These data are derived from the statements in the called subroutine that access **common** variables and/or parameters bound with the local variables of the calling subroutine. Any subsequent direct or indirect **calls** by the called subroutine may also contribute indirectly to the global effect. Previous studies have been aimed primarily at techniques of identifying accessed elements of array variables and merging different sets of elements with a common array variable name, regardless of the specific statements which carry those global variables [12,13,53].

In determining where a subroutine should be synchronized, the interdependent statements between subroutines need to be located. The global effects of a subroutine call should also include the identification of the statements which carry those global variables. A direct approach to obtain the global effect is to replace the formal arguments of the called subroutine by the actual global variables, and then enumerate those statements with global variables in *lexical order* in the *calling subroutine* environment. This approach is similar to the *call by name* concept but it uses the substituted arguments in a different way. We call this method *partial in-line expansion* of subroutine calls in contrast with parallelization of calls by in-line expansion.

Partial in-line expansion has the following advantages:

1. The aliasing problem is simpler because of direct replacement of formal arguments by actual variables.
2. The *forward analysis* [2] required by other parallelization approaches to get

information such as interprocedural constant propagation, global variables binding relations, and aliasing analysis, is omitted.

3. It is feasible to identify the statements which cause interprocedural dependence.

4. The process of merging the global effect for distinct statements, such as computing the convex hull of the set of the array elements to form the global effect of subroutine is omitted.

5. The process of converting the formal arguments of called subroutines to the actual variables of calling subroutine employed by other parallelization approaches is omitted.

However, considerable code space may be needed for partial in-line expansion.

Definition $OUT_g(S_j^i)$ and $IN_g(S_j^i)$ refer to the set of modified and used *global variables* of a statement S_j in a subroutine i, respectively.

In implementation, to simplify the computation of the subsequent data analysis phase, for a statement S_j accessing global variables in subroutine i we in fact enumerate $OUT_g(S_j^i)$ and/or $IN_g(S_j^i)$ instead of the statement S_j itself.

4.3.2.1 Buildup of A Call-Tree and Duplication of Subroutines

A call tree rooted on the calling subroutine can be used as a vehicle for formulating and implementing partial in-line expansion of subroutine calls. The algorithm for deriving the call tree is shown below. The subroutine may need to be duplicated and renamed simultaneously to cope with the approximation of the global effects of subroutine calls. The subroutine duplication requires extra space, but it makes two

identical subroutine calls with different parameters possibly executable in the synchronous parallel mode, thus exploiting the parallelism. The algorithm for call tree derivation follows:

```
Input      Calling subroutine S with node set V = {S} and edge set E = ∅.
Output     Call tree represented by the node set V and edge set E.
Algorithm  procedure build_tree (S : node; V,E : set);
           examine a statement;
           while not end statement do
             if a subroutine call A then
               if A called previously then
                 duplicate A and call it A';
                 V := V ∪ {A'};
                 E := E ∪ {SA'};
                 build_tree (A',V,E);
               else
                 V := V ∪ {A};
                 E := E ∪ {SA};
                 build_tree (A,V,E);
               endif;
             endif;
             examine statement;
           endwhile;
         endprocedure;
```

4.3.2.2 Intraprocedural Constant Propagation

To improve the effect of semantics analysis techniques on the approximation of access regions of array variables, identification of variables bound with an integer constant at compile-time is a useful approach, because constant variables are possible coefficients of subscript expressions of array variables or loop and array bounds, which are directly concerned with the approximation of array regions. Wegman and Zadeck developed an intraprocedural constant propagation algorithm which produces a good result [2]. This algorithm is incorporated into our partial in-line expansion approach.

4.3.2.3 Subroutine Variables Renaming

As stated before, identification of the global variables in a called subroutine is straightforward via partial in-line expansion, because the formal parameters of the called subroutine are directly replaced by the actual variables of the calling subroutine. In practice, as the names of some local variables of the called subroutines may be the same as that of the passed global variables of the calling subroutine, identifying the global variables may be difficult. To avoid such a situation, we rename all the local variables in the called subroutine such that no local variable has the same name as that of the passing global variables. This is not hard to achieve. We simply append a special character such as '*' after every local variable of a particular subroutine to differentiate it from the variable name of other subroutines. This approach makes it very easy to identify a statement that accesses the global variables in a called subroutine.

4.3.2.4 Identification of Aliasing Relations of Common Variables

The aliasing relation of **common** variables also needs to be recognized. This recognition can be achieved by first identifying the sets of variables with identical **common** names in the scope of the calling subroutine, and then renaming the sets of variables of the same named **common** in a way such that it is easy to recognize the aliasing relation at the dependence analysis phase. The sets of variables sharing the identical named **common** region can be computed at the call-tree buildup stage. That is, during the buildup of the call-tree, the information regarding the **common** regions and the set of variables involved in each of distinct named **common** for each subrou-

tine are also collected. Those sets of variables sharing the identical named **common** are then grouped. The renaming technique is similar to that for subroutine variables stated in Section 3.2.3. For the sets of variables of identical named **common** we append two special characters after each variable. The first character represents a specific **common** region while the second character denotes a specific set. Precise analysis to ascertain two variables of the same named **common** but in different set dealing with same locations may still be required at the dependence analysis phase. Renaming the sets of variables with the identical named **common** can also be done at the call-tree buildup stage.

4.3.2.5 Computation of the Access Region of Array Variables

The scalar global variables accessed in a statement can be computed easily. However, for an array global variable in a statement, the portion of accessed elements may be very large or unknown at compile-time. These elements are hard to enumerate. Such cases are found in statements inside a Do-loop, where the subscript expressions of the array variables are functions of the Do-loop index. In such a case, the set of accessed elements must be approximated and the region accessed expressed. The more precision the approximation, the more parallelism can be exposed in the subroutines. Coarse estimation for an array variable (for example, the whole array) causes higher possibility of data dependence.

An array variable is composed of two parts: array name and subscript expression. Global variables may be found in either part. The accessed region for an array variable will be computed and enumerated no matter which part contains global variables.

To denote the accessed region of an array variable, we can use a system of linear equalities and inequalities to describe the ranges of possible values for each dimension [53].

The accessed region of an array variable A is denoted as a triple (A,d,θ_a), where d is the dimensions of A, and θ_a is a system of linear equalities and inequalities describing the ranges of possible values for each dimension of A. This representation is virtually introduced by Triolet [53]. For example, in Figure 4.3 the accessed region for array S is $(S,1,\theta_s)$,

```
S(5) = 6
Do 10 I = 1, N
Do 10 J = 1, 100
Do 10 K = I, 50
T(I) = B(I+2,J-1,K+3) + 5
10 Continue
```

Fig. 4.3 A code fragment showing accesses of array variables

where θ_s is $\phi_1 = 5$ as ϕ_1 denotes the expression of the 1st dimension of S. Similarly, the accessed region for array T is $(T,1,\theta_t)$, where θ_t is: $\phi_1 = I$, $1 \le I \le N$; the accessed region for array B is $(B,3,\theta_b)$, where θ_b is:

$$\phi_1 = I+2,\ 1 \le I \le N;\ \phi_2 = J-1,\ 1 \le J \le 100;\ \phi_3 = K+3,\ 1 \le K \le 50$$

Thus, ϕ_i denotes the expression of the i-th dimension of B.

To approximate the access region of an array variable in a general statement, we can use *syntax-directed analysis schemes.* A syntax-directed analysis scheme is a context-free grammar in which a program fragment called semantic analysis action is associated with each production (a production is a syntax rule) [1]. For example, suppose the semantic analysis action is associated with production $A \rightarrow XYZ$. In a bottom-

up parser, the action is taken when *XYZ* is reduced to *A*. Straightforwardly, some semantic analysis actions for computing the access region of an array variable are associated with the production rules of a Do-statement and the body of a Do-loop construct.

Approaches such as affine relationship among variables and a simpler regular loop have been proposed to compute the execution context of an array variable [32,53].

4.3.2.6 Algorithm for Computing the Global Effects of Subroutine Calls

The process of computing the global effects of subroutine calls for the calling subroutine is given in the following algorithm:

1. Build a call tree based on the topological subroutine call order in the scope of the calling subroutine; collect the information of **common** variables and rename the **common** variables properly.
2. Compute intraprocedural constant propagation for calling subroutine.
3. For each subtree rooted on the child of the calling subroutine do:

for each node (subroutine) following the *preorder* **do**

if there exist arguments bound with variables of the top-level subroutine **then**

1. Rename all local variables, excluding formal arguments.
2. Replace formal arguments by actual variables of the top-level calling subroutine.
3. Remove argument-list.

4. Compute intraprocedural constant propagation.

endif

endfor;

for each node (subroutine) following the *postorder* **do**

if node has been restructured **then**

1. Compute accessed elements for each statement.

2. Replace the corresponding **call** statement in the parent node by the statements with global variables.

3. Resume original subroutine code.

endif

endfor;

Figure 4.4 presents an example. In Figure 4.4(a) a fragment of a program taken from [53] is given. After computing the global effects of the called subroutines, the code is shown in Figure 4.4(b).

subroutine A (a,lda,n1,n3,b,ldb,n2,c,ldc) real a(lda,*),b(ldb,*),c(ldc,*)	subroutine A (a,lda,n1,n3,b,ldb,n2,c,ldc) real a(lda,*),b(ldb,*),c(ldc,*)
do i = 1, n3	do i = 1, n3
	$IN_g(S_1^B)$ = {n1} $OUT_g(S_2^B)$ = {(a,2,θ_a)}, θ_a = (ϕ_1 = j, 1 ≤ j ≤ n1, ϕ_2 = i)
call B (n2,a(1,i),n1,ldb,c(1,i),b)	$IN_g(S_3^B)$ = {n2} $IN_g(S_4^B)$ = {(a,2,θ_a),(c,2,θ_c),(b,2,θ_b)}, θ_a = (ϕ_1 = j, 1 ≤ j ≤ n1; ϕ_2 = i), θ_c = (ϕ_1 = k, 1 ≤ k ≤ n2; ϕ_2 = i), θ_b = (ϕ_1 = j, 1 ≤ j ≤ n1; ϕ_2 = k, 1 ≤ k ≤ n2) $OUT_g(S_4^B)$ = {(a,2,θ_a)}, θ_a = (ϕ_1 = j, 1 ≤ j ≤ n1; ϕ_2 = i)
enddo	enddo
end	end
subroutine B (n2,y,n1,ldm,x,m) real x(*),y(*),m(ldm,*)	
S_1 do j = 1, n1 S_2 y(j) = 0 S_3 do k = 1, n2 S_4 y(j) = y(j) + x(k) * m(j,k) enddo enddo	
end	
(a) The original code	(b) After computing the global effect

Fig. 4.4

4.3.3 Analysis of Data Dependence

Data dependence analysis exposes the ordering of interdependent statements between called subroutines and/or between called subroutines and the calling subroutine. The **call** statements at this phase have been replaced by the expanded statements. As data dependence refers to common access variables between two statements, the accessed variables for each statement in the calling subroutine need to be computed prior to the test of data dependence. The method of computation is similar to that stated in Section 4.3.2.5.

4.3.3.1 Method of Analysis

The sets OUT(S_j^i) and IN(S_j^i) are associated with each statement S_j of each directly or indirectly called subroutine i in the calling subroutine body. Two statements S_p^q S_r^s, where S_p^q appears earlier than S_r^s and $q \neq s$, have the following possible dependence relations:

1. S_r^s is true-dependent on S_p^q if OUT(S_p^q) $\cap$ IN(S_r^s) $\neq \varnothing$.

2. S_r^s is anti-dependent on S_p^q if IN(S_p^q) $\cap$ OUT(S_r^s) $\neq \varnothing$.

3. S_r^s is output-dependent on S_p^q if OUT(S_p^q) $\cap$ OUT(S_r^s) $\neq \varnothing$.

Therefore, to test data dependence we need to check if either common accessed scalar variables or overlapped access regions for two identical array variables exist between two statements. For an array variable A of which the subscript expression involves global variables but the array name is not a global variable, we can identify the used global variables from θ_a of (A,d,θ_a), and then test the common accessed variables.

Identification of common accessed scalar variables is trivial. The technique for examining the overlapped regions between array variables is stated below.

Two regions $r_1 = (T_1, d_1, \theta_{t_1})$ and $r_2 = (T_2, d_2, \theta_{t_2})$ cannot overlap if $T_1 \neq T_2$. If $T_1 = T_2$ then we need to determine if overlapped parts exist for θ_{t_1} and θ_{t_2}. We first explain the approach to test the overlapped parts for a single dimension array variable with the same index range and then extend the approach to compute a general case.

Consider two statements carrying the same array variable but with different linear subscript expressions respectively in a single Do-loop. To test if these two statements are interdependent due to this array variable for a given execution order, the

GCD (greatest common divisor) test and Banerjee's inequalities test [3] are available. For example in Figure 4.5 x is an array variable and f and g are arbitrary subscript expressions. S_2 is true-dependent on S_1 if there exist integers I_1 and I_2 such that $1 \leq I_1 \leq I_2 \leq$ N and f(I_1) = g(I_2). That is, x(f(I_1)) of statement S_1 is overlapped with x(g(I_2)) of statement S_2. If f and g are not linear functions of the index then it is very difficult to determine if S_2 is true-dependent on S_1. However, the determination becomes feasible when f and g are linear functions of the index, i.e.,

$$f(I) = a_0 + a_1 I \qquad g(I) = b_0 + b_1 I$$

Therefore, to test if overlapped elements exist in x, we look for a solution for the equation

$$f(I_1) - g(I_2) = 0$$

That is, check the solutions of I_1 and I_2 for the equation

$$a_1 I_1 - b_1 I_2 = b_0 - a_0$$

The algorithm for testing the solution is:

1. Determine if f and g are linear. If they are, then compute a_0, b_0, a_1, b_1.

2. If neither the GCD test nor the Banerjee's inequality holds, then there is no overlapped elements in x, i.e., no true dependence. Otherwise, assume an overlapped element exists (even though it may not) [6].

```
Do 10 I = 1, N
S1  x(f(I)) =
S2          = x(g(I))
10 Continue
```

Fig. 4.5 Statements carrying a common array variable

Therefore, for two regions r_1 and r_2 where $T_1 = T_2$, θ_{t_1} and θ_{t_2} have the same loop index range and their subscript expressions are linear functions of the index, we can test the dependence. Note, Banerjee's inequality approach is available only for statements under the same loop index range. In the case where θ_{t_1} and θ_{t_2} have different index ranges, we need to temporarily modify the index range for θ_{t_1} or θ_{t_2} prior to using the algorithm. The method is shown below.

1. Examine the index range for θ_{t_1}.
2. Examine the index range for θ_{t_2}.
3. Find the common index range of θ_{t_1} and θ_{t_2}.
4. Replace the index range for θ_{t_1} or θ_{t_2} by the common index range. (Assuming that both Do-loops are normalized.)

For a multiple dimension array variable, the testing of overlapped elements for a particular dimension under a given execution order is more complicated. In fact the existing approach is a generalized GCD and Banergee's inequality algorithm. Allen et al. provides a detailed description of the algorithm [6].

Triolet proposed another approach to compute the data dependence involving the global effects of **call** statements [53].

4.3.3.2 Scope of Analysis

Analysis of data dependences, i.e., true dependence, antidependence, and output dependence [6], are conducted for each group of subroutine calls with identical control dependence. Parallelizable subroutine(s) may be executed in parallel with the calling subroutine and/or with other called subroutines. For each group of identically

control-dependent statements involving the **call** statements, if they are not control-dependent on an iteration statement, the algorithm of analysis is presented below:

1. Identify the lexically first **call** statement, suppose it is S_f.
2. Identify the lexically last **call** statement or ordinary statement, suppose it is S_l.
3. Starting from the first expanded statement of the first **call** statement to the last ordinary statement, or the last expanded statement of the last **call**, inclusively, identify the scopes of the particular group of identically control-dependent subroutines. Each scope covers the maximum number of lexically successive and identically control-dependent statements which involve at least one subroutine call statement. Several disconnected scopes may exist between S_f and S_l.
4. In each scope, determine the interprocedural loop-independent data dependence between called subroutines, and/or calling- and called-subroutines.

In case the subroutines are identically control dependent on an iteration statement, loop-independent and loop-carried dependence analysis needs to be done for all the statements in the scope of the conditional statement. The method is similar to that mentioned before. The dependence distance is measured to determine if the dependence is loop-independent or loop-carried.

Antidependence and output dependence between statements can be eliminated by using extra variables. The recurrence relation of dependence in Do-loops may also be removed [16]. In general, the level of parallelism exposure depends on the ability of the existing dependence analysis techniques. If a dependence relation cannot be determined due to insufficient information or techniques, assume the dependence exists.

4.3.4 Identification of Execution Mode

The execution mode for subroutines is identified on the basis of each particular scope of identically control-dependent subroutines. Note, the task which spawns the subtasks to execute the subroutines will begin to execute the statement right after the last statement of the particular scope only when all subtasks have finished their jobs.

Conditional statements can be further classified as repetitive and non-repetitive conditional statements. For example, a do-statement is a repetitive conditional statement while an if-statement is a non-repetitive conditional statement. Methods of determining the execution mode of subroutines vary with the nature of the conditional statement on which the subroutines depend.

For the subroutine(s) which is free of control dependence (i.e. dependent on the dummy **entry** statement) or control-dependent on non-repetitive conditional statement, the identification of the execution mode is straightforward. For example, if no data dependence exists between two subroutines, these two subroutines can be executed in asynchronous parallel. Similarly, if no data dependence exists between the statements in a called subroutine and in the calling subroutine, the calling subroutine and the called subroutine can be executed in asynchronous mode. When data dependences occur between two called subroutines, these two subroutines can be executed either in synchronous parallel or in serial, depending on the weight between the parallelism benefit and the synchronization cost. This relation between data dependences and execution modes is also true for the case between the calling subroutine and the called subroutine.

For those subroutines which are control-dependent on a repetitively conditional statement, determination of the execution mode is different. Dependence relations between statements in the scope of a repetitively conditional construct can be further classified as either loop-independent or loop-carried. The execution modes of subroutines are determined not only by the data dependence but also by the execution mode of the construct of the conditional statement. Consider a Do-loop construct. If the Do-loop itself is executed in sequential mode then loop-carried dependences among subroutines in the loop body can be ignored. Only loop-independent dependence is relevant for the determination of execution mode. However, if the Do-loop itself is executed in parallel, then each iteration will be executed by a parallel task created by the main task. If the subroutine(s) in the loop is executed by each of the assigned parallel task for a specific index value, then the subroutine(s) can be thought as an ordinary statement. In other words, loop-independent dependences can be ignored, and only loop-carried dependences are considered. But, if each parallel task again creates subtasks to execute the subroutines, then both loop-carried and loop-independent dependences affect the determination of execution modes. Based on these observations, an approach to identify the execution modes of subroutines follows:

1. If the construct of the conditional statement is executed in serial by the original task, the method of detecting parallelizable subroutines is straightforward.

2. If the construct of the conditional statement is executed in parallel, then each iteration is executed by a parallel task created by the main task. The execution mode of the subroutine can be identified as follows:

```
if the parallel tasks can spawn subtasks to execute subroutines then
  if no loop-independent dependence exists between subroutines then
    the subroutines are executed in asynchronous mode in each iteration
  else
    the subroutines are executed in synchronous mode in each iteration
  endif
  if no loop-carried dependence exists between subroutines then
    the subroutines are executed in asynchronous mode between iterations
  else
    the subroutines are executed in synchronous mode between iterations
  endif
else
  if no loop-carried dependence exists between subroutines then
    the subroutines are executed in asynchronous mode between iterations
  else
    the subroutines are executed in synchronous mode between iterations
  endif
endif
```

In general, the deterministic repetitive construct such as the Do-loop is executed in parallel, while the nondeterministic repetitive construct is executed either in serial or in parallel.

4.3.5 Restructuring of Calling and Called Subroutines

The restructuring of calling and called subroutines is by no means trivial. It includes the incorporation of additional statements or library calls to manage parallel tasks and synchronization operations, such that the desired parallelization of subroutines can be achieved correctly and efficiently. The restructuring techniques which regard the creation and termination of the parallel tasks and synchronization primitives are machine-dependent and are not addressed in this chapter. We are concerned with the placement of synchronization primitives for synchronous subroutines. This placement includes identification of synchronization points, passing of synchronization primitives to the called subroutines, and elimination of redundant synchronization. In this chapter we only discuss the identification of synchronization points.

Identification of Synchronization Points

The synchronization points for a pair of interdependent but parallel-processed statements which are under identical control dependence are simple to locate. A **send** operator is placed immediately following the source statement of the dependence; while the synchronization operation **wait** is placed immediately before the sink statement. However, for interprocedural dependence, this placement scheme of synchronization primitives may not be correct if the *source* statement of the dependence is control-dependent on other statement(s). Since the execution of the source statement will depend on the results of its control-dependent statement(s), it is possible to skip the source statement during synchronous execution, causing the task executing the sink statement to be in a waiting state forever. On the contrary, if the sink statement of

the dependence is control-dependent on other statement(s), a run-time error may not exist. The skip of the sink statement may not result in execution failure. Thus, it is necessary to identify the control dependence relation for the source statements of interprocedural dependence.

In the parallel model, the source statements of any synchronous subroutines (between a calling and a called subroutine, or between two called subroutines) are always in the *called* subroutines. To identify the control dependence relations, it is necessary to compute the control dependence for the subroutine involving the source statements. The method is similar to that stated in Section 3.1.

If a source statement is control-dependent on a dummy **entry** statement, i.e., free of any real control dependence, the location of synchronization point is stated before. Otherwise, the synchronization primitive should be placed just before a statement S_p with the following features:

1. the statement which controls the source statement is post-dominated by S_p,
2. S_p is free of any real control dependence in the subroutine, and
3. S_p is lexically the first statement which meets the above two conditions.

For example, if the statement S_5 in Figure 4.2 is the source statement of an interprocedural dependence, the synchronization primitive **send** should be placed just before the statement S_9.

Statement S_p can be derived from the control dependence relation of the subroutine and the post-dominator tree derived from the control flow graph. Detailed analysis may be required if the source statement is in the sub-subroutine directly or indirectly called by the called subroutine. Clearly, there exists one and only one S_p,

and the synchronization primitive **send** is executed at the earliest time whether the source statement is skipped or not.

4.4 An Algorithm for Subroutine Parallelization of A Program

Section 3 presents the principle of parallelizing the called subroutines for a general calling subroutine. To parallelize the subroutines of a general program, we need to apply the principle orderly to all of the existing calling subroutines. We thus first need to develop an approach to find in order all the existing calling subroutines in a program.

4.4.1 An Approach to Search The Calling Subroutines

A main program itself can be thought as a calling subroutine of level 0. Any subroutine called by the program is a calling subroutine of level 1. The subroutines called by the subroutine of level 1 are the calling subroutines of level 2. Similarly, calling subroutines of levels 3, 4, ... , can exist. Beginning with the main program, the relations of the calling and called subroutines of a program are represented by a call tree. That is, the calling subroutine of level 0 is the root and the tree is constructed based on the relation that the children of each node are the subroutines called by the node. Each node, except the root and external node, is itself a calling and called subroutine. If a subroutine is called more than once at different sites, it needs to be replicated and given a new name. Finally the call tree has a node set consisting of distinct nodes. The algorithm for computing the call tree is given in Section 3.2.1.

4.4.2 The Algorithm

After the call tree is built, the calling subroutines of the program are known. We thus can employ the principle of subroutine parallelization in Section 3 to each of the calling subroutine in order, beginning with the subroutine of level 0. The algorithm for parallelizing subroutines of a program is as follows:

Input The calling tree (root S, node set V and edge set E) of a program and $E' = \{SK \mid SK \in E\}$

Output A subroutine-parallelized program.

Algorithm

```
procedure parallel_routine (S : node; E' : set);
if E' ≠ ϕ then
   Compute control dependence of S;
   Compute global effects of the called subroutines in S;
   Analyze data dependence;
   Identify execution mode for each subroutine;
   Restructure S and the called subroutines in S;
   while E' ≠ ϕ do
      Get an edge SA ∈ E';
      F := {AK| AK ∈ E};
      E' := E' - {SA};
      parallel_routine (A,F);
   endwhile;
  endif;
endprocedure;
```

There may exist some subroutines which have the same or very similar (except the synchronization primitives) routine body but different routine names. A subroutine optimization phase to merge these subroutines to one subroutine may be needed.

4.5 Summary of Results

To speedup the parallel processing of a program, the utilization of multitasking facilities of a shared-memory multiprocessor should be maximized. One way to provide user access to the facilities is to use the parallel call statement. Thus efficient parallelization of subroutine calls in a program is required.

A called subroutine in a calling subroutine has its optimum execution mode - serial, synchronous parallel, asynchronous parallel, depending on the interprocedural and intraprocedural data dependence at the called site. Five major steps are required to parallelize the subroutines of a calling subroutine: (1) computation of control dependence, (2) approximation of the global effects of called subroutines, (3) analysis of data dependence, (4) identification of execution mode, and (5) restructuring of calling and called subroutines.

To parallelize the subroutines of a program, the calling subroutines of all levels need to be recognized. A program itself is in fact a calling subroutine of level 0. The called subroutines by a calling subroutine of level n are of level $n + 1$. Based on this concept, an algorithm for building a call tree exposing the relations of calling and called subroutines in a program is developed. A subroutine which is called twice must be duplicated and given a new name. This approach makes the call tree consist of only distinct nodes.

With the information in the call tree and the five steps of subroutine parallelization, an algorithm for subroutine calls parallelization in a program is proposed. A subroutine optimization phase is needed to reduce redundant subroutines.

Chapter 5

Summary

Using vector- and multi-processor systems either to solve new large-scale scientific and engineering problems, or to speed up the processing of existing serial code is no doubt the trend of the future. To accelerate the movement of parallel processing into the mainstream of computation, computer scientists need to expand research in many fields, including parallel machines, parallel algorithms, parallel languages, parallel compilers and automatic tools.

The goal of this dissertation has been to develop strategies to several concurrency issues, by which the capabilities of parallelism exploitation of parallel compilers or source-to-source translators can be enhanced. The issues include recurrence resolution for vector processing, dependence cycle statement ordering for parallel loops, and subroutine parallelization for general programs.

5.1 Summary of Results

Recurrences are a type of data dependence relation existing in statement(s) of a loop. In general, the statements involved in a recurrence are unvectorizable. To vectorize the statements in a recurrence, the recurrence needs to be broken first; the resulting code is then reordered such that no backward dependences exist in the statements. A simple multi-statement recurrence with an essential link of antidependence, or output-dependence, or anti- and output-dependence is breakable via a *node splitting* or a *sink variable renaming* technique. The node splitting algorithm, which was submitted in [37] to break an antidependence link, is observed to be available only when

the source of antidependence itself is not the sink of another true dependence. Otherwise, a modified node splitting algorithm must be substituted. To break an output-dependence or an anti- and output-dependence link, a sink variable renaming scheme is effective. This scheme is also valid for breaking an antidependence link. In practice, node splitting and sink variable renaming techniques are suggested to be employed in a complementary manner to break an antidependence link. A recurrence which is formed only by links of true-dependence, or true- and any other (anti-, output-) dependence is unbreakable. A loop with a loop body that is an unbreakable recurrence can only be partially vectorized. Based on a static dependence concept, a general but less efficient partial-vectorization algorithm is observed. In contrast, an alternative partial-vectorization technique, which is derived on basis of a dynamic dependence concept, is more efficient but less general. By integrating these link-breaking strategies and other existing related techniques, an algorithm to resolve a general recurrence in vector processing has successfully been developed.

The approach of parallelism exploitation for the loop executed in vector mode and the loop executed in parallel mode is different. For a parallel loop executed on shared memory multiprocessor system, one way to exploit the parallelism of the statements in an unbreakable dependence cycle is to properly reorder the statements, such that minimal synchronization delay will result. To reorder the statements in a general dependence cycle, a timing formula used for estimating the parallel-execution time of the loop at compile-time is useful. Cytron's formula [21] is available to some extent for this objective, except that it cannot fit some dependence relations accurately. A modified more accurate timing formula for computing the parallel-execution time,

assuming that each statement in the loop will take equal execution time, is derived. Based on the modified formula, a near-optimal algorithm to align the statements is proposed. Meanwhile, a local statement reordering algorithm to tune the order of statements locally is also developed. The statement ordering algorithm is further extended to deal with a simple global dependence cycle in multiple nested loops.

Like the parallelization/vectorization of iterative loops, subroutine parallelization is also a complicated issue. It includes many necessary but time-consuming computations. Both asynchronous and synchronous parallelization of subroutines is required to enhance the performance of a parallel processed program. To acquire accurate results in implementation, many miscellaneous issues such as identifying subroutines of identical control, locating the statements in subroutines which cause global effects during subroutine calls, testing interprocedural dependence, placing synchronization primitives, and restructuring calling and called subroutines need to be considered carefully. A systematic approach to parallelize the called subroutines for a calling subroutine is derived: (1) computation of control dependence, (2) approximation of the global effects of subroutines, (3) analysis of data dependence, (4) identification of execution mode, and (5) restructuring of calling and called subroutines. Continuous employment of these five steps to different levels of calling subroutines in a program addresses the parallelization of subroutines. An algorithm based on this concept is proposed.

5.2 Significance of Results

The significance of the results stated in section 5.1 is outlined as follows.

- Dependence recurrences occur in any repetitive language construct, either in a deterministic loop or in a nondeterministic loop. The vectorization of the statements involved in a recurrence is totally inhibited while the level of parallelization of those statements is reduced. The proposed algorithm for resolving recurrences improves the performance of the loops in vector- or parallel-processing.
- Recurrences are classified as linear recurrence, boolean recurrence, and non-linear recurrence. Non-linear recurrence is the topic of our research. The results obtained are useful for integration with the linear and boolean recurrences resolution strategies [37] to develop a further generalized recurrence resolution system.
- The proposed dynamic dependence concept offers an innovated approach to exploit the parallelism in vector processing.
- The proposed sink variable renaming technique is effective in terms of breaking an output dependence or anti- and output-dependence link. The power of the sink variable renaming is also reflected on the fact that it can be used for breaking an antidependence link.
- The observed limitation in application and the associated revised version of the node splitting algorithm make the node splitting algorithm more reliable.
- The proposed revised timing formula in Chapter 3 provides a more accurate approach to measure the execution time (or parallelism level) of parallel loops at compile-time. The accuracy in measuring the execution time favors the derivation of an optimum alignment of statements in a parallel loop and the

decision of a correct scheduling scheme. The performance of parallel loops is thus improved.

- The dependence cycle statement ordering algorithms provide theoretical approaches to order the statements.
- The proposed subroutine parallelization algorithm offers an approach to parallelize the subroutine, in which the subroutines can be executed in asynchronous or synchronous manner. This theoretical approach provides a basis for the generation of more efficient parallelized code.

5.3 Comparison with Related Research

5.3.1 Issue for Recurrences Resolution

Recurrences are abstracted at the loop distribution phase. There are three types of recurrence: linear recurrences, boolean recurrences, and nonlinear recurrences. Linear recurrences can be speeded up but are still slower than vector operations [35]. Typically, linear recurrence is processed via a library call to a linear recurrence solver [17,37]. Boolean recurrences can be substantially speeded up [10]. No data regarding whether nonlinear recurrences can be speeded up is reported in literature.

Recurrences connected by only true dependences were proposed to be partially vectorized via thresholding [6]. Thresholding, a technique which refers to abstracting dependence-free (and thus vectorizable) instances of statements and transforming them to be the inner loop of a two-layer nested loop, is virtually based on static dependence concept. This approach is somewhat conservative and does not expose the potential parallelism. The dynamic partial-vectorization algorithm proposed in

Chapter 2 overpasses the thresholding technique in that it fully exploits the potential parallelism.

Recurrences connected by at least one essential antidependence link were suggested to be broken via node splitting [6,37,45]. But the dependence relations suitable for applying node splitting have not been established. In Chapter 2 such dependence relations are introduced. In addition, a sink variable renaming technique, which is proposed to break a recurrence connected by at least one anti- and/or output-dependence link, is also newly developed.

5.3.2 Issue for Dependence Cycle Statement Ordering

Other strategies on exploiting the parallelism that exists in dependence cycles of parallel loops are program partitioning and cycle shrinking [43,45]. Program partitioning is a method to partition a loop into many independent tasks such that no synchronization is required for concurrent execution. Proposed techniques to partition a loop include GCD (greatest common divisor) and minimum dependence distance in dependence cycle [42,43]. The cycle shrinking strategy deals with the extraction of dependence-free portions in the dependence cycle, which are then partially parallelized via barrier synchronization mode. In general, for program partitioning, the parallelism level exposed is smaller but there is no synchronization cost. It has been proven that cycle shrinking is always superior to program partitioning (at least theoretically) in terms of parallelism exploitation [45].

In implementation, the cycle shrinking strategy is a barrier synchronization scheme. This scheme allows only lexically forward dependencies. All iterations of the

inner parallel loop must complete execution before the next iteration of the outer serial loop is started. Compared with statement reordering, the synchronization strategy for cycle shrinking is more structured but more restricted [56]. This implies the parallelism exposed in cycle shrinking is less than that of statement ordering. But the synchronization strategy for the statement ordering scheme is harder to manage.

Cytron developed a timing formula which can be used indirectly to measure the parallelism of a parallel loop [21]. This timing formula cannot fit some dependence relations precisely. The proposed modified timing formula in Chapter 3 measures the parallelism more precisely. This measure defines the optimization problem of rearranging statements to minimize synchronization delay. To find an optimum alignment, in [20] a general and exhaustive branch and bound algorithm was developed; in Chapter 3 a heuristic including a branch and bound algorithm is proposed. Theoretically, the latter is superior to the former in terms of the efficiency of the algorithm itself.

5.3.3 Issue for Subroutine Parallelization

Most research in subroutine parallelization deals with the theory and implementation of asynchronous parallel subroutines [2,12,13,53]. The subroutine parallelization scheme proposed in Chapter 4 involves asynchronous and synchronous parallel subroutines. The difference of parallelization schemes makes the accompanied algorithms differ significantly. For example, the *forward analysis, backward analysis, alias analysis, computing convex hull* and *interprocedural dependence analysis* for asynchronous subroutine scheme are replaced by *partial in-line expansion, subroutine*

variables renaming and *approximating the global effect of subroutine* for asynchronous and synchronous subroutine scheme. Some other analysis phase techniques such as *identification of execution mode* and *restructuring calling and called subroutines* are not required for asynchronous subroutine scheme. In addition, the technique of data dependence analysis proposed in Chapter 4 is more straightforward but may not be more efficient in general.

Each of these two subroutine parallelization schemes has its own benefit and cost in implementation. Detailed comparison between them is the future research topic. Theoretically, the associated subroutine parallelization scheme in Chapter 4 provides a sound basis for further parallelism exploitation for multiprocessing.

5.4 Future Research

For recurrence resolution in vector processing, some further research are required. The research includes:

- Generalize the partial vectorization algorithm that is derived on dynamic dependence concept.
- Examine the efficiency of the general recurrence resolution algorithm itself in detail. Modify the algorithm to improve its efficiency.
- Develop an approach to predict the necessity of recurrence resolution for a particular loop. This includes the development of a model with parameters such as loop-index length, break-even length, and cost of each vector instruction to evaluate the possible speedup after the resolution of recurrences of the code.

- Integrate the techniques for handling linear recurrence, boolean recurrence, and non-linear recurrence.

The dependence cycle statement ordering for efficient concurrent execution of loops is an inherently difficult issue. Further studies on this issue include:

- Study and modify the proposed statement ordering algorithm based on the actual operation of the algorithm by removing some assumptions, such as that each statement in the dependence cycle takes equal execution time.
- Study the optimum data structure used in the statement ordering algorithm such that the performance of the algorithm can be improved.
- Find an alternative heuristic approach to properly order the statements in a dependence cycle.

The subroutine parallelization is an important issue in terms of enhancing the speedup in multiprocessing. Further research include:

- Compare the efficiency of implementation in detail between the proposed subroutine parallelization algorithm and other related schemes.
- Improve the efficiency of implementation for each step of the proposed subroutine parallelization algorithm.
- Characterize a program's adaptability for subroutine parallelization.
- Study the subroutine parallelization and program adaptability for subroutine parallelization for multiprocessor message-passing systems.

Bibliography

[1] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman, Compilers : Principle, Techniques, and Tools, Addison-Wesley Publishing Company, ISBN 0-201-10088-6, 1986.

[2] F. Allen, M. Burke, P. Charles, R. Cytron, and J. Ferrante, "An Overview of the PTRAN Analysis System for Multiprocessing," Proceedings of the 1987 International Conference on Supercomputing, June 1987, pp. 194-211.

[3] John R. Allen and Ken Kennedy, "A Parallel Programming Environment," IEEE Software, 2(4):22-29, July 1985.

[4] J. R. Allen and Ken Kennedy, "PFC: A Program to Convert Fortran to Parallel Form," Technical Report MASC-TR 82-6, Rice University, Houston, Texas, March 1982.

[5] R. Allen, D. Baumgartner, K.Kennedy, and A. Porterfield, "Ptool: A Semi-automatic Parallel Programming Assistant," Proceedings of the 1986 International Conference on Parallel Processing, IEEE Computer Society Press, August 1986.

[6] Randy Allen and Ken Kennedy, "Automatic Translation of Fortran Program to Vector Form," ACM Trans. on Programming Languages and Systems, Vol. 9, No. 4, October 1987, pp. 491-542.

[7] Bill Appelbe, Kevin Smith and Charlie McDowell, "Start/Pat: A Parallel-Programming Toolkit," IEEE Software, July 1989, pp. 29-38.

[8] J. L. Baer, "Theoretical Aspects of Multiprocessing," Computer Survey, Vol. 5, March 1973, pp. 31-80.

[9] U. Banerjee, "Data dependence in ordinary programs," Department of Computer Science, University of Illinois at Urbana-Champaign, Rep. 76-837, Nov. 1976

[10] U. Banerjee, D. Gajski, and D. J. Kuck, "Array Machine Control Units for Loops Containing IFs," Proceedings of the 1980 International Conference on Parallel Processing, Harbor Springs, MI, August 1980, pp. 28-36.

[11] A. J. Bernstein, "Analysis of Programs for Parallel Processing," IEEE Trans. on Electronic Computers, Vol. EC-15, No. 5, October, 1966, pp. 757-763.

[12] Michael Burke and Ron Cytron, "Interprocedural Dependence Analysis and Parallelization," SIGPLAN Notices 21, 7 (July 1986), pp. 162-175.

[13] David Callahan and Ken Kennedy, "Analysis of Interprocedual Side Effects in a Parallel Programming Environment," Proceedings of the 1986 International Conference on Supercomputing, June 1987, pp. 138-171.

[14] Hai-Bo Chen and Yun-Gui Ci, "Parallel Execution of Non-Do Loops," Poceedings of the 1987 International Conference on Parallel Processing, pp. 512-516.

[15] Hui Cheng, "Vector Pipelining, Chaining, and Speed on the IBM 3090 and Cray X-MP," IEEE Computer, September 1989, pp. 31-46.

[16] Chih-Ping Chu and Doris L. Carver, "An Analysis of Recurrence Relations in Fortran Do-loops for Vector Processing," Proceedings of 5th International Parallel Processing Symposium, Anaheim, California, April 30 - May 2, 1991, pp.

619-625.

[17] Howard B. Coleman, "The Vectorizing Compiler for the Unisys Isp," Proceedings of the 1987 International Conference on Supercomputing, June 1987, pp. 567-576.

[18] P. Cousot, N. Halbwacks, "Automatic Discovery of Linear Restrain among Variables of a Program," Conference Record of the Fifth ACM Symposium on the Principles of Programming Languages, 1978, pp. 84-96.

[19] Wayne R. Cowell and Christopher P. Thompson, "Tools to Aid in Discovering Parallelism and Localizing Arithmetic in Fortran Programs," Software - Practice and Experience, Vol. 20(1), January 1990, pp. 25-47.

[20] R. Cytron, "Compile-Time Scheduling and Optimization for Asynchronous Machines," University of Illinois at Urbana-Champaign, Ph.D. Thesis, October 1984.

[21] R. Cytron, "Doacross: Beyond Vectorization for Multiprocessors," Proceedings of the 1986 International Conference for Parallel Processing (August, 1986), pp. 836-844.

[22] Ron Cytron and Jeanne Ferrante, "The Value of Renaming for Parallelism Detection and Storage Allocation," Proceedings of the 1987 International Conference on Parallel Processing, August 1987, pp. 19-27.

[23] Ron Cytron, Michael Hind and Wilson Hsieh, "Automatic Generation of DAG Parallelism," Proceedings of the SIGPLAN '89 Conference on Programming

Language Design and Implentation, Portland, Oregon, June 21-23, 1989, pp. 54-68.

[24] James Davis, Christopher Huson, Thomas Macke, Bruce Leasure and Michael Wolfe, "The KAP/S-1: An Advanced Source-to-Source Vectorizer for the S-1 Mark IIa Supercomputer," Proceedings of the 1986 International Conference on Parallel Processing, August 19-22, 1986, pp. 833-835.

[25] J. Ferrante, K. Ottenstein, and J. Warren, "The Program Dependence Graph and Its Use in Optimization," ACM Trans. on Programming Languages and Systems, July 1987, pp. 319-349.

[26] Christopher Huson, Thomas Macke, James Davies, Michael Wolfe and Bruce Leasure, "The KAP/205: An Advanced Source-to-Source Vectorizer for the Cyber 205 Supercomputer," Proceedings of the 1986 International Conference on Parallel Processing, August 19-22, 1986, pp. 827-832.

[27] Kai Hwang and Faye A. Briggs, Computer Architecture and Parallel Processing, McGraw-Hill Book Company. ISBN 0-07-031556-6, 1984, pp. 1-50.

[28] Kai Hwang and Douglas Degroot, Parallel Processing for Supercomputers and Artificial Intelligence, Mcgraw-Hill Book Company, 1989.

[29] IBM. Parallel Fortran Language and Library Reference, International Business Machines, Publication No. SC23-0431-0.

[30] Alan H. Karp, "Programming for Parallelism," Computer, May 1987, pp. 43-57.

[31] Alan H. Karp and Robert G. Babb II, "A Comparison of 12 Parallel Fortran

Dialects," IEEE Software, September 1988, pp.52-66.

[32] M. Karr, "Affine Relationships among Variables of a Program," Acta Informatica, No. 6, 1976, pp. 133-151.

[33] G. Killdal, "A Unified Approach to Global Program Optimization," Proceedings of the 1st POPL, 1973.

[34] D. E. Knuth, The Art of Computer Programming, Vol. 1: Fundamental Algorithms, Addison-Wesley, Reading, Mass., 1973.

[35] D. J. Kuck, "The Structure of Computer and Computations," Vol. 1, Wiley, New York, 1978.

[36] D. J. Kuck, R. H. Kuhn, B. Leasure and M. Wolfe, "The Structure of An Advanced Vectorize for Pipelined Processors," Proceedings of CompSAC 80 (Fourth International Computer Software and Applications Conference), October 1980, pp. 709-715.

[37] D. J. Kuck, R. H. Kuhn, D. A. Padua, B. Leasure, and M. Wolfe, "Dependence Graphs and Compiler Optimizations," Conference Record of 8th ACM Symposium on Princ. of Prog. Lang., 1981, pp. 207-218.

[38] Z. Li and W. Abu-Sufah, "On Reducing Data Synchronization in Multiprocessed Loops," IEEE Trans. on Computers, Vol. C-36, No. 1, January 1987, pp. 105-109.

[39] Z. Li, P.-C. Yew, and C.-Q. Zhu, "An efficient data dependence analysis for parallelizing compilers," IEEE Trans. on Parallel and Distributed System, Vol. 1,

No. 1, pp. 26-34, Jan. 1990.

[40] A. Lichnewsky and F. Thomasset, "Introducing symbolic problem solving techniques in the dependence testing phase of a vectorizer," Proceedings of 1988 International Conference on Supercomputing, July, 1988.

[41] Samuel P. Midkiff and David A. Padua, "Compiler Generated Synchronization for Do Loops," Proceedings of the 1986 International Conference on Parallel Processing (August, 1986), pp. 544-551.

[42] D. A. Padua, "Multiprocessors: Discussion of Some Theoretical and Practical Problems," University of Illinois at Urbana-Champaign, Ph.D. Thesis, November 1979.

[43] J. Peir, "Program Partitioning and Synchronization on Multiprocessor Systems," University of Illinois at Urbana-Champaign, Ph.D. thesis, March 1986.

[44] Constantine D. Polychronopoulos and David J. Kuck, "Guided Self-Scheduling Scheme for Parallel Supercomputers," IEEE Trans. on Computers, Vol. C-36, No. 12, December 1987, pp. 1425-1439.

[45] Constantine D. Polychronopoulos, "Compiler Optimizations for Enhancing Parallelism and Their Impact on Architecture Design," IEEE Trans. on Computers, Vol. 37, No. 8, August 1988, pp. 991-1004.

[46] Constantine Polychronopoulos, Milind Girkar, Mohammad Reza, Haghighat, Chia Ling Lee, Bruce Leung and Dale Schouten, "Parafrase-2: A New Generation Parallelizing Compiler," Proceedings of 1989 International Conference on

Parallel Processing.

[47] Michael J. Quinn, Designing Efficient Algorithms for Parallel Computers, McGraw-Hill Book Company, ISBN 0-07-051071-7, 1987.

[48] Edward M. Reingold, Jurg Nievergelt, and Narsingh Deo, Combinatorial Algorithms: Theory and Practice, Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1977.

[49] Zhiyu Shen, Zhiyuan Li, and Pen-Chung Yew, "An Empirical Study of Fortran Programs for Parallelizing Compilers," IEEE Transactions on Parallel and Distributed Systems, Vol. 1, No. 3, July 1990, pp. 356-364.

[50] K. Smith, "Pat: An Interactive Fortran Parallelizing Assistant Tool," Doctoral Dissertation, Georgia Institute of Technology, Atlanta, 1988.

[51] Peiyi Tang, "Self-Scheduling, Data Synchronization and Program Transformation for Multiprocessor Systems," University of Illinois at Urbana-Champaign, Ph.D. Thesis, CSRD Rpt. No. 809, January 1989.

[52] R. Tarjan, "Depth First Search and Linear Graph Algorithms," SIAM J. on Computing, Vol. 1, No. 2, June 1972, pp. 146-160.

[53] R. Triolet, "Direct Parallelization of Call Statement," SIGPLAN Notices 21, 7 (July 1986), pp. 176-185.

[54] S. G. Tucker, "The IBM 3090 System: An Overview," IBM System Journal, Vol 25, No. 1, 1986, pp. 4-18.

[55] Augustus K. Uht, "Incremental Performance Contributions of Hardware Concurrency Extraction Techniques," Proceedings of the 1987 International Conference on Supercomputing, June 1987, pp. 355-376.

[56] Michael Wolfe, "Multiprocessor Synchronization for Concurrent Loops," IEEE Software, 5:34-42, January 1988.

[57] Pen-Chung Yew, Lecture Note at Chin Hwa University, Taiwan, R.O.C., 1988

Vita

Chih-Ping Chu was born in Shanghai, China, on January 10, 1949. He received his B.S. degree in Agricultural Chemistry in 1971 from National Chung Hsing University, Taiwan, Republic of China. Prior to enrolling in Louisiana State University in August of 1987, he did graduate work at University of California at Riverside, where he received his Master's degree in Computer Science in the Spring of 1987. Since January of 1989 he has been a teaching assistant in the Computer Science Department. His current research interests include parallel compilers, software engineering, operating systems, image processing, and neural networks.

After graduation, he will join the Institute of Information Engineering at National Cheng Kung University, Taiwan, Republic of China, as an associate professor.

DOCTORAL EXAMINATION AND DISSERTATION REPORT

Candidate: Chih-Ping Chu

Major Field: Computer Science

Title of Dissertation: A Theoretical Approach Involving Recurrence Resolution, Dependence Cycle Statement Ordering and Subroutine Transformation for the Exploitation of Parallelism in Sequential Code

Approved:

Major Professor and Chairman

Dean of the Graduate School

EXAMINING COMMITTEE:

Date of Examination:

7/9/91