# COMPILER ANALYSIS OF SPARSE AND IRREGULAR COMPUTATIONS

BY

YUAN LIN

B.S., Fudan University, 1991
M.S., Fudan University, 1994

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2000

Urbana, Illinois

COMPILER ANALYSIS OF SPARSE AND IRREGULAR COMPUTATIONS

Yuan Lin, Ph.D.
Department of Computer Science
University of Illinois at Urbana-Champaign, 2000
David A. Padua, Advisor

Sparse and irregular computations pose some of the most interesting and challenging problems in compiler analysis. Such computations often use complicated data structures with irregular data access patterns that cannot be handled by traditional compiler techniques. This complicates many steps of an optimizing compiler, from parallelism detection and code transformation to data distribution and organization. On the other hand, as computer architecutures continue to incorporate more new features, compiler optimizations have come to play a vital role in the delivery of high performance. In this dissertation, we present compiler analysis techniques for sparse and irregular computations.

To understand what kinds of compiler techniques can significantly improve the effectiveness of an optimizing compiler, we studied this problem in the context of automatic parallelization of Fortran programs. By studying a collection of Fortran 77 programs with sparse and irregular data access patterns, we have identified several important problems that must be tackled. The compiler analysis of irregular array accesses and the parallelization of irregular reduction loops are the two most important ones.

We have developed compiler analysis techniques for the two most common irregular array accesses: single-indexed array accesses and simple indirect array accesses. For single-indexed array accesses, we use a *bounded depth-first search* method to identify the pattern the index variable follows as its value changes bewteen array accesses. For simple indirect array accesses, we developed a *demand-driven interprocedural array property analysis* technique. We can use this technique to find the property an index array has and use the property information in the analysis of its host array. Our experiments have demonstrated that a parallelizing compiler incorporated with these techniques can detect more inherently parallel loops than traditional compilers.

We have studied five different possible parallelization methods for irregular reduction loops, all of which can be applied automatically by a compiler. We compared their ease of use, applicability,

supporting compiler techniques required, run-time resource requirement, and, most importantly, run-time performance. From our analysis and experiments, we developed a general guideline for choosing an efficient parallel irregular reduction method for given programs and input data sets.

To Yuanling, Xuhong and Zhuozhuo.

# Acknowledgments

I feel very fortunate to have had Professor David Padua as my advisor. He has been very supportive and encouraging throughout my time at UIUC. His contagious cheerfulness and confidence in my abilities have always kept my spirits high. His insight has always kept me on the right track. The completion of this thesis would have not been possible without his guidance.

I also want to express my gratitude to the members of my thesis committee, Professors Michael Heath, Wen-mei Hwu, and Laxmikant Kale. They have provided me with many valuable insights and thought-provoking questions.

My thanks and admiration also go to the many past and present members of the Polaris group at Illinois, with whom I have been privileged to work. They include Rudi Eigenman, Jay Hoeflinger, Lawrence Rauchwerger, Bill Pottenger, Yun Paek, Jaejin Lee, Calin Cascaval, Peng Wu and Gheorghe Almasi. Special thanks to Professor E. Zapata, R. Asenjo, and E. Gutierrez from the University of Malaga for their support in the early stages of my research. Thanks to H. Han and Professor C. W. Tseng from the University of Maryland and Professor D. O'Hallaron from Carnegie Mellon University for providing me with the programs and data for my experiments.

Also, I would like to thank Sheila Clark for proofreading this thesis and many other papers, and for her excellent administrative support that made my life as a student a lot easier.

Finally, I would like to thank my family for their support and love in all my endeavors. To my parents who tolerated, encouraged, and supported me while I pursued my dream. To my wife, Zhuozhuo, who has always been the bright spot in my life. I cannot think of the words to express how much she means to me. It is a pleasure for me to dedicate this work to them, for it owes them so much.

# Table of Contents

# List of Tables

# List of Figures

# Chapter 1

# INTRODUCTION

## 1.1 Sparse and Irregular Computation

### 1.1.1 Sparse Matrix Computation

Most scientific computing problems represent a physical system in the form of sparse matrices, which are matrices whose majority elements are zero. For example, in many computation problems, the continuous domain of the physical system being modeled is discretized by imposing a grid or a mesh over the domain. Most computation problems simulate the physical changes at each grid point based on the influence of the neighboring elements. The simulation is often equivalent to solving sets of linear equations whose variables are associated with the grid points. A system of $n$ linear equations can be represented in a matrix of the form $Ax = b$, where $A$ is an $n \times n$ coefficient matrix, $b$ is a known vector, and $x$ is the unknown solution vector to be computed. Because the value of a variable depends on only a few other variables that correspond to its neighboring grid points, most of the coefficients in the system of equations are zero; hence the coefficient matrices are sparse matrices.

The storage requirements and computation time of applications dealing with sparse matrices can be reduced significantly by taking advantage of the presence of many zero elements in the sparse matrices. Only nonzero elements of a sparse matrix are stored explicitly; only nonzero elements are operated upon; and, additions or multiplications with zero elements are avoided. The amount of storage and the number of arithmetic operations are then proportional to the number of nonzero elements rather than to the number of elements in the matrix. This improvement does not, however, come without a cost. The cost is an increased complexity in developing sparse

computation programs.

### 1.1.2 Why Sparse Matrix Computations Are More Complicated Than Dense Matrix Computations

Designing and implementing a sparse computation program is complicated because it tries to achieve three goals [60]:

1. to store only the nonzero elements,
2. to operate only on the nonzero elements, and
3. to preserve sparsity.

Sparse matrices are stored in compact formats to reduce the storage requirements and computation time. The compact representation must store not only the values of the nonzero elements, but also the indexing information which specifies the position of each nonzero element in the *regular* matrix. Sparse computation is complicated by the fact that many different schemes of storing formats have been devised to take advantage of the structure of the sparse matrix or the specialty of the problem being solved. For example, if a matrix is known to consist of a few diagonals, each of these diagonals can be stored as a column of a dense matrix, and the offsets of each diagonal vector with respect to the main diagonal are stored separately. This *diagonal format* is illustrated in Figure 1.1. If the matrix is not regularly structured, then the simple *coordinate storage format* can be used. In this scheme, the values of nonzero elements and their row and column indices are stored in three vectors, as illustrated in Figure 1.2. A third popular format is the *compressed row storage format*. As shown in Figure 1.3, the nonzero elements are stored row by row in an array $data()$, the column indices for each nonzero element are stored in array $column$, and the starting positions of the nonzero elements of each row in $data()$ are stored in array $rowptr$. In fact, there are more than fifteen common storage schemes for sparse matrices [66].

The use of compact data structure makes the implementation of sparse matrix computations more complicated than that of dense matrix computations. For example, the implementation of sparse matrix addition or multiplication often splits into two parts: the *symbolic section*, which does storage allocation or data structure set-up, and the *numerical section*, which performs the actual

$$\begin{pmatrix} 1 & 2 & 0 & 0 & 0 & 0 & 0 \\ 3 & 4 & 5 & 0 & 0 & 0 & 0 \\ 0 & 6 & 7 & 0 & 0 & 0 & 0 \\ 8 & 0 & 9 & 10 & 11 & 0 & 0 \\ 0 & 12 & 0 & 0 & 13 & 14 & 0 \\ 0 & 0 & 15 & 0 & 16 & 17 & 18 \\ 0 & 0 & 0 & 19 & 0 & 20 & 21 \end{pmatrix}$$

(a) A sparse matrix

diagonal vectors

| | | | |
|---|---|---|---|
| - | - | 1 | 2 |
| - | 3 | 4 | 5 |
| - | 6 | 7 | 0 |
| 8 | 9 | 10 | 11 |
| 12 | 0 | 13 | 14 |
| 15 | 16 | 17 | 18 |
| 19 | 20 | 21 | - |

offset

| | | | |
|---|---|---|---|
| -3 | -1 | 0 | 1 |

(b) Diagonal Storage Format

Figure 1.1: Diagonal storage format

$$\begin{pmatrix} 1 & 2 & 0 & 0 & 0 & 0 & 0 \\ 3 & 0 & 0 & 4 & 0 & 0 & 0 \\ 0 & 0 & 5 & 0 & 0 & 0 & 0 \\ 6 & 0 & 7 & 0 & 0 & 8 & 9 \\ 0 & 10 & 0 & 0 & 11 & 0 & 0 \\ 0 & 0 & 12 & 0 & 0 & 0 & 0 \\ 0 & 13 & 0 & 14 & 0 & 0 & 0 \end{pmatrix}$$

(a) A sparse matrix

data

| | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 |

row

| | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 1 | 2 | 2 | 3 | 4 | 4 | 4 | 4 | 5 | 5 | 6 | 7 | 7 |

column

| | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 2 | 1 | 4 | 3 | 1 | 3 | 6 | 7 | 2 | 5 | 3 | 2 | 4 |

(b) Coordinate Storage Format

Figure 1.2: Coordinate storage format

$$\begin{pmatrix} 1 & 2 & 0 & 0 & 0 & 0 & 0 \\ 3 & 0 & 0 & 4 & 0 & 0 & 0 \\ 0 & 0 & 5 & 0 & 0 & 0 & 0 \\ 6 & 0 & 7 & 0 & 0 & 8 & 9 \\ 0 & 10 & 0 & 0 & 11 & 0 & 0 \\ 0 & 0 & 12 & 0 & 0 & 0 & 0 \\ 0 & 13 & 0 & 14 & 0 & 0 & 0 \end{pmatrix}$$

(a) A sparse matrix

data

| | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 |

column

| | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 2 | 1 | 4 | 3 | 1 | 3 | 6 | 7 | 2 | 5 | 3 | 2 | 4 |

rowptr

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 1 | 3 | 5 | 6 | 10 | 12 | 13 | 15 |

(b) Compress Row Storage Format

Figure 1.3: Compress row storage format

```
ip = 1
ix(1:m) = 0
do i=1, n
   ic(i) = ip
   do jp=ia(i),ia(i+1)-1
      j = ja(jp)
      jc(ip) = j
      ip = ip + 1
      ix(j) = i
   end do
   do jp=ib(i),ib(i+1)-1
      j = jb(jp)
      if ( ix(j) .neq. i ) then
         jc(ip) = j
         ip = ip + 1
      end if
   end do
end do
ic(n+1) = ip

do i=1, n
   do ip=ic(i), ic(i+1)-1
      x(jc(ip)) = 0
   end do
   do ip=ia(i), ia(i+1)-1
      x(ja(ip)) = an(ip)
   end do
   do ip=ib(i), ib(i+1)-1
      x(jb(ip)) = x(jb(ip))+bn(ip)
   end do
   do ip=ic(i), ic(i+1)-1
      cn(ip)=x(jc(ip))
   end do
end do
```

Figure 1.4: Sparse matrix addition

numerical addition or multiplication. By contrast, in dense matrix addition or multiplication, we often have only a simple numerical section. Figure 1.4 shows a code section that implements the addition of two sparse matrices stored in the compressed row storage format [60]. The first loop calculates how many nonzero elements each row in the summation matrix will have and finds the column number for each nonzero element in the summation matrix. The second loop does the numerical summation. The resulting matrix is also stored in the compressed row storage format.

The goal of preserving sparsity also complicates the design and programming of sparse matrix computation. A typical example is the handling of *fill-in*. A fill-in refers to an element whose value is initially zero but later becomes nonzero during computations (like Gauss elimination). The number of fill-in's can be reduced by reordering the columns or rows of the sparse matrix before the computation. Hence, sparse computation algorithms, such as sparse Cholesky factorization, often employ an extra reordering phase that is not necessary in dense Cholesky factorization. Moreover, finding a permutation that minimizes fill-in is an NP-complete problem [83]. Different heuristic approaches have been proposed to solve this problem, which further complicate the design and programming of sparse matrix computation.

### 1.1.3 Irregular Computation

Depending upon one's point of view, there are different criteria to determine whether an application or a problem is *irregular*. In this thesis, we consider an application *irregular* if it employs indirectly accessed arrays or pointers to deal with linked lists, trees, or graphs. The data access patterns in such applications cannot be analyzed by traditional compiler techniques which assume affine loop bound expressions and affine array subscript expressions and, therefore, are considered dynamic or irregular.

Sparse computation is one kind of irregular computation. In a sense, a sparse matrix should not be thought of as a matrix at all, but rather as a graph. Other examples of irregular computation problems include: finite element methods, hierarchical N-body problems, molecular dynamics, etc.

Like sparse computation, efficient irregular computation programs tend to be difficult to design and implement.

## 1.2 Compiler Analysis and Parallelization of Sparse and Irregular Applications

### 1.2.1 Compiler Support

Ever since the first compiler in the mid-1950s, compilers have been used to facilitate the development of efficient and sophisticated software at an affordable cost. Compilers serve as a connection between high level languages and machine architectures. Architectural details are hidden from programmers so that the programmers can focus on designing algorithms rather than worry about mapping computations to hardware features. Compiler support is especially important in developing efficient sparse and irregular applications for modern computer architectures, where both the algorithms and the machines are complicated.

Compiler support for sparse and irregular programming can be put into two classes: high level language level and intermediate representation (IR) level.

1. High level language level

   Sparse and irregular applications employ complicated data structures. When the sparse and irregular data structures are implemented in data types provided by traditional programming languages, like arrays and pointers of Fortran and C, two problems occur. First, the high level semantic information about the data structure usually is lost. Compilers have to rely on expensive and complicated analyses, such as data dependence tests and pointer analyses, to "uncover" the data structure information in order to generate quality codes. The optimization of the code generated often suffers in the absence of powerful IR level support. Second, because the data types used are very primitive, the programmers have to be aware of the machine architectures to generate codes with good performance. This not only increases the burden of programmers, but also makes codes less readable, less flexible, and less portable.

   A way to improve this is to use new languages or extend the constructs or semantics of existing programming languages. For example, storage schemes are expressed explicitly so that the high level semantic information can be preserved. Typical examples are the extension to the HPF, Vienna Fortran, and Fortran D [31, 75, 78]. As another example, a user can write sparse

programs in the same way that dense programs are written, and use a compiler to translate the codes into their sparse versions which use storage schemes specified by the programmer and chosen automatically by the compiler [12, 46].

2. Intermediate representation level

   The goal of support at this level is to develop new or extend traditional compiler analysis and optimization techniques to better serve sparse and irregular applications. Examples of analyses and optimizations include constant propagation, dead-code elimination, loop-invariant code motion, unnecessary bounds-checking elimination, software pipelining, and data prefetching. These have been proven important in improving program performance.

   The dynamic and irregular nature of sparse and irregular programs presents challenges to applying these techniques. For example, indirectly accessed arrays are frequently used in sparse and irregular programs, and most traditional compiler techniques can handle only arrays with affine subscripts. Thus, a compiler has to assume the subscript can have any value and gives up some optimizations. Figure 1.5 illustrates an example of loop-invariant code motion. Expression $x(i)$ in statement (1) is loop-invariant in loop `do j`, and the value of $a(x(i))$ is never changed during the execution of loop `do j`. Therefore, the computation of $a(x(i)) * a(x(i))$ in statement (2) can be moved to the front of loop `do j`, as shown in Figure 1.5.(b). This code motion would be wrong if $x(i)$ were ever equal to $j$ such that statement (1) were flow dependent on statement (2). Traditional techniques assume this dependence and do not perform the optimization. However, advanced techniques, like those discussed in this thesis, can be used to replace the $x(i)$ in statement (1) with $i + 1$ and make the subscript an affine expression.

   The compiler support at this level does not require explicit user or high level language support. In fact, from the compiler point view, these techniques do not distinguish between dense and regular applications and sparse and irregular applications. They solve problems that appear more frequently in sparse and irregular programs than in dense and regular programs. Nevertheless, these techniques also are useful for analyzing complicated dense and regular programs. The compiler analysis techniques presented in this thesis fall into this category.

```
do k=1, n
   x(k) = k+1
end do

do i=1, n
   do j=1, i
      t = a(x(i))                (1)
      a(j) = t*t + a(j)          (2)
   end do
end do
```

(a)

```
do k=1, n
   x(k) = k+1
end do

do i=1, n
   t = a(x(i)) * a(x(i))
   do j=1, i
      a(j) = t + a(j)
   end do
end do
```

(b)

Figure 1.5: Loop-invariant code motion

One compiler optimization that is essential nowadays is parallelization, which transforms the program to exploit the parallel execution functions provided by the underlying machine architecture.

### 1.2.2 Parallelization of Sparse and Irregular Applications

Sparse and irregular computations usually contain more inherent parallelism than their dense and regular counterparts, yet are more difficult to execute efficiently on parallel machines [43, 30]. The low efficiency is partly due to poor data distribution and task balancing. For example, the normal data distribution schemes for dense matrices, such as block, cyclic, and a combination of both, are not suitable for sparse matrices whose structures usually are irregular. Solving these problems with a compiler has been the focus of many recent studies. Several compile-time or run-time solutions have been proposed, and great improvements have been achieved [56, 78, 76]. However, the very first question of parallelization (i.e., how to find the parallelism) still remains a problem that, to a large degree, has to be solved manually.

Most sparse and irregular computations have different levels of parallelism. For example, the sparse Cholesky factorization based on the elimination tree method has three levels of parallelism [51, 43]. In coarse-grain parallelism, each thread processes all columns in a branch of the elimination tree. In medium-grain parallelism, each thread updates a single column. And, in fine-grain parallelism, each thread does a multiply-add operation on one element in a column.

Current computer architecture exhibits at least three levels of parallel execution functions: clusters, within a cluster, and within a processor. For a sparse and irregular application to be scalable

and execute efficiently, it is important to exploit the different levels of parallelism. Algorithm designers and programmers are good at dealing with high level or coarse grain parallelism. However, exploiting medium or fine grain parallelism by hand is tedious and prone to errors and is a job better left to the compiler.

## 1.3 Difficulties in Automatic Parallelization of Sparse and Irregular Applications

### 1.3.1 Parallelism Detection

Parallelizing compilers analyze sequential programs to detect inherent parallelism and then use this information to generate parallel programs. Because the most computationally intensive part of scientific computations is often inside loops, most parallelization techniques focus on loop parallelization.

While determining the semantic validity of executing two sections of a sequential program in parallel is undecidable [11], the sequential order of statement execution can be relaxed and the semantics of the original program is retained if the execution order specified by *dependence relation* is maintained.

A statement $S$ is dependent on a statement $T$ if, during the program execution, statement $T$ provides a value that is later used by statement $S$. This is called *flow* dependence or *true* dependence, and it is the dependence relation that should be maintained in parallelizing functional and dataflow languages.

In imperative languages, a programmer can use and reuse memory locations via accesses to variables. Therefore, in addition to flow dependences, *anti* dependences and *output* dependences need to be maintained. A statement $S$ is anti-dependent on a statement $T$ if, during the program execution, statement $T$ reads a variable that is later modified by statement $S$. A statement $S$ is output-dependent on a statement $T$ if, during the program execution, statement $T$ writes a variable that is also later written by statement $S$. Anti-dependence and output-dependence are artificial dependences. They can be eliminated by introducing new variables without changing the meaning of the program.

Array data dependence tests are used in finding whether a loop can be parallelized. An array data dependence test checks whether two array accesses (at least one being a write) in two different iterations of the loop may access a common array element. This problem can be formalized as finding an integer solution to a *dependence system* [84], which is a system of equations representing array subscript expressions in terms of loop indices and a set of inequalities describing the boundaries of the loop indices. For dense and regular programs, both the equations and the bound constraints are often linear and the array data dependence test is equivalent to the integer programming. During the last ten years, many algorithms (both approximate and exact) have been proposed for dealing with this NP-Complete problem and have demonstrated good results in practice [8, 62].

Unfortunately, in sparse and irregular applications, there are many cases when the array subscripts cannot be represented in closed-form expressions in terms of loop indices. In such cases, the data dependence problem cannot be characterized by the traditional dependence system, and dependence is assumed. For example, the subscript expression of array $a()$ in statement (1) of Figure 1.5 is an array element $x(i)$. In the traditional data dependence test framework, $x(i)$ is considered "unanalyzable" and dependence is assumed.

### 1.3.2 Parallel Transformation

After inherent parallelism is detected, a compiler then determines which of the parallel parts to run in parallel and generates the code for the target machine. The irregular and dynamic nature of sparse and irregular applications also makes it difficult to generate efficient parallel programs.

For example, at the processor level, due to the overhead of parallel execution, coarse grain parallelism is preferred to fine grain parallelism. Loop interchanging is a useful method to change the granularity of parallel loops [81]. However, loop interchanging is not always legal. Data dependence information is required to test the validity of loop interchanging. As in the dependence test for parallelism detection, new techniques are required to analyze irregular memory accesses in sparse and irregular programs; otherwise, loop interchanging cannot be performed safely.

A compiler also can transform programs to expose more parallelism, such as by eliminating anti-dependences and output-dependences. For example, array privatization [28, 47, 54, 74] is a

technique to remove these artificial dependences among array accesses. Current array privatization techniques are based on the common "set-before-use" access pattern of temporary arrays found in dense and regular programs. In sparse and irregular programs, the "set-before-use" way of using temporary arrays may cause very large overhead; thus, a "reset-after-use" approach is often taken instead. New array privatization techniques need to be developed to handle this case.

### 1.3.3 "Mission Impossible"?

Automatic parallelization has been a useful alternative to manual parallelization for regular/dense computations [16]. However, automatic parallelization for sparse and irregular problems is not well understood. Although it is widely believed that automatic detection of parallelism in sparse and irregular computations is difficult or impossible due to the presence of complex subscript array expressions, there is practically no empirical evidence to support this belief.

In this research, we developed several compiler techniques for analyzing and transforming sparse and irregular programs. These techniques were identified as important in parallelizing a collection of sparse and irregular programs. Experimental results showed that it was possible for a parallelizing compiler to automatically parallelize sparse and irregular programs as well as it could for dense and regular programs.

## 1.4 Research Overview

### 1.4.1 Empirical Study

In order to understand the challenges to analyzing and parallelizing sparse and irregular programs, we started our research by studying a collection of Fortran programs with irregular data access patterns.

We used a current parallelizing compiler (Polaris) to parallelize these programs and examined its output. By studying the loops that should have been parallelized but were not and the loops that were parallelized but executed poorly on the target machine, we identified several key techniques that must be studied. They are:

- new ways to detect privatizable arrays,

- analysis of irregular array accesses,
- elimination of overhead in run-time tests,
- efficient parallelization methods for irregular reduction loops, and
- parallelization of premature exit loops.

We then proposed several techniques to solve each of the above problems.

### 1.4.2 Compiler Analysis of Irregular Array Accesses

Irregular array accesses are array accesses whose array subscripts do not have closed-form expressions in terms of loop indices. As we have discussed above, traditional array analysis and loop transformation techniques cannot handle irregular array accesses. In our research, we studied, in detail, two kinds of simple and common cases of irregular array accesses: *single-indexed access* and *indirect array access*. We present techniques to analyze these two cases at compile-time.

An irregular array access is *single-indexed* in a loop if the array is always subscripted by the same index variable in the loop. An array reference is *indirectly accessed* if the subscript of the array contains another array.

We use different techniques to handle these two different kinds of irregular accesses. For irregular single-indexed accesses, we use a *bounded depth-first search* method to trace how the values of index variables are changed between two array accesses. We present techniques that can determine two useful access patterns: *consecutively-written* and *stack access*. For simple indirect array accesses, we use an *interprocedural demand-driven array property analysis* method. These two techniques are simple and effective. They take advantage of the fact that, in real programs, irregular array accesses often follow a few fixed patterns and have detectable properties. We also found it important to use these two techniques together in analyzing irregular array accesses.

The compile-time analysis of irregular array accesses can enhance other analyses and optimizations, such as data dependence tests, privatization tests, loop parallelization, loop interchanging, and eliminations of run-time array bounds-checking.

We present experimental results showing the effectiveness of the techniques in finding more implicit loop parallelism at compile-time. Nine more loops in five real programs were found par-

allel, and four programs achieved considerable performance improvement after these loops were parallelized.

### 1.4.3 Parallelization of Irregular Reduction Loops

Irregular reduction loops appear in the kernels of many sparse and irregular scientific computation programs. We studied and compared five different parallelization methods which fall in one of two classes: iteration domain decomposition methods and data domain decomposition methods.

Irregular reduction loops can take many different forms. In our study, we found iteration domain decomposition methods were more versatile than data domain decomposition methods and could easily parallelize different forms of reduction loops. Data domain decomposition methods require more compiler support to transform nested loops or loops with multiple access patterns.

To compare the effectiveness of these methods, we used five different applications, three of which have been previously used by other researchers in their study of irregular reduction loops. We generated five different versions using the five different methods for each of the programs and compared the running performance of these versions.

Our study concludes that there is not a single best way to parallelize irregular reduction loops. The best strategy depends on the loop pattern, the distribution of input data, and the resource constraints. We derived some guidelines for method selection.

### 1.4.4 Organization of Dissertation

This dissertation is organized as follows. Chapter 2 describes the empirical study we did at the beginning of this research and discusses the key techniques we identified. The following four chapters detail two of the techniques, namely irregular array access analysis and parallelization of irregular reduction loops. Chapter 3 discusses analysis of single-indexed array access and Chapter 4 discusses analysis of indirectly accessed arrays. The effectiveness of the techniques presented in these two chapters is demonstrated in the experiments described in Chapter 5. Chapter 6 discusses five different methods of parallelizing irregular reduction loops and the experiment we used to evaluate their performance. Finally, Chapter 7 concludes this dissertation and gives several directions for future work.

# Chapter 2

# TECHNIQUES NEEDED FOR AUTOMATIC PARALLELIZATION OF SPARSE AND IRREGULAR FORTRAN PROGRAMS

## 2.1 Introduction

As shown in Chapter 1, sparse and irregular applications pose new challenges to compiler analysis and optimization techniques, such as automatic parallelization. To understand the problem, we began our research by identifying the compiler techniques that were required to automatically parallelize a collection of scientific computation programs with irregular memory access patterns. We used the Polaris compiler [14] to facilitate our identification.

The Polaris compiler is a research parallelizing compiler developed at the University of Illinois at Urbana-Champaign. It takes conventional Fortran programs and generates parallel versions of these programs for shared-memory multiprocessors and scalable machines with a global address space. The previous work in the Polaris project has demonstrated that significant progress in translating programs written in conventional languages for parallel computer architectures is possible [14].

To identify the techniques required for parallelizing sparse and irregular programs, first we used Polaris to parallelize our benchmark programs and measured their execution time. Then, we hand-analyzed the parallelized programs. Through this hand-analysis, we studied whether the loops detected as sequential by Polaris were, in fact, parallelizable or not, and classified the reasons why Polaris was not able to parallelize them. We also studied those automatically parallelized loops

| Code | Description | Origin | LOC | Seq. Exe. Time |
|---|---|---|---|---|
| CHOLESKY | Sparse Cholesky factorization | HPF-2 | 1284 | 323s |
| EULER | Euler equations on a irregular mesh | HPF-2 | 1990 | 972s |
| SPLU | Sparse LU factorization | HPF-2 | 363 | 1958s |
| GCCG | Computational fluid dynamics | Univ of Vienna | 407 | 374s |
| LANCZOS | Eigenvalues of symmetric matrices | Univ of Malaga | 269 | 389s |
| P3M | N-body (Particle-mesh method) | NCSA | 2414 | 890s |
| DYFESM | Analysis of anisotropic structures | PERFECT | 7650 | 25s |
| BDNA | Modular dynamics | PERFECT | 4896 | 83s |
| TRFD | Simulation of integral transformation | PERFECT | 380 | 47s |
| TREE | Hierarchical N-body | Univ. of Hawaii | 1533 | 60s |
| SparAdd | Addition of two sparse matrices | Pissanetzky | 67 | 2.40s |
| ProdMV | Prod of a sparse matrix by a col vector | Pissanetzky | 28 | 1.28s |
| ProdVM | Prod of a row vector by a sparse matrix | Pissanetzky | 31 | 1.07s |
| SparMul | Product of two sparse matrices | Pissanetzky | 64 | 3.32s |
| ProdUB | Product of matrices $U^{-T}$ and $B$ | Pissanetzky | 49 | 3.72s |

Table 2.1: Benchmark codes

that did not execute efficiently on the target machine. For the studied loops that were inherently parallel, we hand-transformed them into an efficient parallel form. We applied transformations that could be potentially implemented by a parallelizing compiler. The improved versions were run on the same machine and their execution times were compared with the originals. In most of the programs we inspected, we found that our transformations could improve the program performance by a significant factor. Roughly speaking, the average speedups of these programs were comparable to the results of previous studies of programs without irregular access patterns parallelized by Polaris. This strengthened our belief that automatic parallelization can work on sparse and irregular programs as well as it does on dense and regular programs.

In this chapter, we discuss the techniques we identified. They can be classified as parallelism detection techniques (Section 2.4, Section 2.5, and Section 2.6) and parallel transformation techniques (Section 2.7 and Section 2.8). Although these techniques were developed with parallelization in mind, most of them are general analysis techniques that can be used in optimizations beyond parallelization.

Before we present these techniques, we first describe, in the following two sections, the benchmark codes we used and the experimental results.

## 2.2 The Benchmark Suite

Our benchmark suite consists of a collection of sparse and irregular programs written in Fortran 77. Table 2.1 lists all the codes [5]. CHOLESKY, EULER and SPLU are from the HPF-2 motivating application suite [31]. DYFESM, BDNA and TRFD are from the PERFECT benchmark suite [27]. P3M is a Grand Challenge code from the National Center for Supercomputing Applications (NCSA). GCCG was developed at the Institute for Software Technology and Parallel Systems at the University of Vienna. LANCOS was contributed by E. Gutierrez at the University of Malaga, Spain [5]. And, TREE was written by J. E. Barnes at the Institute of Astronomy, University of Hawaii [10]. We also used five kernel codes that implemented several sparse matrix computation algorithms given by S. Pissanetzky [60]. These programs were chosen because they had sparse and irregular data access patterns and they included parallel idioms that were important to full-scale sparse and irregular applications. The sequential execution times in Table 2.1 were obtained on the SGI Challenge machine described in the next section.

As usual, the choice of benchmark codes is critical. The techniques and results in this study are based on the codes in this suite. There is the open question of whether our findings carry over to other programs. To answer this question, besides studying a broader collection of programs, we need to understand the rationale of the existence of problems our techniques try to solve. Are they caused by some "fancy coding tricks" used by one or two "smart" programmers? Or are they the result of some common programming patterns that most programmers follow? For each of techniques presented in this chapter, we describe not only what it is, but also why it is important.

## 2.3 Experimental Results

We used Polaris to automatically parallelize the programs in our benchmark suite for SGI Challenge machines. The parallelized versions were compiled by the native SGI MIPSPro Fortran compiler 7.30 (option -O2) and ran a four-processor SGI Challenge machine (four 200MHz R4400 MIPS processors, 256MB memory, 16KB instruction cache, 16KB data cache, 4MB second level cache, and running IRIX64 6.5). In Figure 2.1, the speedups of the Polaris versions are represented by the white bars and the speedups of the manually improved versions are represented by the black bars.

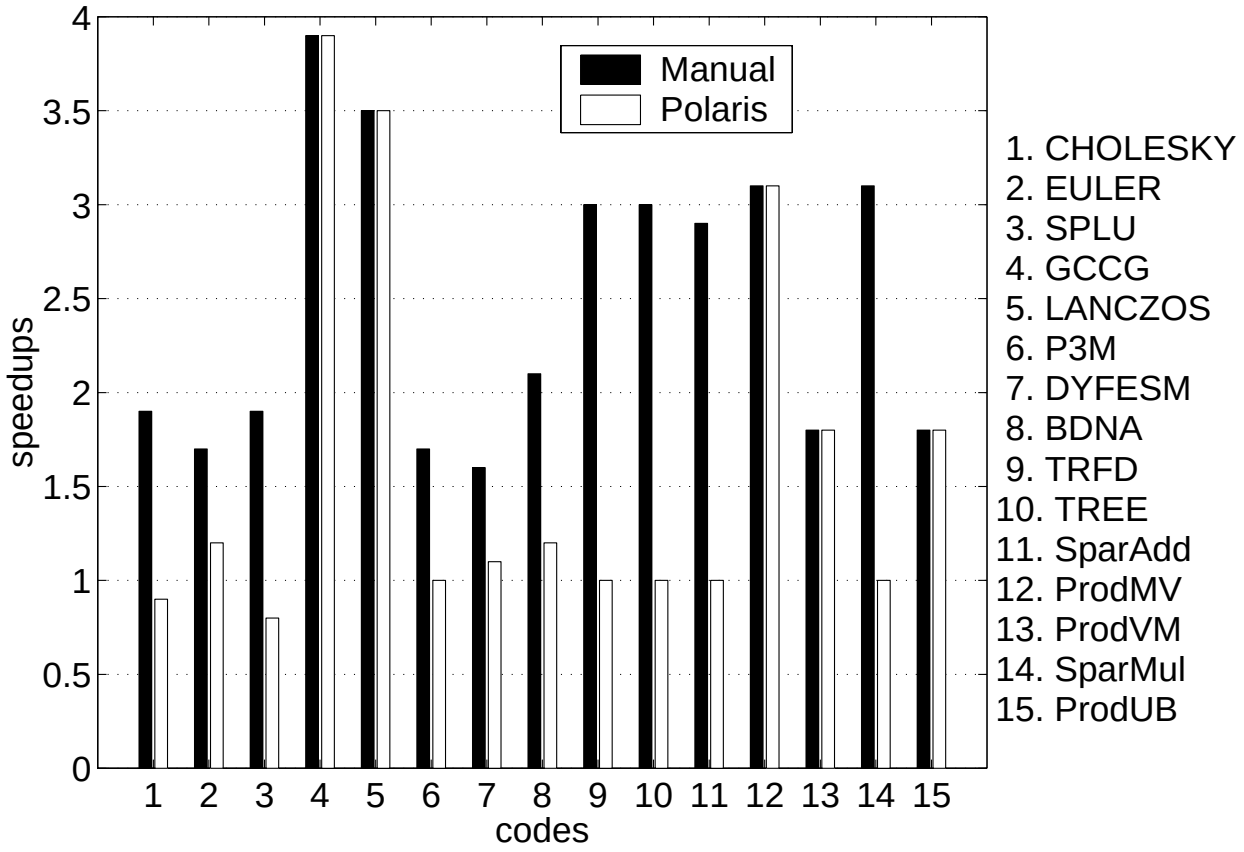



Figure 2.1: Experimental results

For five of the fifteen programs, namely GCCG, LANCZOS, ProdMV, ProdVM, and ProdUB, the Polaris parallelized versions had the same speedups as the manual versions; for the remaining ten, the manually improved versions achieved better performance.

The five Polaris parallelized versions achieved the same speedups as their corresponding manual versions for two different reasons. First, in programs GCCG, LANCZOS, and ProdMV, all the irregular array accesses appear in the arrays that are read-only in loops, as illustrated in Figure

```
DO nc = nintci, nintcf
   direc2(nc) =   bp(nc)*direc1(nc)
                - bs(nc)*direc1(lcc(nc,1))
                - bw(nc)*direc1(lcc(nc,4))
                - bl(nc)*direc1(lcc(nc,5))
END DO
```

Figure 2.2: Array `direc1()` is read-only in the loop.

| | | Parallelism Detection | | | Transformation | |
|---|---|---|---|---|---|---|
| | | New Array Privatization | Irregular Array Access | Run-time Test | Irregular Reduction | Premature Exit Loop |
| 1 | CHOLESKY | x | x | x | x | x |
| 2 | EULER | | | | | |
| 3 | SPLU | x | x | x | x | |
| 6 | P3M | | x | | | |
| 7 | DYFESM | | x | | | |
| 8 | BDNA | | x | | | |
| 9 | TRFD | | x | | | |
| 10 | TREE | | x | | | |
| 11 | SparAdd | | x | x | | |
| 14 | SparMul | | x | x | | |

Table 2.2: Compiler techniques required to parallelize the benchmark codes

2.2. Therefore, these irregular accesses did not cause any difficulties for traditional data dependence tests. The Polaris versions had the same parallel loops as the manual versions did, and both versions did the same transformations.

Second, in programs ProdVM and ProdUB, the parallel loops are irregular reduction loops. These loops were parallelized by Polaris with the replicated copy method. These loops are simple. When only four processors were used, the replicated copy versions had relatively small overhead and had better performance than the versions parallelized by other methods, although the latter might be better if more processors were used. We will elaborate on this in Section 2.7.

Table 2.2 shows the new techniques required or the techniques that need to be improved to parallelize the other benchmark codes. An 'x' in the table means the technique that solves the problem in the corresponding column is needed for parallelizing the code in the corresponding row. The techniques are described in the following sections.

The rest of this chapter is organized as follows. Section 2.4 discusses two new methods of detecting privatizable arrays. Section 2.5 shows that we need compiler analysis for irregular memory accesses. Section 2.6 gives three methods to improve run-time dependence tests. Section 2.7 discusses irregular reduction. And, Section 2.8 describes how to parallelize premature-exit loops. The techniques discussed in Sections 2.5 and 2.7 are not detailed because the next four chapters are dedicated to these topics.

## 2.4 New Methods of Detecting Privatizable Arrays

### 2.4.1 Traditional Array Privatization Techniques

Array privatization is one of the most important transformations for program parallelization [28, 47, 54, 74]. It identifies arrays that are used as temporary work spaces within a loop iteration, and then allocates local (private) copies of the arrays for each iteration or thread. By doing this, it eliminates any cross-iteration anti-dependence or output-dependence caused by the reuse of the temporary arrays, and potentially exposes more parallelism.

Generally speaking, an array section $a[p:q]$ can be privatized in a loop if there does not exist an iteration $i$ and an array element $a[k]$ ($k \in [p,q]$) such that the definition of $a[k]$ in iteration $i$ can reach any use of $a[k]$ in any iteration $j$ that is later than iteration $i$. It can be formalized as follows.

**Proposition 2.1** *Given a loop with iteration range of* 1 *to* $n$*, let* $write(i)$ *be the set of array elements in* $a[p:q]$ *that are written in iteration* $i$*, and* $use_{exp}(i)$ *be the set of array elements in* $a[p:q]$ *that have upwards exposed use in iteration* $i$*. Then the array section* $a[p:q]$ *can be privatized in the loop if*

$$\forall 1 \leq i < j \leq n, write(i) \cap use_{exp}(j) = \emptyset \qquad (1)$$

To test the condition (1), *element-based* array dataflow analysis [29, 54, 64, 53] is required to gather flow information for each array element in the section $a[p:q]$. This approach involves solving several integer programming problems and is expensive [29, 33].

In practice, a sufficient condition of (1) is often tested to determine the validity of privatization.

**Corollary 2.1** *Given a loop with iteration range of* 1 *to* $n$*, let* $use_{exp}(i)$ *be the set of array elements in* $a[p:q]$ *that have upwards exposed use in iteration* $i$*. Then the array section* $a[p:q]$ *can be privatized in the loop if*

$$\forall 1 \leq i \leq n, use_{exp}(i) = \emptyset \qquad (2)$$

To test condition (2), a *section-based* array dataflow analysis is usually sufficient [33, 34, 74]. The test consists of two steps. First, the section of array elements that each statement can read or write in one iteration is calculated. Second, the read sections of each statement are subtracted by the write sections of the statements that dominate the read statement to get the upwards exposed read sets.

This approach works well in many programs because iteration-based temporary arrays are often first initialized upon entering the iteration and then used through the rest of the iteration. They follow the "set-before-use" pattern.

Besides the simplicity of calculation, this method has other advantages. First, "copy-in" is not required. The "copy-in" refers to the operations that copy the values of the global arrays to the private arrays. If an array is found privatizable by using the test in Proposition 2.1, then the value of an array element read in the loop may come from outside the loop. Thus, the array values should be assigned to the private copies before entering the loop. On the other hand, if the array is found privatizable by using the test in Corollary 2.1, then the values of the private array elements always come from the same iteration. Hence, copy-in is not necessary. Second, arrays that are found privatizable by this method can be privatized for each processor instead of for each iteration, and the iterations assigned to a processor can be executed in any order. If an array is found privatizable by satisfying condition (1), the array also can be privatized for each processor, but all the iterations assigned to one processor must be executed in the same order as in the sequential loop.

However, this simple approach sometimes does not always work for sparse and irregular applications. In sparse and irregular applications, although temporary arrays sometimes follow the set-before-use pattern, the irregular access pattern often makes it impossible to get the precise section of array elements that are read or written by one statement in an iteration. For example, for the array $x()$ in the loop `do i` in Figure 2.3, it is trivial to know that $x[1:m]$ is written by statement (1); but, with no further information about array $pos()$, we have to assume that any element of $x()$ may be read by statement (2). This assumption makes the calculated upwards exposed read set not empty and, as a result, array $x()$ cannot be privatized by testing condition (2).

There are also cases in sparse and irregular applications where temporary arrays do not follow the set-before-use pattern. In these cases, although the arrays are privatizable, the upwards exposed

```
do i=1, n
   do j=1, m
      x(j) = ..              (1)
   end do
   do k=1, l
      y(i,k) = x(pos(k))     (2)
   end do
end do
```

Figure 2.3: An example of self-covered dominating write

read sets are not empty, and condition (1) must be tested.

We also found that, in sparse and irregular applications, some arrays could be privatized even though their access patterns did not satisfy condition (1). In these cases, we need to extend the domain of privatizable arrays.

Here, we present two new techniques that can detect a broader range of privatizable arrays in sparse and irregular programs than the traditional methods can.

### 2.4.2 Self-covered Dominating Writes

In this section, we give an array privatization detection technique that tests another sufficient condition for (1). This technique does not require the calculation of read sets.

**Corollary 2.2** *Given a loop with iteration range of* 1 *through* $n$*, condition (1) is true if*

1. *in each iteration, all the write accesses to array* $x()$ *happen before any read access to array* $x()$*, and*

2. $\forall 1 \leq i < j \leq n, write(i) \subseteq write(j)$*, where* $write(k)$ *represents the set of array elements written in iteration* $k$*.*

*Proof*

To see why the corollary is correct, notice that because of the first condition, the upwards exposed read set for iteration $i$ can be calculated by using the following equation:

$$use_{exp}(i) = use(i) - write(i),$$

and $\forall 1 \leq i < j \leq n$,

$$\begin{aligned} & write(i) \cap use_{exp}(j) \\ = \; & write(i) \cap (use(j) - write(j)) \\ = \; & write(i) \cap use(j) - write(i) \cap write(j) \\ = \; & write(i) \cap use(j) - write(i) \\ = \; & \emptyset \end{aligned}$$

□

In general, to verify the first condition in Corollary 2.2, we need to solve an integer programming problem. However, there are simple cases where the access order can be verified by checking the dominance relationship between the read statements and the write statements in the loop body.

The biggest advantage of this method is that it does not require the computation of the read sets nor the upwards exposed sets. Thus, it can be used for loops with read sets that are difficult to compute. For example, for the loop in Figure 2.3, because the write set of array $x()$ is always $x[1:m]$, we can easily determine that $x()$ is privatizable without knowing which elements of $x()$ are read in each iteration.

This method does not compute the read set, nor does it know whether an iteration may read an array element that has a value coming from outside the loop. Thus, *copy-in* is generally required for the privatized arrays to ensure the semantic correctness, unless it can be verified that the smallest write set covers the domain of the array declared in the program.

### 2.4.3 Reset-after-use Pattern

In sparse and irregular applications, we found a class of arrays that did not satisfy the condition (1) and still could be privatized. An example of this kind of privatizable array is shown in Figure 2.4. In this example, array $t()$ can be privatized in loop `do j`. Before entering loop `do j`, the program initializes the value of $t()$ to 0 in loop `do i`. In each iteration of loop `do j`, the elements of $t()$ that are modified in statement (3) are always reset to 0 in loop `do l` before the program leaves the iteration. Because the values of the elements in $t()$ are always 0 when the program enters each

```
real t(1:m)

do i=1, m
   t(i) = 0                          (1)
end do
do j=1, n
   k = 0
   ...
   while (..) do
       .. = t(..) op express         (2)
       k = k+1
       t(k) = ..                     (3)
   end while
   ...
   do l = 1, k
      t(l) = 0                       (4)
   end do
end do
```

Figure 2.4: An example of reset-after-use pattern

iteration, a private copy of $t()$ for each iteration with the initial value 0 can be used. Thus, array $t()$ can be privatized. Because the value of $t()$ read by statement (2) may come from an earlier iteration of loop `do j`, condition (1) is not satisfied and $t()$ is not privatizable according to the traditional criteria.

There are two reasons that people use temporary work space in this "reset-after-use" way instead of following the set-before-use pattern. First, the size of $t()$ declared may be very large and the number of elements that are modified in each iteration may be very small, thereby creating a high overhead cost to initialize all elements of $t()$ in the beginning of each iteration. Second, in many sparse and irregular programs, the positions where array $t()$ will be read or written in each iteration are usually unknown until the array is actually being referenced, such as the range $[1 : k]$ in the example. It is difficult to write initialization statements prior to the references. In such cases, reset-after-use is simpler than set-before-use and can avoid unnecessary computations. This programming style reflects the dynamic nature of sparse and irregular programs. We found this access pattern in CHOLESKY and SPLU.

If we define downwards exposed write for an iteration as the set of array elements that is written in an iteration but not reset to the initial value before the end of the iteration, then an array can be

privatized if the downwards exposed write is empty for each iteration. By working on the reversed control flow graph, we can check this condition in a way similar to that of checking condition (2). In practice, this is more difficult because arrays that are reset-after-use are more likely to have irregular access patterns than the arrays that are set-before-use. To get the write or read sets, other techniques are often required, such as the analysis of irregular memory accesses described in the next section or the run-time test methods described in Section 2.6.

## 2.5 Compiler Analysis of Irregular Memory Access

Traditional loop optimization and array analysis techniques work on DO loops and require array subscript expressions to be expressed as closed-form expressions in terms of loop indices. In addition, most methods require the subscript expressions to be linear. However, in most sparse and irregular programs, array accesses are often *irregular*.

**Definition 2.1** *We define an array access as irregular if*

1. *no closed-form expression, in terms of the loop indices, for the subscript of the accessed array is available at compile-time, or*
2. *the subscript expression of the accessed array contains any unknown function of the loop indices.*

Because current analysis techniques cannot handle irregular array accesses, many code sections are left unoptimized. This was the most important reason that Polaris found fewer parallel loops than the manual version in the experiment discussed above. In particular, this was the case for CHOLESKY, SPLU, P3M, DYFESM, BDNA, TRFD, TREE, SparAdd and SparMul.

For example, as discussed in Section 2.4, array privatization [28, 47, 54, 74] is an important technique in loop parallelization. In many cases, an array can be privatized if all elements of the array that are read in one loop iteration are always first defined in the same iteration. In each iteration of the outermost loop `do k` in Fig. 2.5.(a), any element of $x()$ read by statement (3) in loop `do j` is defined by statements (1) and (2) in the *while* loop. Therefore, array $x()$ can be privatized in loop `do k`. Because there are no cross-iteration dependences, loop `do k` can be

```
do k=1, n
   p = 0
   i = link(1,k)
   while ( i != 0 ) do
      p = p + 1
      x(p) = y(i)          (1)
      i = link(i,k)
      if ( cond(k,i) ) then
         p = p + 1
         x(p) = y(i)       (2)
      end if
      i = link(i,k)
   end do
   do j=1, p
      z(k,j) = x(j)        (3)
   end do
end do
```

(a)

```
do i = 1, n
   do j = 1, m
      x(j) = ..
   end do
   do k = 1, l
      y(i,k) = x(pos(k))
   end do
end do
```

(b)

Figure 2.5: Examples of irregular array accesses

parallelized. Current privatization tests require a closed-form expression of the array subscripts in terms of the loop indices in order to compute the section of array elements read or written in the loop. In this example, because there is no such expression for the index variable $p$, these techniques can determine only that section $[1:p]$ of array $x()$ is read in loop `do j`, but they cannot determine that the same section is also written in the *while* loop. Therefore, they fail to privatize $x()$.

A second example is the loop in Figure 2.5.(b), which has the same loop as in Figure 2.3. It illustrates an indirect array access that cannot be handled by current privatization tests. As discussed in Section 2.4, traditional techniques cannot determine whether array $x()$ can be privatized because $pos()$ may have any value if no further knowledge about $x()$ is available. Although we can privatize $x()$ according to Corollary 2.2, we need to test condition (2) in Section 2.4 if we want to avoid copy-in. If, by doing global program analysis at compile-time, we know that the value of elements in $pos[1:l]$ is within the range $[1,m]$, then we can privatize $x()$ for loop `do i` and parallelize loop `do i` without the need for copy-in.

The array accesses in the above two examples are irregular for two different reasons. The first one is irregular because the array index variable is modified conditionally in the loop and, therefore, no closed-form expression of the index variable is available. The second one is irregular because of

the use of an index array.

These two examples also illustrate two kinds of common irregular array accesses: *irregular single-indexed access* and *simple indirect array access.*

**Definition 2.2** *The accesses to an array in a given loop are irregular single-indexed if the array is always subscripted by the same index variable in the loop and the accesses are irregular.*

The access of array $x()$ is single-indexed in the *while* loop in Figure 2.5(a).

**Definition 2.3** *An array reference is indirectly accessed if the subscript of the array contains another array. We call the array itself the "host array", and the array in the subscript the "index array". An indirect array access is simple if the enclosing loop is a DO loop and the subscript of the index array is the loop index.*

The reference to array $x()$ in loop `do k` in Fig. 2.5(b) is a simple indirect array access. As a counter-example, $x(pos(k+i))$ is not a simple indirect array access.

The arrays we have discussed thus far are single dimensional. In the case of multi-dimensional arrays, we consider the subscript expressions dimension by dimension.

Although these two kinds of irregular array accesses are more simple than the general cases, they are the most common forms of irregular array accesses in sparse and irregular programs we have studied. We have developed compiler techniques to handle these two cases. For irregular single-indexed accesses, we use a *bounded depth-first* search method to trace how the values of index variables are changed between two array accesses. In Chapter 3, we present techniques that can determine two useful access patterns: *consecutively-written* and *stack access.* For simple indirect array accesses, we present, in Chapter 4, an array property analysis method that finds the properties of an index array and applies the property information in the analysis of the access pattern of its host array.

## 2.6 Overhead Elimination in Run-time Dependence Tests

### 2.6.1 What is Run-time Dependence Test?

When static analysis is complex or information that depends on the input data is needed, a run-time method becomes necessary. Run-time techniques can succeed where compile-time analyses fail because they have access to data in "real time".

The scheme of using a run-time dependence test is very simple. Suppose a given loop can be parallelized only when a certain condition is true, and the condition is unable to be verified at compile-time. The compiler can generate a parallel version and a sequential version of the loop and guard these two versions by a conditional statement that tests this condition. If, at run-time, the condition is true, then the parallel version is executed; otherwise, the sequential version is executed. A more complicated method is to run the parallel version speculatively instead of waiting for the test results, in the hope that the results will favor the speculation. If the test fails, the execution of the loop is "rolled back" and the sequential version is executed. To facilitate the rollback, temporary storage space is needed to save the value of variables that will not be modified in the loop. Speculative parallelization is useful when the test of the condition takes a considerable amount of time. To be efficient, speculative parallelization should be used heuristically to reduce the number of false speculations.

### 2.6.2 Current Run-time Dependence Tests

Depending on how the test conditions are obtained, run-time tests fall into two categories. The first kind of approach is a straightforward extension of compile-time tests [13]. As usual, a compile-time data dependence test or an array privatization test is performed. If the test is simplified to a set of conditions that cannot be verified statically, then testing codes are generated to check these conditions at run-time.

The advantage of this method is that the condition tests usually are simple and the execution time to perform the tests is often negligible compared with the execution time of the loop body. However, the effectiveness of this approach is limited by the compile-time tests on which the run-time tests are based. Because a compile-time dependence test usually checks a necessary condition

```
do i = 1, n
   if ( ... ) then        (0)
      a(i+m) = ...        (1)
   else
      ... = a(i)          (2)
   end if
end do
```

(a)

```
if ( (n-m < 1) or (n+m < 1) ) then
         doall i = 1, n
                ...
         end do
else
         do i = 1, n
                ...
         end do
end if
```

(b)

Banerjee's Test:

$$f(i) = i + m, 1 \leq i \leq n \Rightarrow [f_{min}, f_{max}] = [1 + m, n + m]$$

$$g(i) = i, 1 \leq i \leq n \Rightarrow [g_{min}, g_{max}] = [1, n]$$

$$[1 + m, n + m] \cap [1, n] = \emptyset? \Leftrightarrow n - m < 1 \ or \ n + m < 1?$$

(c)

Figure 2.6: A run-time test that is conservative

of dependence, there are cases where the test assumes dependences that do not exist. For instance, for the loop in Figure 2.6.(a), Banerjee's test [7] checks whether the range $[f_{min}, f_{max}]$ of the subscript of $a()$ written in statement (1) overlaps the range $[g_{min}, g_{max}]$ of the subscript of $a()$ read in statement (2). If they do not overlap, there is definitely no dependence between statement (1) and statement (2) for loop `do i`. If they overlap, then there may be dependence. Because of the use of the symbolic terms $m$ and $n$, Banerjee's test is not able to compare the ranges at compile-time. A run-time test can be inserted to test the relationship between $n$ and $m$, as shown in Figure 2.6.(b). However, during the real execution of the loop, if only statement (1) is executed and statement (2) never gets executed because of the $if$ condition in statement (0), then there is no dependence between statements (1) and (2) even when $1 - n \leq m \leq n - 1$. Obviously, run-time information is not fully utilized in this approach.

To overcome the shortcomings of the above approach, a second approach can be used that actually executes an *inspector* version of the loop, in which the data arrays are replaced with "shadow arrays" [65]. Accesses to the data arrays are traced by marking the shadow arrays during

the execution of the inspector loop. The result of the trace is then analyzed to determine whether the loop can be parallelized. Because the array accesses are actually recorded, this second approach makes it possible to know with certainty whether there exists any data dependence or not. To minimize the overhead caused by executing the inspector loop, instead of having a separate inspector loop, the original loop usually is augmented with shadow arrays and marking codes, and the loop is executed speculatively in parallel. The execution is rolled back if the trace result indicates a dependence has been violated because of the parallel execution. This approach is more suitable for loops with irregular array access patterns.

In its straightforward form, the run-time test method in the second category uses shadow arrays that have the same size as the data arrays, and checks every read/write operation on the data array element. This method introduces high overhead on today's machines whose memory access is significantly slower than its computations.

### 2.6.3 Overhead Elimination

We found that the overhead of the run-time test in the second category could be considerably reduced by using some compiler techniques. For example, Figure 2.7 shows a simplified form of a loop from program CHOLESKY. We want to check whether loop `do i` can be parallelized. In the body of the inner loop `do j`, there are three read references and one write reference to array $data()$. We could use one mark operation for each reference. There could be four in total for each iteration of the inner loop. A compile-time value numbering analysis [18, 21, 69] can reveal that the subscript of the element read in statement (1) is the same as the subscript of the array elements written in statement (2). Therefore, the array element read in statement (1) and written in statement (2) is the same one. A dominance analysis can determine that statement (1) dominates statement (2), which means that the array element written by statement (2) is always read first by statement (1) in the same iteration. Therefore, instead of using one mark operation ("read") in statement (1) and another one ("write") in statement (2), we can simply use only one mark ("read-first-write") in statement (1). The compile-time analysis would save 25% of the mark operations (from 4 to 3) at run-time.

The number of mark operations can be further reduced by noticing that the subscript $pos(k) +$

```
do i=k+1, k+nafter(k)
   do j=1, ksize-(i-k)
      t = data(pos(i)+j-1)                                          (1)
      data(pos(i)+j-1) = t - data(pos(k)+i-k)*data(pos(k)+i+j-k-1)  (2)
   end do
end do
```

Figure 2.7: The run-time test for this loop can be improved

$i - k$ of the first read access in statement (2) is a loop-invariant expression of loop `do j`. The mark operation ("read") for array element $data(pos(k) + i - k)$ needs to be performed only once for each iteration of loop `do i`. We can move this mark operation to the front of loop `do j` and guard it by the condition *if (ksize-(i-k)) ≥ 1*. Note that it is illegal to move the read access to $data(pos(k)+i-k)$ to the front of loop `do j` without knowing there is no data dependence between this read reference and the write reference in statement (2). It is legal, however, to move the mark operation because $data(pos(i)+j-1)$ is always, as discussed above, read first during the execution.

Another method to reduce the number of mark operations is mark aggregation. Notice that statement (1) reads a continuous section of $data()$ and statement (2) reads and writes continuous sections of $data()$. Instead of marking the shadow array on every occurrence of a write or a read access, we can mark a shadow section. For each iteration of loop `do i`, we can record the type of access to two array sections (i.e., $[pos(i) : pos(i) + ksize - i + k - 1]$ and $[pos(k) + i - k : pos(k) + ksize - 1]$) rather than to each array element. Although this coarse grain section level approach may not be as accurate as the element level approach and, thus, will report false dependences, its low overhead cost justifies its application in some programs.

In our experiments, we parallelized the loop `do 1020` and loop `do 1021` of CHOLESKY by using the run-time test. We compared the overhead cost of run-time test in both the straightforward version and the version simplified by using the three mark-elimination techniques mentioned above; we found that the overhead dropped from 120% to 20% of parallel execution time on four processors.

## 2.7 Parallelization of Irregular Reduction Loops

The core of many sparse/irregular applications is comprised of reductions on array elements, such as the one shown in Figure 2.8. The operation `op` is an arithmetic associative operation (e.g., sum,

```
do i = 1, n
   a(x(i)) = a(x(i)) op expression
end do
```

Figure 2.8: An example of an irregular reduction loop

product, maximum, and minimum), and `expression` is an expression that does not contain any reference to array $a()$. Because of the use of index array $x()$, the access pattern of $a()$ in the loop may be irregular. Therefore, this kind of reduction loop is called an *irregular reduction loop*.

Most irregular reduction loops can be statically detected and automatically parallelized by a compiler. Recognizing an irregular reduction loop involves pattern matching on the array subscript expressions and knowing what kind of operations are associative/communicative. A common way to parallelize an irregular reduction loop, as used by Polaris [14] and SUIF [37], is *replicated copy*. The replicated copy method is easy to for a compiler to apply. The reduction array is replicated on all the processors. The execution of the parallelized code has three phases. In the *initialization* phase, the replicated copies are initialized to the identity of the reduction operation. In the *parallel reduction* phase, all processors execute in parallel and each processor computes a portion of the reduction on its private copy. Then, in the *cross reduction* phase, the partial reduction results are combined to the global reduction array. Figure 2.9 shows the parallelized version of the loop in Figure 2.8 using the replicated copy method.

The disadvantage of this method is that the parallelized code does not scale well with the number of processors used. The reason is that the execution time of the initialization phase and the cross processor reduction phase is proportional to the size of the reduction array, and it cannot be reduced by adding more processors. In practice, the execution time of these two phases often increases with the number of processors used due to the overhead of parallel execution, such as false sharing.

There are ways to reduce the overhead. For example, in subroutine `eflux` of program Euler, loop `do 200` and loop `do 300` are two irregular reduction loops with the same reduction array `dw()`, and these two loops are next to each other. Therefore, in the parallelized version, we eliminate the cross processor reduction phase of loop `do 200` and the initialization phase of loop `do 300`. By doing this, we reduce the parallel execution time of these two loops (four processors) by 29%.

```
/* initialization phase */
doall i = 1, nproc
      do j = 1, m
         pa(j, i) = reduction_identity
      end do
end do

/* parallel reduction phase */
doall i = 1, n
      pa(x(i), proc_id) = pa(x(i), proc_id) op expression
end do

/* cross processor reduction phase */
doall i = 1, m
      do j = 1, nproc
         a(i) = a(i) op pa(i, j)
      end do
end do
```

Figure 2.9: An irregular reduction loop parallelized by using the replicated-copy method

In the manually improved parallel version of EULER, we also parallelized several irregular reduction loops by using a pre-scheduling method. In this method, the data, namely the reduction array, is partitioned among the processors and each processor is responsible for updating the array elements assigned to it. A processor executes only the iterations that will modify the array elements belonging to the processor. A pre-scheduling phase is used to find the scheduling of the iterations. The major overhead of this method is the pre-scheduling phase. In EULER, loop `do 100` in subroutine `eflux`, loops `do 100` and `do 200` in subroutine `dflux`, and loop `do 20` in subroutine `psmoo` are irregular reduction loops that share the same access pattern of the reduction arrays. And, the access pattern does not change during the execution of the program. Hence, the pre-scheduling needs to be done only once and the schedule can be reused in all instances of these loops. By doing this, we also reduce the parallel execution time of these loops (four processors) by 24%.

In Chapter 6, we will discuss different parallelization methods for irregular reduction in detail.

## 2.8 Parallelization of Premature-exit Loops

A premature-exit loop is a do loop containing a **goto** or **break** statement that directs the program flow out of the loop before all iterations have been executed. Because it is impossible to know a priori which iterations will (or will not) be executed, a premature-exit loop makes processor scheduling difficult. For example, a loop with sixteen iterations is executed by four processors in parallel using block scheduling (i.e., processor $i$ executes iterations $(i-1)*4+1$ through $i*4$). Suppose the loop terminates at iteration seven, then iterations assigned to processors three and four should not be executed. The execution result of the loop may be incorrect if processor three or four modifies any global memory.

Generally, we can use three methods to parallelize premature-exit loops.

### 2.8.1 Inspector/Executor

In this approach, an inspector loop is executed at run-time, before the premature-exit loop, to find the range of iterations that should be executed. The inspector can be constructed by using program slicing techniques [79, 71]. The disadvantage of this approach is that the original loop body may not be easy to decouple. Even if such a separation is possible, the inspector may contain most of the computation, which makes performance improvement impossible.

### 2.8.2 Post/Wait

In this approach, the loop body is divided into two parts separated by a check for premature-exit, as illustrated in Figure 2.10.(a). The parallelized version of the loop is shown in Figure 2.10.(b). The processors are scheduled interleavely and are synchronized by using *post/wait* primitives. Here, we assume that there is no cross iteration data dependence.

A processor starts executing an iteration by waiting for the post message from the previous iteration. If the previous iteration (executed by another processor) does not premature-exit, this processor continues executing part one of the current iteration; otherwise, it sends a post message to the next iteration, and then quits the loop. After the processor checks the premature-exit condition for the current iteration, if it finds that the loop should terminate, then it sets the `terminate` variable to *true*, sends a post message to the next iteration, and quits the loop; otherwise, the loop

will not exit in this iteration, and the processor sends a post message to the next iteration and continues executing the rest of the current iteration.

Figure 2.11 illustrates an execution of a premature-exit loop using four processors. In this example, the loop premature-exits at iteration nine. The earlier the termination check occurs during the execution of the loop body, the better the performance of parallel execution.

### 2.8.3 Speculative Execution

In this approach, local copies of the variables that are going to be modified are allocated on each processor. All processors execute speculatively in parallel and write on their local copies. The contents of the local copies are committed to the global copy after the loop terminates. To implement such an execution scheme in software is difficult and complicated, in general, due to the book-keepings and cross-iterations data dependences. This approach works very well, however, in the special case where the operations on the global variables are associative.

We describe the scheme by using a premature-exit loop found in CHOLESKY as an example. In this example, shown in Figure 2.12, `reduct()` is a reduction operation, and we assume four processors are used. The parallel execution is illustrated in Figure 2.13.

1. The iteration space is divided into $m$ stages.

2. Each stage is divided in blocks with each block assigned to one processor. Processor one gets the first block, processor two gets the second block, and so on. To be concise, we assume that the iteration space can be evenly divided by the block size.

3. All processors execute in parallel. Each processor starts from stage one and executes the iterations assigned to it, beginning with the smallest iteration in its block, and then moves on to the next stage after finishing the current stage.

4. A global variable *last_iter* records the smallest iteration number currently found in which the loop will exit. The *last_iter* is initialized to $n + 1$ before the loop starts.

5. A global variable *last_stage* records the smallest stage number currently found in which the loop will exit. The *last_stage* is initialized to $m + 1$ before the loop starts.

```
Sequential Version:

do i=1, n

   /* computation part 1 */
   compute_part_1

   /* checking of premature-exit */
   if (premature-exit) then goto 10

   /* computation part 2 */
   compute_part_2

end do

10: ...
```

(a)

```
Parallelized Version:

terminate = false
post t(1)

doall i=1, n  (interleaved scheduling)

   /* sync  */
   wait t(i)
   if (terminate) then
      post t(i+1)
      quit
   end if

   /* computation part 1 */
   compute_part_1

   /* checking of premature-exit */
   if (premature-exit) then
      terminate = true
      post t(i+1)
      quit
   else
      post t(i+1)
   end if

   /* computation part 2 */
   compute_part_2

end do
```

(b)

Figure 2.10: Post/wait method for parallelizing premature-exit loop

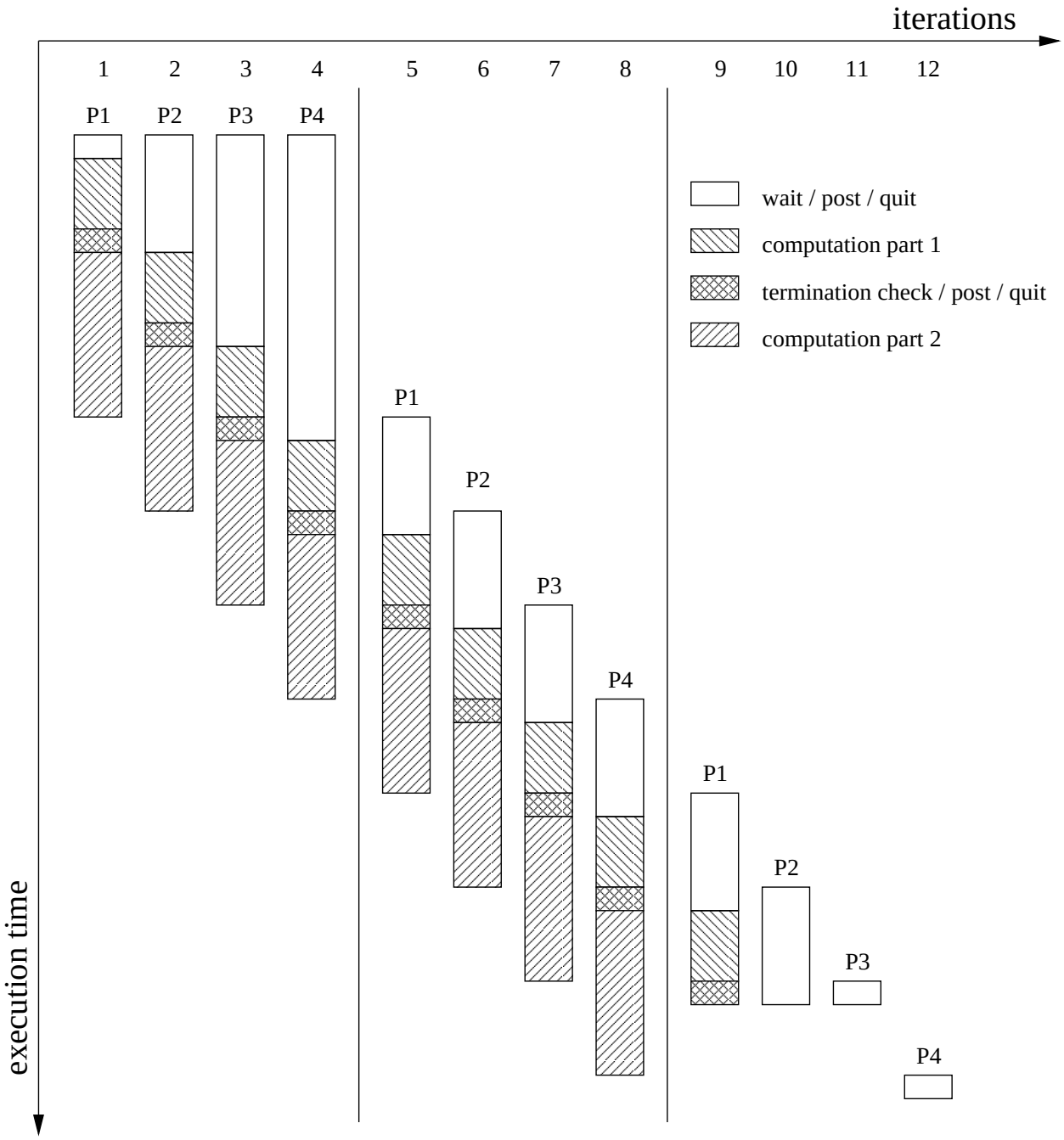



Figure 2.11: A parallel execution of a premature-exit loop using the post/wait method

```
a = initvalue
do i = 1, n
   if (cond(i)) break
   a = reduct(a,i)
end do
```

Figure 2.12: A premature-exit loop with a reduction operation in the loop body.

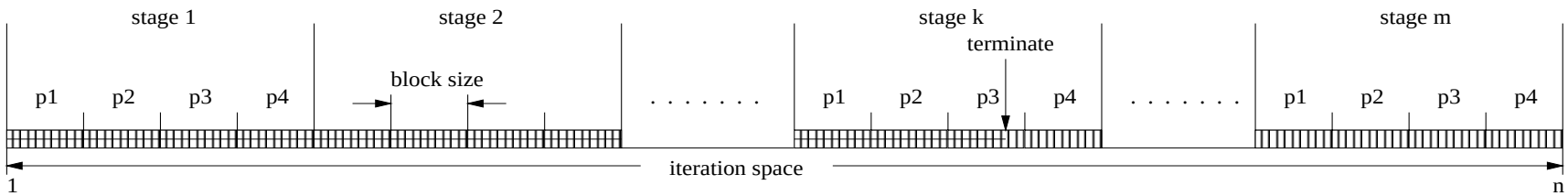



Figure 2.13: Stages in speculative execution

6. A global variable *last_proc* records the id of the processor that executes the iteration terminating the loop.

7. When a processor executes an iteration, it first compares the current iteration number with *last_iter*. If the current iteration number is larger than *last_iter*, then the processor quits executing the loop and goes to the global reduction phase.

8. If, during the execution of an iteration, a processor checks the terminate condition and finds that it is true, then the processor does the following atomically: it compares the current iteration number with *last_iter*; if the current iteration number is smaller than *last_iter*, then it sets *last_iter* to the current iteration number, *last_stage* to the current stage number, and *last_proc* to the current processor id. The processor then quits executing the loop and goes to the global reduction phase.

9. During the parallel execution phase, processors write their partial reduction results in private copies. A private copy of the variable $x$ is allocated per stage per processor. We use $x_{i,j}$ to represent the local copy of variable $x$ for processor $j$ at stage $i$. The value of all local copies are initialized to the reduction identity before the loop starts.

10. The global reduction phase starts when all processors exit the parallel reduction phase. The global reduction phase combines the value from $x_{i,j}$ ($1 \leq i < last_{stage}$, $1 \leq j \leq 4$) and $x_{last_{stage},j}$ ( $1 \leq j \leq last_{proc}$).

## 2.9 Summary

This chapter has shown that, contrary to common belief, parallelizing compilers can be used to automatically detect the parallelism in sparse and irregular programs. An empirical study of a collection of sparse and irregular codes reported in this chapter shows that several important common loop patterns exist in sparse and irregular codes. Based on these patterns, automatic parallelism detection can be applied. Some loops can be parallelized by using existing compiler techniques, while some others require new methods. The new techniques are identified and discussed. We have shown that good speedups can be achieved by applying these techniques to our collection of sparse

and irregular programs.

# Chapter 3

# ANALYSIS OF IRREGULAR SINGLE-INDEXED ARRAY ACCESS

## 3.1 Introduction

Many compiler techniques, such as loop parallelization and optimizations, need analysis of array subscripts to determine whether a transformation is legal. Traditional methods require the array subscript expressions to be expressed as closed-form expressions of loop indices. Furthermore, most methods require the subscript expression to be linear. However, in many programs, especially sparse and irregular programs, closed-form expressions of array subscripts are not available, and many codes are left unoptimized. Clearly, more powerful methods to analyze array subscripts are needed.

In this chapter, we introduce the notion of *irregular single-indexed array access*. An array access is *irregular* in a loop if no closed-form expression for the subscript of the array access in terms of loop indices is available. An array access is *single-indexed* in a loop if the array is always subscripted by the same index variable in the loop. An array access is *irregular single-indexed* in a loop if the array access is both irregular and single-indexed in the loop . For example, the access of array $x()$ in the repeat-until loop in Figure 3.1 is an irregular single-indexed access.

We chose to investigate irregular single-indexed array accesses for several reasons. First, in the programs we have studied, single-indexed array accesses often follow a few patterns. These array accesses exhibit properties that are useful in compiler optimizations. Second, many irregular array

```
i = 1
do k=1, n
   p = 0
   i = link(i,k)
   repeat
     p = p + 1
     x(p) = y(i)            (1)
     i = link(i,k)
   until ( i == 0 )
   do j=1, p
      z(k,j) = x(j)         (2)
   end do
end do
```

Figure 3.1: An example of a loop with an irregular single-indexed array

accesses are single-indexed. Developing analysis methods for irregular single-indexed array accesses is a practical approach toward the analysis of general irregular array accesses, which is believed to be difficult. Third, it is easy to check whether an array access is single-indexed. Efficient algorithms can be developed to "filter" single-indexed array accesses out of general irregular array accesses.

In this chapter, we present two important patterns of irregular single-indexed array accesses: *consecutively-written* and *stack-access*. We present the techniques to detect these two patterns and show how to use the properties that irregular single-indexed array accesses have to enhance compiler optimizations.

Throughout the rest of this chapter, we will use "irregular single-indexed array access" and "single-indexed array access" interchangeably.

## 3.2 Consecutively Written Arrays

An array is *consecutively written* in a loop if, during the execution of the loop, all the elements in a contiguous section of the array are written in a non-increasing or a non-decreasing order. For example, in the repeat-until loop in Figure 3.1, array element $x(2)$ is not written until $x(1)$ is written, $x(3)$ is not written until $x(2)$ is written, and so on. That is, array $x()$ is written consecutively in the $1, 2, 3, \ldots,$ order in the loop.

To be concise, we consider only arrays that are consecutively written in the non-decreasing

order. It is trivial to extend the techniques to handle the non-increasing case and the cases where the increment (or the decrement) is a constant value other than 1.

### 3.2.1 Algorithm for Detecting Consecutively Written Arrays

In this section, we present an algorithm that tests whether a single-indexed array is consecutively written in a loop.

Since we are dealing with irregular array accesses, we must consider not only do loops, but also other kinds of loops, such as while loops and repeat until loops. In general, we consider natural loops [2]. A natural loop has a single entry node, called the *header*. The header dominates all nodes in the loop. A natural loop can have multiple exits, which are the nodes that lead the control flow to nodes not belonging to the loop.

Before we present the algorithm, we first describe a *bounded depth-first search* (bDFS) method, which is used several times in this chapter.

The bDFS is shown in Figure 3.2. Like the standard depth-first search, a bDFS does a depth-first search on a graph $(V, E)$, where $V$ is the set of vertices and $E$ is the set of edges in the graph. bDFS uses three auxiliary functions ($f_{bound}()$, $f_{failed}()$, and $f_{proc}()$) to change its behavior during the search. The auxiliary functions are defined before the search starts. $f_{bound}()$ maps $V$ to $\{true, false\}$. Suppose the current node is $n_0$. If $f_{bound}(n_0)$ is $true$, then, bDFS does not search the nodes adjacent to $n_0$. The nodes whose $f_{bound}()$ values are $true$ are the boundaries of the search. $f_{failed}()$ also maps $V$ to $\{true, false\}$. If, for the current node $n_0$, $f_{failed}(n_0)$ is $true$, then the whole bDFS terminates with a return value of $failed$. The nodes whose $f_{failed}()$ values are $true$ cause an early termination of the bDFS. $f_{proc}()$ does not have a return value; it does predefined computations for the current node. The running time of bDFS is $O(|V| + |E|)$.

Now we can show the algorithm that detects consecutively written arrays.

- **Input:** a loop $L$ with header $h$ and a set of exit nodes $\{t_1, t_2, ..., t_n\}$, a single-indexed array $x()$ in the loop, and the index variable $p$ of $x()$.
- **Output:** the answer to the question whether $x()$ is consecutively written in $L$.
- **Steps:**

```
bDFS(u)
    visited[u] := true ;
    f_proc(u) ;
    if (not f_bound(u)) {
        for each adjacent node v of u {
            if (f_failed(v))
                return failed ;
            if ((not visited[v]) and (bDFS(v) == failed))
                return failed ;
        }
    }
    return succeeded ;
```

Before the search starts, $visited[]$ is set to $false$ for all nodes.

Figure 3.2: Bounded depth-first search

1. Find all the definition statements of $p$ in the loop. If any of them are not of the form "$p = p + 1$", then return $NO$. Otherwise, put the definition statements in a list $lst$.

2. For each statement $n$ in $lst$, do a bDFS on the control flow graph from $n$ using the following auxiliary functions:

$$f_{bound}(n) = \begin{cases} true, & \text{if } n \text{ is "}x() = ..\text{"} \\ false, & \text{otherwise} \end{cases}$$

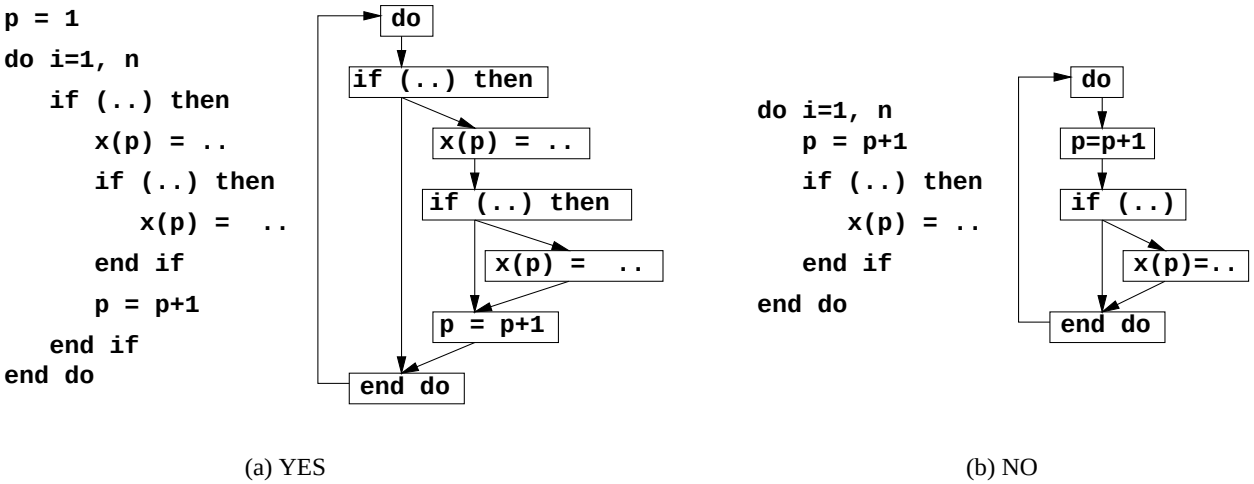
$$f_{failed}(n) = \begin{cases} true, & \text{if } n \text{ is "}p = p + 1\text{"} \\ false, & \text{otherwise} \end{cases}$$

$$f_{proc}(n) = NULL$$

If any of the bDFSs returns $failed$, then return $NO$. Otherwise, return $YES$.

The algorithm starts by checking whether the index variable is ever defined in any way other than being increased by 1. If it is, we assume the array is not consecutively written. Step 2 checks whether in the control flow graph there exists a path from one "$p = p + 1$" statement to another "$p = p + 1$" statement[1] and the array $x()$ is not written on the path. If such a path exists, then there may be "holes" in the section where the array is defined and, therefore, the array is not consecutively written in the section. For example, the array $x()$ is consecutively written in Figure

[1]These two statements can be the same statement, in which case the path is a circle.




(a) YES (b) NO

Figure 3.3: Consecutively written or not?

3.3.(a), but is not in Figure 3.3.(b). The algorithm allows an array element to be written multiple times before the index variable is increased by 1.

The running time of this analysis is $O(n+n_{inc}*n_{assign})$, where $n$ is the number of statements in the loop body, $n_{inc}$ is the number of "p=p+1" statements, and $n_{assign}$ is the number of "x(p)=.." statements. When the analysis finds $x()$ to be a consecutively written array, each of the bDFS's performed searched a distinct partition in the control flow graph. The paritions are separated by the "x(p)=.." statements. When the analysis finds $x()$ not to be a consecutively written array, one bDFS has visited $O(n)$ statements, and the previous bDFS's were performed on their own partitions.

### 3.2.2 Applications

#### Dependence Test and Parallelization

Suppose a single-indexed write-only array $x()$ with index variable $p$ is consecutively written in a loop, where the assignments of $p$ are of the form "$p = p + 1$". If there does not exist a path from one "$x() = ..$" assignment to another "$x() = ..$" assignment such that the loop header is on the path, but there is no "$p = p + 1$" statement on the path, then $x()$ does not cause any loop-carried dependence in the loop. For example, in Figure 3.4, array $x()$ is consecutively written in both loop `do i` and loop `do j`. In loop `do i`, there is no dependence between different instances of the access of $x()$. In loop `do j`, because statement (2) and statement (1) may write to the same array element

```
do i=1, n                          do j=1, n
   x(p) = ..                          x(p) = ..     (1)
   p = p + 1                          p = p + 1
end do                                x(p) = ..     (2)
                                   end do
```

Figure 3.4: Data dependence for consecutively written arrays

in two different iterations, there is a loop-carried output dependence between statement (2) and (1).

This kind of dependence can be detected by using the following method. Here, we assume $x()$ is write-only and found consecutively written with the method described in the previous section.

1. Using the following auxiliary functions, do a bDFS on the control flow graph from the loop header, where the value of $tag1$ is initially set to $null$:

$$f_{bound}(n) = \begin{cases} true, & \text{if } n \text{ is "}x()\text{=..." or "}p = p + 1\text{"} \\ false, & \text{otherwise} \end{cases}$$

$$f_{failed}(n) = \begin{cases} true, & \text{if } tag1 \text{ is } asgn \\ false, & \text{otherwise} \end{cases}$$

$$f_{proc}(n) = \begin{cases} \text{set } tag1 \text{ to } asgn, \text{ if } n \text{ is "}x() = ..\text{"} \\ \text{set } tag1 \text{ to } incr, \text{ if } n \text{ is "}p = p + 1\text{" and } tag1 \text{ is } null \\ \text{do nothing, otherwise} \end{cases}$$

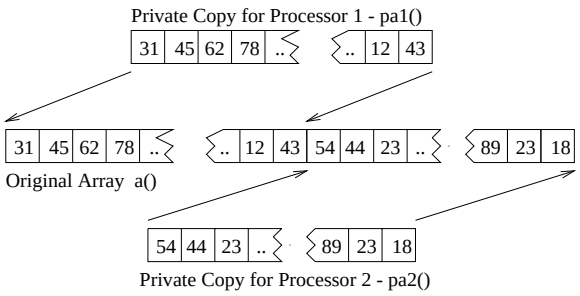
   If $tag1$ is $incr$ after the bDFS, then there is no dependence; otherwise, goto step 2.

2. Using the same auxiliary functions as in the previous step, do a bDFS on the reversed control flow graph from the loop header, with $tag1$ being replaced with $tag2$. If, after the bDFS, both $tag1$ and $tag2$ are $asgn$, then there is loop-carried output dependence for $x()$; otherwise, there is no such dependence.

In order to parallelize the loop with single-indexed and consecutively written arrays, we also need to eliminate the flow dependence caused by the index variable. If the index variable is not

```
Sequential version:
      k = k0
      do i = 1, n
         while (..) do
               a(k) = ..
               k = k+1
         end while
      end do
```

(a)




(b)

```
Parallel version:
    pk(1) = 1
    pk(2) = 1
    parallel do i = 1, n
      /* pid is the processor id */
      while (..) do
        pa(pk(pid), pid) = ..
        pk(pid) = pk(pid) + 1
      end while
    end do
    parallel section
      do i = 1, pk(1)-1
         a(k0+i-1) = pa(i,1)
      end do
    //
      do i = 1, pk(2)-1
         a(k0+pk(1)+i-2) = pa(i,2)
      end do
    end parallel section
    k = k0+pk(1)+pk(2)-2
```

(c)

Figure 3.5: An example of *array splitting and merging*

used anywhere other than in the array subscript and the increment-by-1 statements, then the *array splitting-and-merging* method [49] can be used to parallelize the enclosing loop.

Array splitting and merging consists of three phases. First, a private copy of the consecutively written array is allocated on each processor. Then, all the processors work on their private copies from position 1 in parallel. After the computation, each processor knows the number of array elements of its private copy that are written in the loop; hence, the starting position in the original array for each processor can be calculated by using the parallel prefix method. Finally, the private copies are copied back (merged) to the original array. Figure 3.5 shows an example when two processors are used.

### Privatization Test

As we have illustrated at the beginning of this chapter, with consecutively written array analysis. We can extend the privatization test to process irregular single-indexed arrays and more general

loops.

Suppose a single-indexed array $x()$ with index variable $p$ is found consecutively written in a loop by using the method described in the previous section, we can use the following two steps to calculate the section of $x()$ written in the loop.

1. Using the following auxiliary functions, do a bDFS on the control flow graph from the loop header $h$, where the value of $tag1$ is initially set to $null$:

$$f_{bound}(n) = \begin{cases} true, & \text{if } n \text{ is "}x() = ..\text{" and } tag1 \text{ is } asgn \\ true, & \text{if } n \text{ is "}p = p + 1\text{" and } tag1 \text{ is } incr \\ false, & \text{otherwise} \end{cases}$$

$$f_{failed}(n) = \begin{cases} true, & \text{if } n \text{ is "}x() = ..\text{" and } tag1 \text{ is } incr \\ true, & \text{if } n \text{ is "}p = p + 1\text{" and } tag1 \text{ is } asgn \\ false, & \text{otherwise} \end{cases}$$

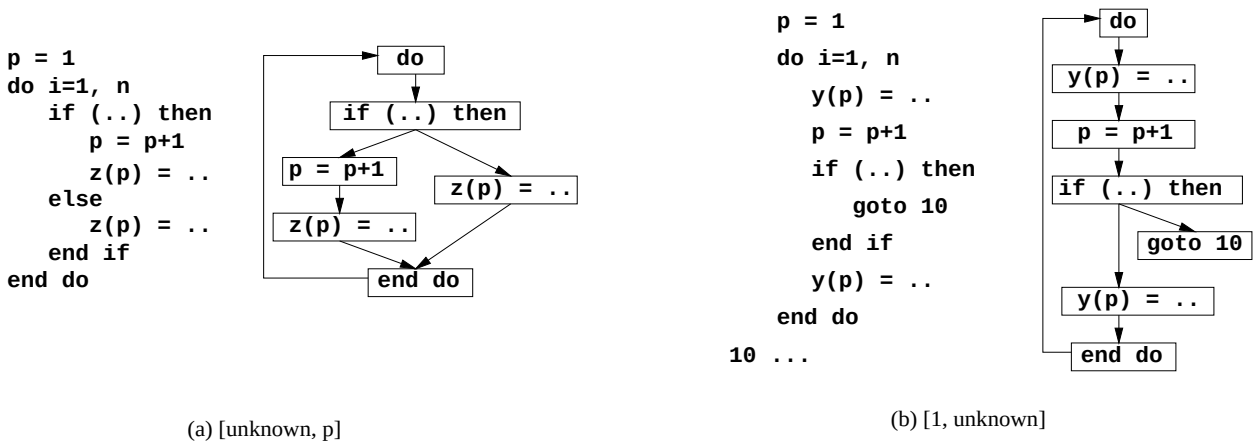
$$f_{proc}(n) = \begin{cases} \text{set } tag1 \text{ to } asgn\text{, if } n \text{ is "}x() = ..\text{" and } tag1 \text{ is } null \\ \text{set } tag1 \text{ to } incr\text{, if } n \text{ is "}p = p + 1\text{" and } tag1 \text{ is } null \\ \text{do nothing, otherwise} \end{cases}$$

   If the bDFS returns a $failed$, then set $tag1$ to $null$.

2. Using the same auxiliary functions as in the previous step, do a bDFS on the reversed control flow graph from each of the exit nodes (including the loop header), with $tag1$ being replaced with $tag2$. If any of the bDFSs returns a $failed$, then set $tag2$ to $null$.

3. The section where $x()$ is written in the loop is $[lower, upper]$, where

$$lower = \begin{cases} p_0, & \text{if } tag1 \text{ is } asn \\ p_0 + 1, & \text{if } tag1 \text{ is } incr \\ unknown, & \text{otherwise,} \end{cases}$$




(a) [unknown, p]

(b) [1, unknown]

Figure 3.6: The section of consecutively written array

$$upper = \begin{cases} p, & \text{if } tag2 \text{ is } asn \\ p-1, & \text{if } tag2 \text{ is } incr \\ unknown, & \text{otherwise.} \end{cases}$$

and $p_0$ is the value of $p$ before entering the loop.

For example, the section of $x()$ written in the loop in Figure 3.3.(a) is $[1, p-1]$. The section of $z()$ written in the loop in Figure 3.6.(a) is $[unknown, p]$, and that of $y()$ in Figure 3.6.(b) is $[1, unknown]$.

### Index Array Property Analysis

As described in Chapter 2, the *indirectly accessed array* is another kind of irregular array. An array is indirectly accessed if its subscript is another array, such as $x()$ in statement "$x(ind(i)) = ..$". $x()$ is called the *host array*, and $ind()$ is called the *index array*. Traditional techniques cannot handle indirectly accessed arrays. However, recent studies [13, 49] have shown that index arrays often have simple properties which can be used to produce more accurate analysis of host arrays. An *array property analysis* method has been developed to check whether an index array has any of these key properties [50]. We will discuss array property analysis in detail in Chapter 4.

Consecutively written array analysis can be used to find the properties an index array has in array property analysis. For example, two of the key properties are *injectivity* and *closed-form bounds*. An array section is injective if any two different array elements in the section do not have

```
do k = 1, n
   q = 0
   do i = 1, p
      if ( x(i) > 0 ) then
         q = q + 1
         ind(q) = i
      end if
   end do
   do j = 1, q
      jj = ind(j)
      z(k,jj) = x(jj) * y(jj)
   end do
end do
```

Figure 3.7: An example of a loop with an inner index gathering loop

the same value. An array section has closed-form bounds if the lower bound and upper bound of the values of array elements in the section can be expressed by closed-form expressions. Detecting whether an array section has any of the two properties is difficult, in general. However, in many cases, we only need to check whether the array section is defined in an *index gathering loop*, such as the `do i` loop in Figure 3.7.

In Figure 3.7, the indices of the positive elements of array $x()$ are gathered in array $ind()$ in loop `do i`. After the gathering loop is executed, all the array elements in section $ind[1:q]$ are defined; the values of the array elements in array section $ind[1:q]$ are injective; the lower bound of the values of the array elements in section $ind[1:q]$ is 1; and, the upper bound is $q$.

With this information available at compile-time, the compiler is now able to determine that there is no data dependence in the `do j` loop, and array $ind()$ can be privatized in the `do k` loop. Thus, depending upon the architecture for which the code is generated, the compiler can choose to parallelize the `do k` loop only, parallelize the `do j` loop only, parallelize both, or parallelize the `do k` loop and vectorize the `do j` loop.

An index gathering loop for an index array has the following characteristics:

1. the loop is a do loop,

2. the index array is single-indexed in the loop,

3. the index array is consecutively written in the loop,

```
do i = 1,  n
   p = 1
   t(p) = ...
   loop
      p = p + 1
      t(p) = ...
      if (...) then
         loop
         if (p>=1) then
            ... = t(p)
            p = p - 1
         end if
         end loop
      end if
   end loop
end do
```

Figure 3.8: An example of an array stack

4. the right-hand side of any assignment to the index array is the loop index, and

5. one assignment to the index array cannot reach another assignment to the index array without first reaching the do loop header.

The fifth condition above ensures that the same loop index value is not assigned twice to the elements of the index array. This condition can be verified using a bDFS. After an index gathering loop, the values assigned to the index array in the loop are injective, and the range of the values assigned is bounded by the range of the do loop bound.

## 3.3 Array Stacks

The stack is a very basic data structure. Many programs implement stacks using arrays because it is both simple and efficient. We call stacks implemented in arrays *array stacks*. Figure 3.8 illustrates an array stack. In the body of the `do i` loop, array $t()$ is used as a stack, and variable $p$ is used as the stack pointer which always points to the top of the stack.

| | $p = p + 1$ | $p = p - 1$ | $x(p) = ..$ | $.. = x(p)$ | $p = C_{bottom}$ |
|---|---|---|---|---|---|
| $p = p + 1$ | $x(p) = ..$ | $.. = x(p)$ | - | $x(p) = ..$ | - |
| $p = p - 1$ | - | $.. = x(p)$ | $p = p{+}1$ | G | - |
| $x(p) = ..$ | - | $.. = x(p)$ | $p = p + 1$ | - | - |
| $.. = x(p)$ | $p = p - 1$ | - | $p = p + 1$ | $p = p + 1$ | - |

Table 3.1: Order for access of array stacks

### 3.3.1 Algorithm for Detecting Array Stacks

In this section, we present an algorithm that checks whether a single-indexed array is used as a stack in a program region. A region [2] is a subset of the control flow graph that includes a header, which dominates all the other nodes in the region.

To be concise, we consider program regions in which the single index variable $p$ is defined only in one of the following three ways:

1. $p := p + 1$,

2. $p := p - 1$, or

3. $p := C_{bottom}$, where $C_{bottom}$ is invariant in the program region.

We check whether a single-indexed array is used as a stack in a region by determining if the statements involved in the array operations appear in some particular orders. These orders are shown in Table 3.1.

The left column and the top row in Table 3.1 give the statements to be checked. If there is a path in the control flow graph from a statement of the form shown in the left column of the table to a statement of the form shown in the top row, then the statement in the corresponding central entry of the table must be on the path. For example, if there is a path from a statement "$x(p) = ..$" to another statement "$x(p) = ..$", then before the control flow researches the second "$x(p) = ..$" statement, it must first reach a "$p = p + 1$" statement. A '-' in a table entry means there is no restriction on what kind of statement must be on the path. The 'G' represents an $if$ statement that is "$if\ (p \geq C_{bottom})\ then$".

Intuitively, we want to ensure the following for an array stack $x()$ with index $p$: (1) $p$ is first set to $C_{bottom}$ before it is modified or used in the subscript of $x()$; (2) the value of $p$ never goes below

| $n$ | $S_{bound}(n)$ | $S_{failed}(n)$ |
|---|---|---|
| $p = p + 1$ | $\{x(p) = .., p = C_{bottom}\}$ | $\{p = p + 1, p = p - 1, .. = x(p)\}$ |
| $p = p - 1$ | $\{p = p + 1, \text{G}, p = C_{bottom}\}$ | $\{p = p - 1, x(p) = .., .. = x(p)\}$ |
| $x(p) = ..$ | $\{p = p + 1, .. = x(p), p = C_{bottom}\}$ | $\{p = p - 1, x(p) = ..\}$ |
| $.. = x(p)$ | $\{p = p - 1, p = C_{bottom}\}$ | $\{p = p + 1, x(p) = .., .. = x(p)\}$ |

Table 3.2: Simplified order for array stacks

$C_{bottom}$; and (3) the accesses of the elements of $x()$ follow the "last-written-first-read" pattern.

Table 3.1 can be simplified to Table 3.2. Any path originating from a node $n$ of the forms in the left column of Table 3.2 must first reach any node of the forms in $S_{bound}(n)$ before it reaches any node of the forms in $S_{failed}(n)$.

Next, we present the algorithm to detect array stacks.

- **Input:** a program region $R$ with header $h$, a single-indexed array $x()$ in the region, and the index variable $p$ of $x()$.
- **Output:** the answer to the question whether $x()$ is used as a stack in $R$. And, if the answer is $YES$, the minimum value $C_{bottom}$ that the index variable $p$ can have in the region.
- **Steps:**
  1. Find all the definition statements of $p$ in $R$. If any are not of a form in the set $\{p = p+1, p = p-1, p = C_{bottom}\}$ (if there are multiple "$p = C_{bottom}$" statements, the $C_{bottom}$ must be the same), where $C_{bottom}$ is invariant in $R$, then return $NO$. Otherwise, put the definition statements in a list $lst$. If there are no statements of the form "$p = C_{bottom}$", then find all $if$ statements of the form "$if\ (p \geq C_{if})\ then$". If all $C_{if}$'s are the same, set $C_{bottom}$ to $C_{if}$; otherwise, return $NO$. If no such $if$ statement is found, set $C_{bottom}$ to $unknown$.
  2. Find all the "$x(p) = ..$" and "$.. = x(p)$" statements in $R$, and add them to $lst$.
  3. For each statement $m$ in $lst$, do a bDFS on the control flow graph from this statement using the following auxiliary functions:

$$
\begin{aligned}
f_{bound}(n) &= \begin{cases} true & n \in S_{bound}(m) \\ false & \text{otherwise} \end{cases} \\
f_{failed}(n) &= \begin{cases} true & n \in S_{failed}(m) \\ false & \text{otherwise} \end{cases} \\
f_{proc}(n) &= NULL
\end{aligned}
$$

If any of the bDFSs returns a $failed$, then return $NO$. Otherwise, return $YES$ and $C_{bottom}$.

### 3.3.2 Applications

**Run-time Array Bound Checking Elimination**

Run-time array bound checking is used to detect array bound violations. The compiler inserts bound checking codes for array references. At run-time, an error is reported if an array subscript expression equals a value that is not within the declared bounds of the array. Some languages, such as Pascal, Ada and Java, mandate array bound checking. Array bound checking also is useful in testing and debugging programs written in other languages. Since most references in computationally intense loops are to arrays, these checks cause a significant amount of overhead.

When an array is used as a stack in a program region, the amount of array bound checking for the stack array can be reduced by 50%. Only the upper bound checkings are preserved. The lower bound checking is performed only once before the header of the program region. Elimination of unnecessary array bound checking also has been studied by many other researchers, including Markstein et al [52], Gupta [35], and Kolte and Wolfe [45]. Gupta and Spezialetti [70] proposed a method to find monotonically increasing/decreasing index variables, which also can be used to eliminate the checking by half. But, their method cannot handle array stacks, which are more irregular.

### Privatization Test

Array stack analysis also can improve the precision of array privatization tests. Here, we consider the loop body as a program region. When an array is used as a stack in the body of a loop, the array elements are always defined ("pushed") before being used ("popped") in the region. If $C_{bottom}$ is a loop invariant, then different iterations of the loop will reuse the same array elements, and the value of the array elements never flow from one iteration to the other. Therefore, array stacks in a loop body can be privatized. For example, the array stack $t()$ in Figure 3.8 can be privatized in the outermost the `do i` loop.

### Loop Interchanging

Loop interchanging [3, 80] is the single most important loop restructuring transformation. It has been used to find vectorizable loops, to change the granularity of parallelism, and to improve memory locality. Loop interchanging changes the order of nested loops. It is not always legal to perform loop interchanging since data dependence cannot be violated. Data dependence tests must be performed before loop interchanging.

Traditionally, loop interchanging is not possible when array stacks are present because current data dependence tests cannot handle irregular arrays. However, as in the privatization test, array stacks cause no loop carried dependences. If the index variables of array stacks are not used in any statements other than stack access statements, then the data dependence test can safely assume no dependence between the stack access statements. The loop interchanging test then can ignore the presence of array stacks and use traditional methods to test other arrays. By using array stack analysis, we have extended the application domain of loop interchanging.

## 3.4 Related Work

There are two closely related studies done by two groups of researchers. M. Wolfe [82] and M. Gerlek, E. Stoltz, and M. Wolfe [32] presented an algorithm to recognize and classify *sequence variables* in a loop. Different kinds of sequence variables are linear induction variables, periodic, polynomial, geometric, monotonic, and wrap-around variables. Their algorithm is based on a demand-driven

```
do i = 1, n
   if ( .. ) then
      x(k) = ..
      k = k + 1                    (1)
   else if ( .. ) then
      x(k) = ..
      k = k + 1                    (2)
   end if
end do
```

Figure 3.9: Both array $x()$ and index $k$ should be analyzed to know that $x()$ is consecutively written.

representation of the Static Single Assignment form [24, 23]. The sequence variables can be detected and classified in a unified way by finding strongly connected components of the associated SSA graph.

R. Gupta and M. Spezialetti [70] have extended the traditional data-flow approach to detect "monotonic" statements. A statement is monotonic in a loop if, during the execution of the loop, the statement assigns a monotonically increasing or decreasing sequence of values to a variable. They also show the application of their analysis in run-time array bound checking, dependence analysis, and run-time detection of access anomalies.

The major difference between both these studies and ours is that we focus on arrays while they focus on index variables. While both of their methods can recognize the index variable for a consecutively written array as a monotonic variable, if the array is defined in more than one statement, then none of them can detect whether the array itself is consecutively written. For example, Gerlek, Stoltz and Wolfe's method can find that the two instances of variable $k$ in statements (1) and (2) in Figure 3.9 have a strictly increasing sequence of values. Gupta and Spezialetti's method can classify statements (1) and (2) as monotonic. However, neither can determine whether the access pattern of the array $x()$ is consecutively written. For array stack analysis, as the index variable does not have a distinguishable sequence of values, both Gerlek, Stoltz and Wolfe's method and Gupta and Spezialetti's method treat the index variable as a generally irregular variable. Without taking the arrays into account in their analysis, they can do little in detecting array stacks.

The authors believe it is often important to consider both index variables and arrays. While both of the two other methods can recognize a wide class of scalar variables beyond the variables

used as subscripts of single-indexed arrays in our method, they are not necessarily more powerful in analyzing the access pattern of the arrays.

## 3.5 Summary

In this chapter, we introduced the notion of irregular single-indexed array access. We described two common patterns of irregular single-indexed array accesses (i.e., consecutively written and stack access) and presented simple and intuitive algorithms to detect these two patterns. More importantly, we showed that array accesses following these two patterns exhibit very important properties. We demonstrated how to use these properties to enhance a variety of compiler analysis and optimization techniques, such as the dependence test, privatization test, array property analysis, loop interchanging, and array bound checking.

In Chapter 5, we will show that, for three real-life programs, the speedups of the parallelized versions generated by the Polaris compiler with single-index array access analysis are much better than those of the versions generated by Polaris without this analysis and those versions generated by the SGI MISPro Fortran compiler.

# Chapter 4

# DEMAND-DRIVEN INTERPROCEDURAL ARRAY PROPERTY ANALYSIS

## 4.1 Introduction

Most loop optimization techniques rely on the analysis of array subscripts. When arrays are subscripted by index arrays[1], current compilers are forced to make conservative assumptions and give up optimizations or apply run-time methods at the cost of significant execution overhead. Index arrays are used extensively in sparse computation codes and also appear in some dense and regular programs. In order to optimize these programs, more aggressive compile-time methods to analyze the index arrays are desired. Having more precise information on index arrays at compile-time not only can enable more optimizations and transformations, but also can lead to more efficient run-time methods.

The compile-time analysis of index arrays has been facilitated by two recent developments. First, recent empirical studies of real programs [13, 49] have shown that the use of index arrays often follows a few patterns. By identifying these patterns at compile-time, more precise analysis can be accomplished. Second, recent progress in interprocedural analysis [22, 38, 42, 48], array data flow analysis [33, 29, 34, 64], and demand-driven approaches [26, 73] have enabled the more efficient and more powerful whole program analysis required by index array analysis.

In this chapter, we present an index array analysis method, called *array property analysis*. By

[1]We call the array that appears in the subscripts of other arrays the *index array* and the indirectly accessed array the *host array*.

performing a whole program analysis, array property analysis can automatically identify whether an index array possesses one of several key properties. These key properties are used in the analysis of the subscripts of the host array to produce more accurate analysis results. We discuss these key properties and show how to use a demand-driven approach to do whole program analysis.

This chapter is organized as follows. Section 4.2 describes the motivations of this work and defines array property analysis. Sections 4.3 and 4.4 detail our demand-driven interprocedural analysis method. Section 4.5 gives a mechanism that uses a run-time test to verify array properties. Section 4.6 describes some related work that inspired our analysis. And, Section 4.7 summarizes the chapter.

## 4.2 The Problem

### 4.2.1 Array Property Analysis

The use of index arrays is a major obstacle to accurate array subscript analysis. For example, when detecting the parallelism of a loop, a dependence test must be conservative in the sense that if it is possible for two references, one of which is a write, to access the same memory location, then their original access order must be preserved.

For instance, when examining the following loop, the access order of $a()$ is unknown since the value of index array $b()$ is unknown. Thus, output dependences have to be assumed.

```
do i=1, n
   a(b(i)) = ...
end do
```

While it is true that the values of index arrays usually are not available until run-time, oftentimes we can get more precise results if global program analysis is performed. In their study of the Perfect Benchmarks, Blume and Eigenmann found that index arrays usually had some detectable properties [13]. By knowing these properties, compilers often can avoid making conservative assumptions. Results similar to those of Blume and Eigenmann also were obtained by the authors in a study of several sparse and irregular programs [49].

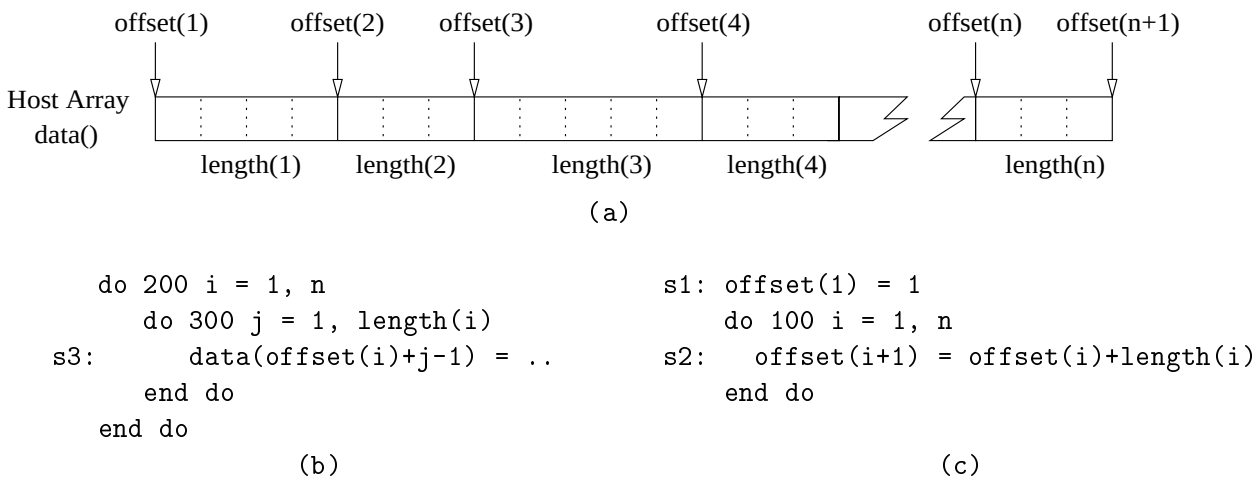



Figure 4.1: Example of closed-form distance

In sparse and irregular programs, index arrays are extensively used. However, the use of index arrays is by no means arbitrary. In fact, it tends to have a few fixed patterns. For example, in the sparse matrix computations based on the Compressed Column Storage(CCS) or Compressed Row Storage(CRS) format, the non-zero elements of the matrix are stored in a one-dimensional host array. The host is divided into several segments, as illustrated in Figure 4.1(a). Each segment corresponds to a column (in CCS) or a row (in CRS). Two index arrays are used here. Index array `offset()` points to the starting position of each segment, and index array `length()` gives the length of each segment. Figure 4.1(b) shows a common loop pattern using the `offset()` and `length()` arrays. The loop traverses the host array segment by segment. Figure 4.1(c) shows a common pattern used to define `offset()`. It is easy to see that loop `do 200` does not carry any dependences if $length(i) \geq 0$ $(1 \leq i \leq n)$, because

$$offset(i) + length(i) - 1 < offset(i+1), \text{where } 1 \leq i < n.$$

This is guaranteed by the fact that

$$offset(i+1) - offset(i) = length(i), \text{where } 1 \leq i \leq n$$

which can be derived from Figure 4.1(c).

Given an array, if the difference of the values of any two consecutive elements can always be

represented by a closed-form expression, we say the array has a closed-form *distance*[17]. For example, the array `offset()` in Figure 4.1 has a closed form distance, which is `length()`. If the compiler knows an index array has a closed-form distance, then it can use this distance information in the analysis of the subscript of the host array. Having a closed-form distance is just one of the five properties of index arrays that can be exploited by the compiler. The other four are:

- Injectivity

  An array $a()$ is injective if $a(i) \neq a(j)$ when $i \neq j$.

  If the array `b()` shown in the loop in the second paragraph of this section is injective, then there is no output dependence due to `a()`, and the loop can be parallelized.

- Monotonicity

  An array $a()$ is monotonically non-decreasing if $a(i) \leq a(j)$, $i < j$. It is monotonically non-increasing if $a(i) \geq a(j)$, $i < j$.

  The following code shows another common pattern of accessing the host array `data()` in Figure 4.1.

```
     do 400 i = 1, n
        do 500 j = offset(i), offset(i+1)-1
s4:        data(j) = ..
        end do
     end do
```

  If array `offset()` is monotonically non-decreasing, then the outer loop can be parallelized.

- Closed-form Value

  An array has a closed-form value if all the values of the array elements can be represented by a closed-form expression at compile-time.

- Closed-form Bound

  An array has a closed-form bound if closed-form expression is available at compile-time for either the lower bound or the upper bound of the values of the elements in this array.

In order to get more precise results when analyzing the subscripts, we would like to know if the index arrays have any of the above properties. This can be described as an *available property* problem. A property of a variable $x$ is *available* at a point $p$ if the execution of the program along any path (not necessarily cycle-free) from the initial node to $p$ guarantees that $x$ has this property. Although the available property problem is undecidable in general, there are simple detection algorithms that are sufficient in most cases.

In real programs, the index arrays are seldom modified once they are defined. Based on this observation, we can take the following approach: when we analyze an indirectly accessed array reference, we check all the definitions of the index arrays that reach the reference. We examine the program pattern at each definition site. If all the definition sites imply that the index array has any of the properties, and none of the statements in between the definition sites and the use site redefines any variables that are used to express the property, then we say that the property is available at the use site. Otherwise, we assume it is not available.

Before we describe our method, there are two other issues to be addressed. First, we must do the analysis interprocedurally. Index arrays are usually used to store data structure information. Most real programs read the input data and construct the data structure in one procedure, and then perform the major numerical computation based on the data structure in another procedure. This means index arrays are often defined in one procedure and used in other procedures. In such cases, the reaching definitions can be found only by doing the analysis interprocedurally.

Second, we want the analysis to be demand-driven. It can be expected that the cost of interprocedural array reaching definition analysis and property checking can be high. However, only the arrays that are used as index arrays require this analysis. Thus, instead of checking the property at all array definition sites and propagating the property exhaustively throughout the program, we choose to apply the analysis on demand. We perform the analysis only when we meet an index array. And, heuristically, we check only the property that the use site suggests. For example, a use site like `s3` in Figure 4.1 indicates that the distance of `offset()` is `length()`. Thus, in this case, we only need to check the property of closed-form distance.

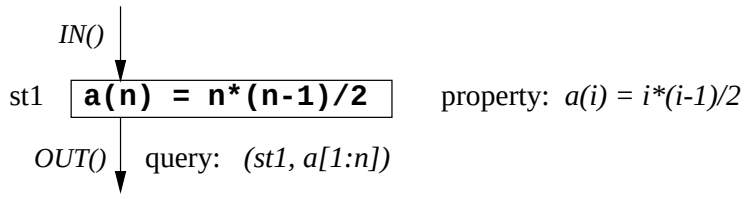



Figure 4.2: A sample query

### 4.2.2 Dataflow Model

We model our demand-driven analysis of the available property as a query propagation problem [26]. A query is a tuple $(st, section)$, where $st$ is a statement and $section$ is an array section[2] for the index array. Given an index array and a property to check, a query $(st, section)$ raises the question whether the index array elements in $section$ always have the desired property when the control reaches the point after $st$. For example, the query $(st1, a[1 : n])$ in Figure 4.2 asks whether $a(i) = (i - 1) * i/2$ for $1 \leq i \leq n$ after statement $st1$.

A query is propagated along the reverse direction of the control flow until it can be verified to be $true$ or $false$. Let

- $OUT(S)$ be the section of index array elements to be examined at statement $S$,
- $GEN(S)$ be the section of the index array elements that possess the desired property because of the execution of statement $S$,
- $KILL(S)$ be the section of the index array elements that are verified not to have the property because of the execution of $S$,
- $IN(S)$ be the section of the index array elements that cannot be determined to possess the property by examining $S$ and, thus, should be checked again at the predecessors of $S$, and
- $DES(S)$ be the section of index array elements in $OUT(S)$ that are verified not to have the property because of the execution of $S$.

[2] An array section can be represented as either a convex region [72], an abstract data access [6, 58], or a regular section [41]. Our method is orthogonal to the representation of the array section. Any representation can be used as long as the aggregation operation in Sect. 4.3.6 is defined.

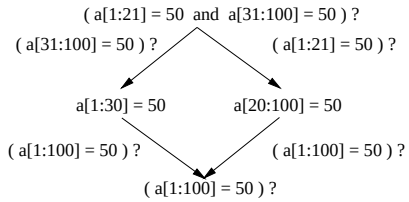



Figure 4.3: Query sections are unioned

The general dataflow equations for the reverse query propagation are

$$
\begin{aligned}
OUT(S) &= \bigcup_{T \text{ is a successor of } S} IN(T) \\
IN(S) &= OUT(S) - GEN(S) \\
DES(S) &= OUT(S) \cap KILL(S)
\end{aligned}
$$

And, for a property query $(st, section)$, initially, $OUT(st) = section$ and $OUT(s) = \emptyset$ for all statements $s$ other than $st$.

When a statement has multiple successors, the query section to be analyzed is the union of the query sections propagated from its successors. This is important because a property is available at a point only when it is available at all paths coming to this point. If it cannot be verified along all branches, it must be checked again at the branch node, as illustrated in Figure 4.3.

If, after the propagation finishes, we have $IN(entry) \neq \emptyset$, where $entry$ is the entry statement of the program, or there exists a statement $s$ such that $DES(s) \neq \emptyset$, then the answer to the original query is $false$. Otherwise the answer is $true$. $IN(entry) \neq \emptyset$ means that some elements in the original $section$ are not defined along some path from the program entry to statement $st$; thus, not all the elements in $section$ have the desired property. As a result, the answer is $false$ in this case. If, for some statement $s$, we have $DES(s) \neq \emptyset$, then some element in the original $section$ has been found not to have the desired property because of the execution of $s$. Hence, in either case, the answer to the query also is $false$.

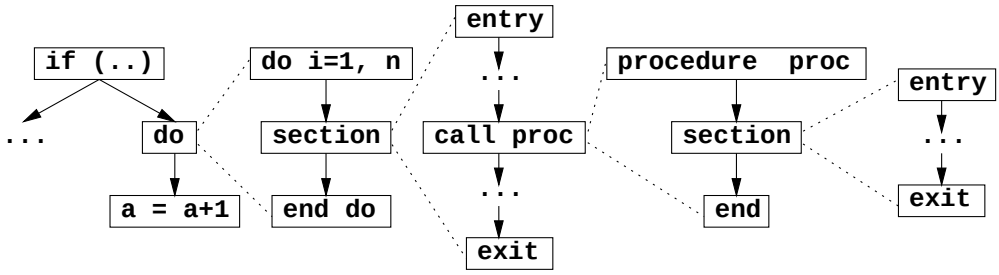



Figure 4.4: An HCG example

## 4.3 The Method

### 4.3.1 Program Representation

We represent the program in a hierarchical control graph (HCG), which is similar to the *hierarchical supergraph* in [34]. Each statement is represented by a statement node. In addition, each loop is represented by a loop node, and each procedure is represented by a procedure node. There also is a section node for each loop body and each procedure body. Each section node has a single entry node and a single exit node. Due to the nested structure of loops and procedure calls, a hierarchy is derived among the HCG nodes. There is a directed edge from one node to the other at the same hierarchical layer if the program control can transfer from one to the other. Figure 4.4 shows an HCG example. We assume the only loops in the program are `do` loops, and we deliberately delete the back edge from the `end do` node to the `do` node. Hence, the HCG is a hierarchical directed acyclic graph.

### 4.3.2 Overview of The Method

Our method consists of three parts, as shown in Figure 4.5. The *QueryGenerator* is incorporated in the data dependence test, the array privatization test, or any other test method that requires the detailed analysis of index arrays. The *QueryGenerator* issues a query when the test wants to verify if an index array has a certain property at a certain point. The *QueryGenerator* also specifies what kind of property to be checked. The *QueryChecker* accepts the query and then uses *QuerySolver* to traverse the program in reverse control flow direction to verify the query. It uses the *PropertyChecker* to get the *GEN* and *KILL* information. The *QueryChecker* returns *true* to *QueryGenerator* if it can determine that the property is available; otherwise it returns $false$.

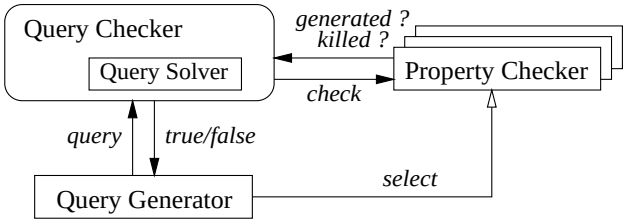



Figure 4.5: The components of array property analysis

The *QueryGenerator* and the *PropertyChecker* are closely related to the test problem and the properties being checked. We will show, in Sect.4.4, how to construct these two parts by using an example. In this section, we focus on the *QuerySolver*, which is a generic method.

To simplify the discussion, we assume no parameter passing, values are passed by global variables only, and if constant numbers are passed from one procedure to another, the callee is cloned. Techniques to handle parameter bounding, array reshaping, and variable aliasing are well known and can be found in [22, 39, 19, 58].

### 4.3.3 The Query Solver

The function of *QuerySolver* is as follows: given a query $(n_{query}, section_{query})$ and a root node $n_{root}$ that dominates $n_{query}$, *QuerySolver* returns a tuple $(anykilled, section_{remain})$. The *anykilled*, which is a boolean, is *true* if some element in $section_{query}$ might be killed when the program is executed from $n_{root}$ to $n_{query}$. When *anykilled* is $false$, $section_{remain}$ gives the array elements that are neither generated nor killed from $n_{root}$ to $n_{query}$.

Thus, in order to check if the index array elements in $section_{query}$ at node $n_{query}$ have the desired property, *QueryChecker* invokes *QuerySolver* with the $n_{root}$ being the entry node of the whole program. If *anykilled* is *true* or *anykilled* is $false$ but $section_{remain}$ is not empty, then we know the index array does not have the desired property. Otherwise, it has the desired property. In other words,

$$QueryChecker(n_{query}, sect_{query}) \doteq \overline{anykilled} \wedge (sect_{remain} = \emptyset),$$

$$\text{where } (anykilled, sect_{remain}) = QuerySolver((n_{query}, sect_{query}), entry_{prog})$$

```
Method:     QuerySolver(query, n_root)
Input:      1) a query query = (n_init, sect_init)
            2) a root node n_root that dominates n_init
Output:     (anykilled, sect_remain)
Begin:
            worklist := ∅ ;
            add_∪((n_init, sect_init), worklist) ;
            anykilled := false ;
            while worklist ≠ ∅ do
              take an element (n, sect) out of the worklist ;
              if ( n is n_root ) then
                sect_remain := sect ;
                break ;
              end if
              (anykilled, sect_remain) := QueryProp(n, sect) ;
              if ( anykilled ) then break ;
              if ( sect_remain ≠ ∅ ) then
                for each node m ∈ pred(n)
                  add_∪((m, sect_remain), worklist) ;
                end for
              end if
            end while
            return (anykilled, sect_remain) ;
End
```

Figure 4.6: QuerySolver

The algorithm for *QuerySolver* is shown in Figure 4.6. A worklist is used, and each element in the worklist is a query. The algorithm takes a query $(n, sect)$ out of the worklist. The query $(n, sect)$ asks whether any array element in $sect$ can have the desired property immediately after the execution of $n$. This is checked by reverse query propagation *QueryProp* (which is detailed in the next section). *QueryProp* returns a tuple $(anykilled, sect_{remain})$, whose meaning is similar to that of *QuerySolver*. The *anykilled* is set to *true* if any elements in $sect$ cannot have the property when $n$ is executed. In this case, a $false$ answer for the original query can be determined; thus, no further query is needed and the algorithm returns. This is an early-termination. When *anykilled* is false, new queries are constructed from the $sect_{remain}$ and the predecessors of $n$ and are inserted into the worklist. This process repeats until the worklist becomes empty or the root node is met. The use of a worklist makes early-termination possible.

The worklist is a priority queue. All the elements, which are queries of the form $(node, section)$,

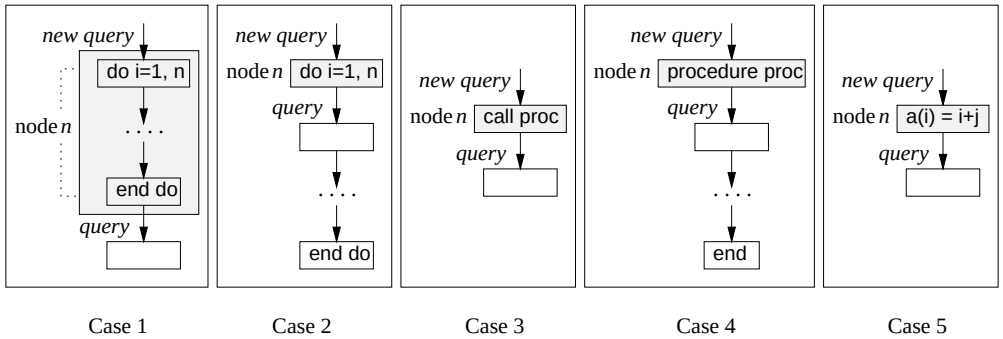



Figure 4.7: The five cases

are sorted in the reverse topological ($rTop$) order of $node$ in the control flow graph. Only the queries at the head position can be removed from queue. When a query is added to the worklist, it is inserted in the proper position according to the $rTop$ order. The first time, the query with the smallest $rTop$ order number is inserted into the empty worklist. Then, after a query $(node, section)$ is taken out, new queries for the predecessors of $node$ will be inserted into the worklist.

At any moment, the queries in the list are ordered such that the query at the head position always has the smallest $rTop$ number. Any query inserted into the list has a larger $rTop$ number than the query just removed from the worklist. Hence, it is easy to see that the algorithm visits the statements in the program according to their $rTop$ order. In other words, a node is not checked until all its successors have been checked. Thus, the query presented for a node can be composed from the queries propagated from its successors.

Queries are inserted into the list by using $add_{\cup}()$. The general $add_{op}()$, where $op$ can be either $\cap$ or $\cup$, is defined as

$$add_{op}((n, section), worklist)$$

$$\doteq \begin{cases} \text{if there exists a query } (n, section') \text{ in the} \\ worklist\text{, then replace } (n, section') \text{ with} \\ (n, section\ op\ section'); \\ \\ \text{otherwise,} \\ \text{insert } (n, section) \text{ into the } worklist \text{ according} \\ \text{to the } rTop \text{ order.} \end{cases}$$

| | |
|---|---|
| **Method**: | $QueryProp(n, section)$ |
| **Input**: | A query $(n, section)$ |
| **Output**: | $(anykilled, section_{remain})$ |
| **Begin**: | |
| 1 | $(Kill, Gen) := Summarize(n)$ ; |
| 2 | $section_{remain} := section - Gen$ ; |
| 3 | $anykilled := ((Kill \cap section) \neq \emptyset)$ ; |
| 4 | return $(anykilled, section_{remain})$ ; |
| **End** | |

Figure 4.8: A general framework of reverse query propagation $QueryProp$

### 4.3.4 Reverse Query Propagation

Conceptually, reverse query propagation $QueryProp$ computes the $IN$ and $DES$ from $OUT$, $GEN$ and $KILL$. Figure 4.8 shows a general framework of $QueryProp$ .

The $QueryProp$ uses $Summarize$ to summarize the effect of executing a node. The effect of executing a statement can be represented by the $(Kill, Gen)$ tuple, where $Kill$ is the section of the array elements that cannot possess the desired property because of the execution of $n$, and $Gen$ is the section of array elements that possess the desired property because of the execution of $n$. Whether an array element is generated or killed is determined by the *PropertyChecker*. The output of $QueryProp()$ is a tuple $(anykilled, section_{remain})$ whose meaning has been described in Sect.4.3.3.

The $Kill$ and $Gen$ evaluated by the summarization method are often approximate values. There are several reasons for this. First, the index array may be assigned variables whose values or relationships with other variables cannot be determined by the compiler; therefore, the section of the array elements being accessed cannot be represented precisely. Second, the summarization method works on array sections, but the set operations being used usually are not closed on section representations. Hence, the results can be only approximated.

In order not to cause incorrect transformations, the approximation must be conservative. $Kill$ is a $MAY$ approximation and $Gen$ is a $MUST$ approximation. In the worst case, $Kill$ can be the universal section $[-\infty, \infty]$ and $Gen$ can be $\emptyset$.

In real programs, the effect of executing a node may not be able to be derived explicitly, and the $Summary$ method is not available for all kinds of nodes. Hence, it is not always possible to

use the general framework in Figure 4.8. Depending on the semantics of the node $n$, there are five different cases, as illustrated in Figure 4.7:

- Case 1: $n$ is a do loop node,
- Case 2: $n$ is a `do` statement,
- Case 3: $n$ is a `call` statement,
- Case 4: $n$ is a procedure head, or
- Case 5: otherwise.

Accordingly, each case is handled by different reverse query propagation methods.

$$QueryProp \doteq \begin{cases} QueryProp_{simple} & \text{case 5} \\ QueryProp_{do\_loop} & \text{case 1} \\ QueryProp_{do\_header} & \text{case 2} \\ QueryProp_{proc\_call} & \text{case 3} \\ QueryProp_{proc\_head} & \text{case 4} \end{cases}$$

### 4.3.5 Simple Reverse Query Propagation

In case 5, the node $n$ is a statement other than a `do` statement, an `end do` statement, a `call` statement, or a procedure head. In this case, the effect of executing node $n$ can be derived by examining $n$ alone. $QueryProp_{simple}$ uses the same framework as $QueryProp$ in Figure 4.8 with $Summarize$ being replaced by $SummarizeSimpleNode$. The $SummarizeSimpleNode$ is also the interface between the $QuerySolver$ and the $PropertyChecker$ (see Sect.4.4.2).

**Example 4.1** *In Figure 4.9, statements $st1$ and $st2$ are simple assignments of array $a()$. The property to be checked is $a(i) = i * (i-1)/2$. Hence,*

$$SummarizeSimpleNode(st1) = (Kill = \emptyset, Gen = [n : n])$$
$$SummarizeSimpleNode(st2) = (Kill = [1 : 1], Gen = \emptyset)$$

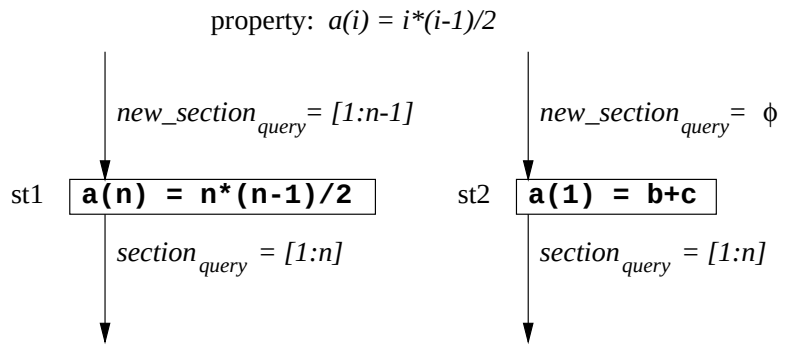



Figure 4.9: An example of simple reverse query propagation

*Thus, after the propagation, for statement st1 we have:*

$$anykilled = false, section_{remain} = [1 : n - 1].$$

*And, for statement st2 we have:*

$$anykilled = true, section_{remain} = \emptyset.$$

### 4.3.6 Loop Analysis

Cases 1 and 2 deal with loops. Array dataflow analysis is different from scalar analysis because different array elements might be accessed in different iterations of a loop, while the same set of scalars are usually accessed in all iterations. To summarize the effect of the loops, aggregation methods such as the one proposed by Gross and Steenkiste[33] (for one dimensional arrays) or by Gu et al.[34] (for multiple dimensional arrays) are used to aggregate the array access.

Given a section expression, $section_i$, which contains the loop index $i$, let $low$ be the lower bound of the loop and $up$ be the upper bound of the loop, $Aggregate_{low \leq i \leq up}(section_i)$ computes the section spanned by the loop index across the iteration space.

#### Reverse Query Propagation for Do Loops

In case 1, the initial query comes from outside the loop. Like the simple node case, the framework in Figure 4.8 can be used. The only difference is that we summarize the effect of executing the whole loop rather than a single statement.

| | |
|---|---|
| **Method**: | $SummarizeLoop(m)$ |
| **Input**: | A do loop node $m$ |
| **Output:** | $(Kill, Gen)$ |
| **Begin**: | |
| 1 | Let $n$ be the node corresponding to the loop body, and assume $i$ is the loop index ; |
| 2 | $(Kill_i, Gen_i) := SummarizeSection(n)$ ; |
| 3 | Let $up$ be the upper bound of loop $m$, and $low$ be the lower bound of loop $m$ ; |
| 4 | $Kill := Aggregate_{low \leq i \leq up}(Kill_i)$ ; |
| 5 | $temp := Aggregate_{i+1 \leq j \leq up}(Kill_j)$ ; |
| 6 | $Gen := Aggregate_{low \leq i \leq up}(Gen_i - temp)$ ; |
| 7 | return $(Kill, Gen)$ ; |
| **End** | |

Figure 4.10: *SummarizeLoop*

The method *SummarizeLoop* that summarizes a loop is given in Figure 4.10.

Basically, *SummarizeLoop* first computes the effect of executing the loop body, which corresponds to one iteration of the loop. Then, the effects are aggregated for the whole loop.

A collection of summarization methods are used here. The loop body is summarized by *SummarizeSection*. And, *SummarizeSection* uses three other methods (i.e., *SummarizeSimpleNode*, *SummarizeProcedure*, and *SummarizeLoop*) recursively depending on the type of statements used in the loop body.

*SummarizeProcedure* summarizes the effect of calling a procedure. Without considering the parameter bounding, *SummarizeProcedure* can be defined as

$$SummarizeProcedure(n) \doteq SummarizeSection(m),$$

where $m$ is the section node representing the procedure body of the procedure called by the call node $n$.

*SummarizeSection* is shown in Figure 4.12. It computes the effect of executing a section by reverse propagation of the *Kill* and *Gen* set from the exit node to the entry node. It also uses a worklist similar to the one used in *QuerySolver*. The elements $(n, gen)$ in the worklist, however, are not queries here. The *gen* is the section of array elements that have been generated because of the execution of the program from the exit of node $n$ to the exit of the section. Another difference

**Method**: $QueryProp_{do\_header}(m, sect)$
**Input**: A query $(m, sect)$, where $m$ is a node of `do` statement node
**Output**: $(anykilled, sect_{remain})$
**Begin**:
1 Let $n$ be the section node of the loop body, and assume $n$ represents the $i$th iteration of the loop ;
2 $(Kill_i, Gen_i) := SummarizeSection(n)$ ;
3 Let $up$ be the upper bound of loop, and $low$ be the lower bound of loop;
4 if ( $sect \cap Aggregate_{low \leq j \leq (i-1)}(Kill_j) \neq \emptyset$) then
5 return $(true, \emptyset)$ ;
6 end if
7 $sect_{remain\_i} := sect - Aggregate_{low \leq j \leq (i-1)}(Gen_j)$ ;
8 $sect_{remain} := Aggregate_{low \leq i \leq up}(sect_{remain\_i})$ ;
9 return $(false, sect_{remain})$ ;
**End**

Figure 4.11: $QueryProp_{do\_header}$

is that elements are inserted into the worklist by using $add_{\cap}$ instead of $add_{\cup}$. The effect of a section also could be computed in the forward direction. We use a backward method here because it is more efficient. It can early-terminate once the kill information is over-approximated to be the universal section (lines 21-24).

### Reverse Query Propagation for Do Loop Headers

In case 2, the initial query comes from one iteration of the loop. The method is a bit more complicated than the framework in Figure 4.8. Conceptually we do not summarize the whole loop, but rather the previous iterations of the loop. This gives us the new query section corresponding to one iteration, which should then be aggregated in order to get the query section for the predecessors of the loop. The method $QueryProp_{loop\_header}$ is shown in Figure 4.11.

```
Method:        SummarizeSection(n)
Input:         A section node n
Output:        (Kill, Gen)
Begin:
               Let n_entry be the entry node of section n, and
               let n_exit be the exit node of section n.
               Gen := ∅ ;
               Kill := ∅ ;
               WorkList := ∅ ;
               add_∩((n_exit, ∅), Worklist) ;
               while WorkList ≠ ∅ do
                 take an element (n, gen') out of the Worklist ;
                 if (n = n_entry) then
                   Gen := gen' ;
                   break ;
                 end if
                 begin case
                   case n is a call statement:
                     (kill, gen) := SummarizeProcedure(n) ;
                   case n is a do node:
                     (kill, gen) := SummarizeLoop(n) ;
                   otherwise:
                     (kill, gen) := SummarizeSimpleNode(n) ;
                 end case
                 if ( n dominates n_exit ) Gen := gen' ;
                 if ( kill = [−∞, ∞] ) then
                   Kill := kill ;
                   break ;
                 end if
                 Kill := Kill ∪ (kill − gen') ;
                 for each m ∈ pred(n)
                   add_∩((m, gen' ∪ gen), WorkList) ;
                 end for
               end while
               return (Kill, Gen) ;
End
```

Figure 4.12: *SummarizeSection*

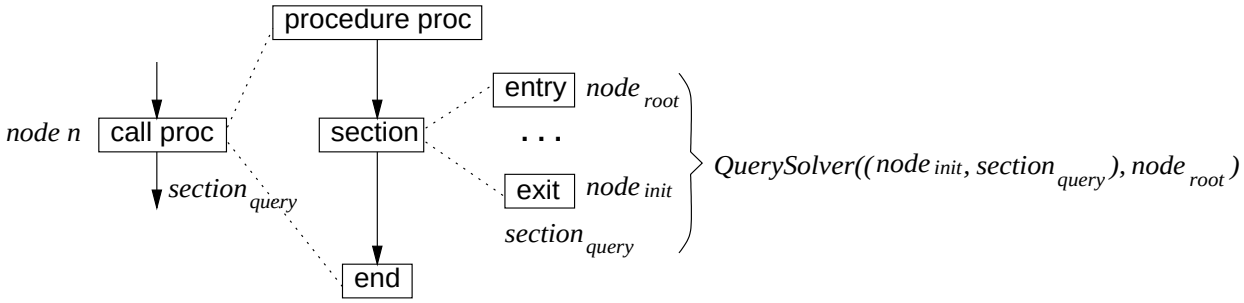



Figure 4.13: Reusing $QuerySolver$ for $QueryProp_{proc\_call}$

### 4.3.7 Interprocedural Analysis

The interprocedural reverse query propagation also involves two cases (i.e. cases 3 and 4 in Figure 4.7).

#### Reverse Query Propagation for Procedure Calls

In case 3, the node $n$ is a `call` statement. There are several ways to handle this case. A straightforward method is to use the framework in Figure 4.8 directly, with $Summarize$ being replaced with $SummarizeProcedure$. With the help of *memoization*(see Sect.4.3.8), the procedure needs to be summarized only once. The result can be reused again in later query propagations for this procedure, if there are any.

Another method is more demand-driven. We can construct a new query problem with the initial query node being the exit node of the callee and the root node being the entry node of the caller, as illustrated in Figure 4.13. Then, we can reuse $QuerySolver$ to propagate the query in a demand-driven way. $QuerySolver$ can early-terminate once any array element in query section is found not to have the desired property or all of them are found to have the property.

$$
\begin{aligned}
& QueryProp_{proc\_call}(n, section) \\
& \quad \doteq \; QuerySolver((m_{exit}, section), m_{entry}), \\
& \qquad \text{where } m_{exit} \text{ and } m_{entry} \text{ are the exit node and the} \\
& \qquad \text{entry node in the body section of the callee,} \\
& \qquad \text{respectively}
\end{aligned}
$$

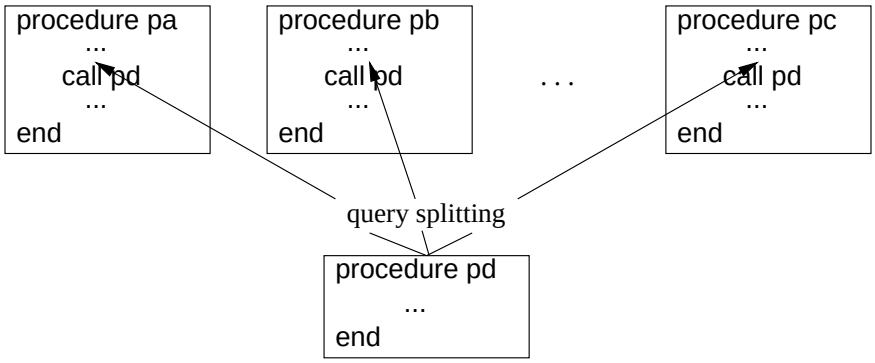



Figure 4.14: Query splitting

A third method is to combine the previous two. When propagating a query for a procedure call, we first check whether the summarized effect is already computed for the procedure. If it is available, we use the first method. Otherwise, we switch to the second one.

So far, we found that the performance of the three (in terms of execution time and precision) varies with the programs.

### Reverse Query Propagation for Procedure Entries

In case 4, the node $n$ is the entry node of a procedure. If $n$ is not the program entry, then the query will be propagated into the callers of this procedure. The framework in Figure 4.8 cannot be used. Instead, we use a query splitting method shown in Figure 4.15.

Suppose the property query at the entry node $n$ of a procedure $proc$ is $(n, sect_{query})$, and the call sites of $proc$ are $n_1, n_2, n_3, ...,$ and $n_m$. If $n$ is the program entry, then the array elements in $sect_{query}$ are not generated in this program and, as a result, whether they have the desired property cannot be verified. In this case, $QueryProp_{proc_{head}}$ returns with the $anykilled$ being true, and the property analysis can terminate with the answer being $false$. Otherwise, the query is split into $m$ sub-queries, each of which has a set of initial queries as $\{(n', sect_{query}) | n' \in pred(n_i)\}$, as illustrated in Figure 4.14. The original query has a $true$ result when all the sub-queries terminate with a $true$ result. Otherwise, the initial query has a $false$ result.

```
Method:        QueryProp_proc_head(n, sect)
Input:         a query (n, sect)
Output:        (anykilled, sect_remain)
Begin:
               if ( n is the program entry ) then
                 return (false, sect) ;
               end if
               Let p be the current procedure ;
               for each call site n of p
                 for each m ∈ pred(n)
                   if ( QueryChecker(m, sect) = false ) then
                     return (true, sect) ;
                   end if
                 end for
               end for
               return (false, ∅) ;
End
```

Figure 4.15: $QueryProp_{proc\_head}$

### 4.3.8 Memoization

Processing a sequence of $k$ queries requires $k$ separate invocations of *QueryChecker*, which may result in the repeated evaluations of the same intermediate queries. Similarly because a procedure may be called multiple times in a program, the summarization methods also may repeat several times for the same node. These repeated computations can be avoided by using *memoization*[1].

When memoization is applied, a procedure records, in a table, values that have previously been computed. When a memoized procedure is asked to compute a value, it first checks the table to see if the value is already there. If already there, it just returns the value; otherwise, it computes the new value in the ordinary way and stores the value in the table. In our analysis, the *QueryChecker*, the *QuerySolver*, the *QueryProp* and the *Summarize* methods all can be memoized. We choose to memoize the *QueryChecker* and the *Summarize* because they are most likely to be invoked with the same input repeatedly.

In a loop with indirectly accessed arrays, we found that different host arrays often shared the same index array and access pattern. For example, when detecting the dependences in the loop in Figure 4.16, the compiler should check the dependences between `a()` at `s1` and `a()` at `s2`, between two different instances of `a()` at `s1`, and between two difference instances of `b()` at `s3`. Suppose

```
s0: ...
    do i = 1, n
       do j = 1, length(i)
s1:       a(offset(i)+j-1) = ...
s2:                   ...  = a(offset(i)+j-1) + ...
s3:       b(offset(i)+j-1) = ...
       end do
    end do
```

Figure 4.16: Array $a()$ and $b()$ share the same access pattern

the query demands are issued at statement `s0`. Because the same index array and the same access pattern is used, the same query will be evaluated three times. If memoization is used, only one evaluation is needed.

Summarization methods are called independently of the query sections. Memoization tends to be more effective for summarizations than *QuerySolver* and *QueryProp*, which depend not only on the index array and the property to be checked, but also on query sections, which often vary from instance to instance.

### 4.3.9 Cost Analysis

Let $|N|$ be the number of HCG nodes and $|E|$ be the number of edges in a program. We assume $|E| = O(|N|)$ because we are working on structured programs. For a single query, memoizing the summarization methods requires a storage space of size $O(|N|)$. To determine the time complexity, we consider the number of array section operations (intersection, union, subtraction and aggregation) and the number of $PropertyCheck$ methods. The latter is $O(|N|)$ as $PropertyCheck$ is invoked only in $SummarizeSimpleNode$. $PropertyCheck$ is executed once at most for each node because of the memoization. The array section operations are performed for each edge in the query propagation methods and the summarization methods. The number is $O(n_{inlined})$ for query propagation methods and $O(|N|)$ for summarization methods, where $n_{inlined}$ is the number of statement nodes if the program is fully inlined. Hence, the complexity of execution time is $O((C_{pc} + C_{as}) * |N| + C_{as} * n_{inlined})$, where $C_{pc}$ is the cost of a $PropertyCheck$ and $C_{as}$ is the cost of an array section operation. Because we make approximations when property check cannot be performed locally or array sections become complicated, both $C_{pc}$ and $C_{as}$ can be considered as

constants here.

## 4.4 Examples of Query Generator and Property Checker

Of the three major components of array property analysis, the *QueryChecker* is a generic method, while the *QueryGenerator* and the *PropertyChecker* are specific to the base problem (e.g., dependence tests and privatization tests) and specific to the potential property the array is likely to hold. In this section, we use a data dependence test problem to illustrate how to construct the *QueryGenerator* and the *PropertyChecker*.

### 4.4.1 Generating Queries

We first present a new dependence test called *offset-length* test. The *offset-length* test is designed to detect the data dependence in loops where indirectly accessed arrays are present. It is called the *offset-length* test because it can disprove dependences when the index arrays are used as offset arrays and length arrays, such as the `offset()` and the `length()` in Figure 4.1. The *offset-length* test needs array property analysis to verify the relationship between the *offset* arrays and the *length* arrays; hence, it also serves as a demand generator.

We consider two array accesses in statements (1) and (2), at least one of which is a write, in the loop nest in Figure 4.17.

Loops $do\_i_1$ through $do\_i_r$ are common to statements (1) and (2). Loops $do\_j_1$ through $do\_j_p$ are private to statement (1). And, loops $do\_k_1$ through $do\_k_q$ are private to statement (2). The loop bounds of loops other than loops $do\_i_1$ through $do\_i_t$ and the subscript functions $f()$ and $g()$ may contain index arrays.

We test whether there is a loop carried dependence between access (1) and access (2) for loop $do\_i_t$ (i.e., whether there is a dependence with the dependence direction vector $(=_1, =_2, ..., =_{t-1}, \neq_t, *, ..., *)$). We assume that the index arrays in the loop nest are arrays of the $t$ outermost loops only.

We first compute both the ranges of values of $f(i_1, ..., i_t, *, ..., *)$ and $g(i_1, ..., i_t, *, ..., *)$ when $i_1, i_2, ..., i_t$ are kept fixed. Because the index arrays are arrays only of the outermost $t$ loops, the loop indices $i_1, i_2, ..., i_t$ and the index arrays can be treated as symbolic terms in the range

```
do i_1 = 1, m
  ...
  do i_t = 1, n
    ...
    do i_r = ...
      do j_1 = ...
        ...
        do j_p = ...
          a(f(i_1, i_2, ..., i_t, ..., i_r, j_1, j_2, ..., j_p))  (1)
        end do
      end do
      do k_1 = ...
        ...
        do k_q = ...
          a(g(i_1, i_2, ..., i_t, ..., i_r, k_1, k_2, ..., k_q))  (2)
        end do
      end do
    end do
  end do
end do
```

Figure 4.17: A loop nest

computation. If, except for the index arrays, $f()$ or $g()$ is an affine function of the loop indices, the range can be calculated by substituting the loop indices with their appropriate loop bounds, as done in *Banerjee's test* [7]. Otherwise, the ranges are calculated with the method used in some nonlinear data dependence tests, such as the *Range test* [15].

Before we continue, let's present the following proposition.

**Proposition 4.1** *Let $R_1, R_2, ..., R_n$ and $R'_1, R'_2, ..., R'_n$ be two sequences of ranges, where $R_i = [x(i) + c_1,\ x(i) + y(i) - d_1]$ and $R'_i = [x(i) + c_2,\ x(i) + y(i) - d_2]$ for $1 \leq i \leq n$; $x()$ and $y()$ are two arrays; $c_1$ and $c_2$ are non-negative constants; and $d_1$ and $d_2$ are positive constants. If*

$$x(i+1) = x(i) + y(i),\ \textit{for}\ 1 \leq i \leq n-1$$

*and*

$$y(i) \geq 0,\ \textit{for}\ 1 \leq i \leq n$$

*then*

$$R_i \cap R_j = \emptyset, \textit{when}\ i \neq j, 1 \leq i, j \leq n$$

*and*

$$R_i \cap R'_j = \emptyset, where\ i \neq j, 1 \leq i, j \leq n$$

**Corollary 4.1** *For the loop in Figure 4.17, if*

1. *Condition 1*

   *the range of* $f(i_1, ..., i_t, *, ..., *)$ *can be presented as*

   $$[x(i_t) + f_{low}\ , x(i_t) + y(i_t) + f_{up}],$$

   *and the range of* $g(i_1, ..., i_t, *, ..., *)$ *can be presented as*

   $$[x(i_t) + g_{low}\ , x(i_t) + y(i_t) + g_{up}],$$

   *where* $x()$ *and* $y()$ *are two index arrays, and*

   $$\begin{aligned} f_{low} &= e(i_1, ..., i_{t-1}) + c_1 \\ f_{up} &= e(i_1, ..., i_{t-1}) - d_1 \\ g_{low} &= e(i_1, ..., i_{t-1}) + c_2 \\ g_{up} &= e(i_1, ..., i_{t-1}) - d_2 \end{aligned}$$

   $e(i_1, ..., i_{t-1})$ *is an expression of indices* $i_1$, $i_2$,...,$i_{t-1}$ *and index arrays of the outermost* $t-1$ *loops;* $c_1$ *and* $c_2$ *are some non-negative integers; and* $d_1$ *and* $d_2$ *are some positive integers, and*

2. *Condition 2*

   *we know that index array* $x()$ *has a closed-form distance* $y()$*, and*

3. *Condition 3*

   *the value of array elements of* $y()$ *are not negative,*

```
s0: do i=1, n
       do j=2, iblen(i)
          do k=1, j-1
             ...
s1:          x(pptr(i)+k-1) = ...
             ...
          end do
       end do
       ...
       do j=1, iblen(i)-1
          do k=1, j
             ...
s2:          ... = x(iblen(i)+pptr(i)+k-j-1)
             ...
          end do
       end do
    end do
```

Figure 4.18: A loop form DYFESM

*then there is no loop carried dependence between* (1) *and* (2)*, between two instances of* (1)*, or between two instances of* (2) *for loop* $do\_i_t$*.*

In Corollary 4.1, Condition 1 can be checked locally after the ranges are computed, while Conditions 2 and 3 need be to verified by array property analysis.

**Example 4.2** *Figure 4.18 shows a loop nest excerpted from the subroutine SOLXDD of Perfect Benchmark code DYFESM.*

*We want to check if there is any loop-carried dependence between statement* `st1` *and statement* `st2` *for the outermost loop* `do i`*.*

*Here,* $f(i,j,k) = pptr(i) + k - 1$ *and* $g(i,j,k) = iblen(i) + pptr(i) + k - j - 1$*. By substituting the loop indices with the loop bounds, we can compute the ranges of* $f()$ *and* $g()$ *when* $i$ *is fixed, which are* $[pptr(i), pptr(i) + iblen(i) - 2]$ *and* $[pptr(i) + 1, pptr(i) + iblen(i) - 1]$*, respectively. Hence, according to Proposition 4.1, if* $pptr()$ *has a closed-form distance of* $iblen()$*, which is non-negative, then for the outermost loop* `s0` *there is no flow-dependence from* `s1` *to* `s2`*, no anti-dependence from* `s2` *to* `s1`*, and no output-dependence from* `s1` *to* `s1`*.*

**Corollary 4.2** *For the loop in Figure 4.17, if*

1. *Condition 1*

   *the range of* $f(i_1, ..., i_t, *, ..., *)$ *can be presented as*

$$[x(i_t) + f_{low},\ x(i_t) + y(i_t) + f_{up}],$$

   *and the range of* $g(i_1, ..., i_t, *, ..., *)$ *can be presented as*

$$[\min_{1 \le i \le i_t}(x(i) + g_{low}),\ \max_{1 \le i \le i_t}(x(i) + y(i) + g_{up})],$$

   *where* $x()$ *and* $y()$ *are two index arrays, and*

$$\begin{aligned} f_{low} &= e(i_1, ..., i_{t-1}) + c_1 \\ f_{up} &= e(i_1, ..., i_{t-1}) - d_1 \\ g_{low} &= e(i_1, ..., i_{t-1}) + c_2 \\ g_{up} &= e(i_1, ..., i_{t-1}) - d_2 \end{aligned}$$

   $e(i_1, ..., i_{t-1})$ *is an expression of indices* $i_1$, $i_2$,...,$i_{t-1}$ *and index arrays of the outermost* $t-1$ *loops, and* $c_1$, $c_2$, $d_1$ *and* $d_2$ *are some non-negative integers. And,*

2. *Condition 2*

   *same as Condition 2 in Corollary 4.1, and*

3. *Condition 3*

   *same as Condition 3 in Corollary 4.1,*

*then there is no loop carried dependence between two instances of* (1) *for loop* $do\_i_t$, *and there is no loop carried dependence from* (2) *to* (1) *for loop* $do\_i_t$.

**Example 4.3** *Suppose* $ptr()$ *has a closed form distance of* $len()$ *which is non-negative in the loop in Figure 4.19. Then, because of the first dimension of* `x()`, *there is no loop carried dependence from* `s3` *to* `s3`, *nor from* `s4` *to* `s3` *for loop* `s0`. *And, because of the second dimension of* `x()`, *there*

```
s0:  do i=1, n
        do j=1, len(i)
s1:        ij = ptr(i)+j-1
           do k=1, i
              do l=1, len(k)
s2:              kl = ptr(k)+l-1
s3:              x(ij,kl) = ...
s4:              x(kl,ij) = ...
              end do
           end do
        end do
     end do
```

Figure 4.19: Another loop

*is no loop carried dependence from* `s4` *to* `s4`*, nor from* `s3` *to* `s4` *for loop* `s0`*. Hence, the outermost loop is parallel.*

The *offset-length* test can be a stand-alone test or can be integrated with other tests, such as *Banerjee's* test and the *Range* test.

### 4.4.2 Checking Properties

Given a property to be verified and an assignment statement, the property checker $PropertyChecker$ checks whether the assignment will cause any array elements to be generated or killed. In Section 3.2.2, we have already shown an example about how to use irregular single-indexed array access analysis to detect closed-form bounds and injectivity in index gathering loops. In this subsection, we show how to use a simple pattern matching technique to check the *closed-form distance*.

Suppose the given property to be verified is

$$x(i+1) = x(i) + y(i), \text{ for } 1 \leq i \leq n-1.$$

The $PropertyChecker$ can, as an example, take the following steps to inspect an assignment statement:

1. if the left-hand side (LHS) of the assignment is neither the array $x()$ nor the array $y()$, then nothing is generated or killed ;

```
x(1) = ...
do i = 2, n
   x(i) = x(i-1) + y(i-1)
end do
```

(a)

```
t = ...
do i = 1, n
   x(i) = t
   t = t + y(i)
end do
```

(b)

Figure 4.20: Two program patterns for *closed-form distance*

2. if the LHS is an array element $x(i)$, then the assignment and the other statements in the surrounding loops are checked to see if they match any of the following two patterns shown in Figure 4.20. If not, then all elements of $x()$ are killed. Otherwise, $x(i)$ is generated ;

3. in all other cases (this includes the case when the LHS is an array element of $y()$ and the case when the LHS is an array element of $x()$ but the subscript is not a simple index), all elements of $x()$ are killed.

In general, the *closed-from distance* can be detected by using abstract interpretation, such as the recurrence recognition method proposed by Z. Ammarguellat and W. Harrison[4]. Compared with abstract interpretation, our pattern matching method is simplified and, thus, conservative. However, we found this simplification to be very effective in practice. For most cases, $PropertyChecker$ never needs more sophisticated methods to get precise results.

## 4.5 Using Run-time Test to Check Properties

So far, the methods we have described are purely compile-time techniques. There are, however, cases where the index arrays are read from input files and their properties cannot be verified until run-time.

For example, in the property analysis of array $pptr()$ and $iblen()$ in Example 4.2, it can be verified at compile-time, by analyzing the code DYFESM, that array $pptr()$ has a closed-form distance of $iblen()$. But array $iben()$ is defined in a `read` statement. A run-time test is needed to check if the values of elements in $iblen()$ are non-negative.

In this section, we describe a scheme to extend array property analysis to incorporate run-time tests.

### 4.5.1 At Compile-time

A global run-time test repository is used to keep a record of the query demands and the run-time tests generated at compile-time. Each query demand has an entry in the repository. Each entry has a unique demand id as the lookup key. An entry has a *need_runtime_test* tag, which can have two values. A *true* means a run-time test is generated and required. A *false* means no run-time test is generated or the compile-time analysis has found the query is false and no run-time test is needed.

When a query demand is generated, the compiler creates an entry in the repository and sets the *need_runtime_test* tag to *false*.

During array property analysis, if an I/O input statement is met and a run-time test is required, the compiler generates a run-time test node, inserts the node to the program, inserts a reference of the node to the repository, and sets the *need_runtime_test* tag for the original query demand to *true*. Then, array property analysis continues as if the query is verified to be *true* at the input statement. If, during the analysis, the query is found to be *false*, then the compiler sets the *need_runtime_test* tag to *false*.

After array property analysis, if the property is found to be *true* and the *need_runtime_test* tag also is *true*, then the compiler generates two versions of the loop (one optimized and one original) which are guarded by a test of the run-time test results. If the property is found to be *false*, then the repository is looked up to locate all the generated run-time test nodes, which are then removed from the program.

### 4.5.2 At Run-time

Like at compile-time, a global run-time test repository also is used at run-time. Each query demand that requires a run-time test has an entry in the repository. Each entry has the same unique demand id as in the compile-time repository. A *result* tag is kept in each entry to save the run-time results. The tags are set to *true* when the program starts.

When a run-time test is performed, the run-time test repository is looked up to find the entry with the demand id of the query that requires this run-time test. If the *result* is *false*, which means the property has already been verified as *false* by some other run-time test, then the run-time test

is skipped. Otherwise, the run-time test is executed and the result is set to the *result* tag in the repository.

When choosing which version to be executed at the site where the query was generated at compile-time, the repository is looked up again. If the value of the *result* tag is *true*, then the optimized version is executed; otherwise, the unoptimized version is executed.

## 4.6 Related Work

Two different studies, done by Z. Shen, Z. Y. Li, and P.C. Yew [68] and P. Petersen and D. Padua [59], have found that index arrays are a significant source of imprecision in dependence testing. B. Blume and R. Eigenmann studied the Perfect Benchmarks and found detecting index array properties to be important[13]. The same conclusion is also achieved by the authors in a study of several sparse/irregular Fortran programs[49].

Our approach of modeling a demand for property checking as a set of queries was inspired by the work of E. Duesterwald, R. Gupta, and M. Soffa [26]. They proposed a general framework for developing demand-driven interprocedural data flow analyzers. They also show that demand-driven analysis reduces both time and space requirements when compared with exhaustive analysis in practice. However, they use an iterative method to propagate the queries and can handle only scalars. We use a more efficient structural analysis and work on arrays. We can do this because we have a more specific problem.

Array data flow analysis has been studied by many researchers, such as [33, 29, 34, 64]. All of them use exhaustive analysis. The idea of representing array elements by an array section descriptor and using aggregations to summarize the elements accessed by a loop was first proposed by Callahan and Kennedy[20] and was used in our method to handle arrays in loops. Another class of array data flow analysis methods uses the framework of data dependence and can provide fine-grain instance-wise information[29, 64]. However, this approach is difficult to use in whole program analysis.

Compile-time analysis of index arrays was also studied by K. McKinley[55] and K. Kennedy, K. McKinley, and C. Tseng [44]. They investigated how user assertions about the index arrays could be used in dependence tests. The user assertions correspond to the properties in our work. They

focus on how to use the properties, while we focus on how to get the properties automatically. Their work complements ours when the properties are undetectable at compile-time.

We found that, in real programs, subscripts of host arrays are usually simple when index arrays were used. Most of them are of the form $p(i) \pm j$, where $p(i)$ is an index array and $i$ and $j$ are loops indices. The extended SIV tests, such as those described in [55], and our *offset-length* test were usually sufficient to generate the property queries. For more general problems, W. Pugh and D. Wonnacott had proposed a nonlinear array dependence analysis method [63]. In their work, nonlinear expressions (including index arrays) are treated as uninterpreted function symbols, and a dependence test is represented as a Presburger formula. By simplifying the formula, they can determine the conditions under which a dependence exits. Not surprisingly, the conditions they found in their experiments with Perfect Benchmarks corresponded to the properties we described. Hence, their method can be used in the demand generator for general cases in dependence tests.

## 4.7 Summary

Many optimization techniques reply on the analysis of array subscripts. Current compilers often give up optimizations when arrays are subscripted by index arrays and treat the index arrays as unknown functions at compile-time. However, recent empirical studies of real programs have shown that index arrays often possess some properties that can be used to derive more precise information about the enclosing loops. In this chapter, we presented an index array analysis method, called array property analysis, which verifies the property of an array by back propagating a property query along the control flow of the program. This method integrates demand-driven approach, interprocedural analysis, and array dataflow analysis. We also illustrated how to generate the property query in a data dependence test and how to construct a query checker based on simple pattern matching.

We have implemented array property analysis in our Polaris parallelizing compiler and measured its impact in finding more parallelism. We will show the experimental results in the next chapter.

# Chapter 5

# IMPLEMENTATIONS AND EXPERIMENTS

Compile analysis of irregular memory accesses can enable deeper analysis in many parts of an optimizing compiler. To evaluate its effectiveness, we measured its impact in finding more implicit parallelism. In this chapter, we describe the implementation of the irregular single-indexed array analysis and the implementation of the demand-driven interprocedural array property analysis in our Polaris parallelizing compiler. We also show the experimental results using five programs in our benchmark suite described in Chapter 2.

## 5.1 Implementation

### 5.1.1 Reorganize the Phases in Polaris

Like most compilers, Polaris is organized in phases. The high level structure of Polaris is shown in Figure 5.1.(a). Except for the inlining and interprocedural constant propagation, all other phases are intraprocedural. For each program unit, Polaris performs a sequence of analyses and transformations in order. This structure is good for data locality and, therefore, good for the efficiency of Polaris. It also makes debugging Polaris a bit faster and easier.

The structure is not, however, good for array property analysis. As we discussed in Chapter 4, array property analysis must be performed interprocedurally in order to be effective for real programs. Since an interprocedural analysis may need to work on any program unit, we want all program units to have applied the same set of transformations before the analysis starts. We

```
scanner

inlining

interprocedural constant propagation

for each program unit do
    program normalization
    induction variable substitution
    constant propagation
    forward substitution
    dead code elimination
    privatization
    reduction recognition
    data dependence test
end do

postpass
```

(a) before

```
scanner

inlining

interprocedural constant propagation

for each program unit do
    program normalization
    induction variable substitution
    constant propagation
    forward substitution
    dead code elimination
end do

for each program unit do
    privatization
end do

for each program unit do
    reduction recognition
end do

for each program unit do
    data dependence test
end do


postpass
```

(b) after

Figure 5.1: Reorganize the phases in Polaris

reorganize the phases in Polaris, after which the high level structure of Polaris looks like Figure 5.1.(b). The "loop distribution" way of reorganization is made possible due to the good modularity of the phases implemented in Polaris.

We did not remove the inlining phase because most analyses in Polaris were not interprocedural and they relied on inlining to produce precise results. We used the default auto inlining function in Polaris, which inlines procedures that contain no I/O statements and whose call-sites are in a loop and whose number of lines are less than fifty. As not all procedures are inlined, the interprocedural part of our array property analysis is still required and proved to be useful.

### 5.1.2 Array Property Analysis as a Demand-driven Tool

Array property analysis is not a stand alone phase. It is implemented as a independent tool that can be invoked by demand.

A `caller` graph and a `callee` graph that represent the procedural call relationship in the program being parallelized are constructed before the phase that will use array property analysis. We assume that, during the phase, the program is not transformed so that the caller graph and callee graph remain valid during the analysis.

The array property analyses that check different properties are implemented as different subclasses of a common $PropertySolver$ class which realizes the property independent $QuerySolver$ discussed in Section 4.3.3. The subclasses implement the property dependent parts, such as the $PropertyChecker$. When an array property is to be checked, an object of one of the subclasses is created, array sections in the query are passed to the object, and the analysis is invoked by calling the $DdriveSolve()$ method.

In Polaris, the array property analysis is used in the privatization phase and in the data dependence test phase.

### 5.1.3 Array Privatization

The original privatization phase was designed by Peng Tu [74]. The method, based on Corollary 2.1 described in Section 2.4, was used to test the validity of privatization. That is, an array is privatizable if the upward exposed read set of the array in each iteration is empty.

To compute the upward exposed read set, the sets of array elements that are read or written by each statement are calculated. To make it easier for set operations, a set of array elements is represented as an array section. To approximate in the safe direction, a read section can be a superset of its corresponding real read set, and a write section can be a subset of its corresponding real write set.

In the original design, the array subscripts must be linear expressions and the surrounding loops must be `do` loops. The array accesses cannot be irregular; otherwise, the read set has be to approximated to $[-\infty, \infty]$, and the write set has to be approximated to $\emptyset$.

We extended the computation method for the read/write sections so that it can handle the

consecutively-accessed arrays and simple indirectly-accessed arrays. The methods described in Section 3.2.2 are used to get the ranges of the index variables in consecutively-accessed arrays. For the simple indirectly-accessed arrays, array property analysis is used to verify the bounds of the index arrays. A set of indirectly-read array elements now can be approximately represented in array sections. For example, $\{a(p(i))|1 \leq i \leq n\}$ is approximated to $a[low : high]$, where $low = min(p(i))$ and $high = max(p(i))$ for $(1 \leq i \leq n)$. Although this approximation works for read sets only, it has proven to be useful in our experiments.

### 5.1.4 Data Dependence Test

The original data dependence test phase was designed by Bill Blume [17]. A variety of data dependence tests were implemented as "filters" which filter out the non-existed dependence between two statements. Given a data dependence graph $(V, E)$ where $V$ is the set of statements in a loop and $E$ is the set of arcs between the statements, each arc represents a data dependence. The data dependence graph is a multigraph because there may exist more than one data dependence between two statements. For each arc, the data dependence tests are performed in the order of increasing complexity.

An important class of tests used is the *single subscript test*. If the arrays under test are multi-dimensional, these tests check the subscripts one dimension by one dimension. The tests in this class that are used by Polaris include: the simple subscript test, the GCD test [8], and the range test. The simple subscript test removes dependences of the form "$a(i) \rightarrow a(i)$". The range test is a symbolic data dependence test that can identify parallel loops in the presence of certain nonlinear array subscripts and loop bounds [15].

We extended the range test so that it could function like the *offset-length* test discussed in Section 4.4.1 when the index arrays were used as offsets and length. We found the range test a natural place to incorporate the offset-length test because it also computed the symbolic range of subscript expressions which were used in the offset-length test. We also implemented a stand alone *simple offset-length* test that tested the subscripts of the form "a(ptr(i)+j)". It could be used when the user wanted to avoid the overhead of the extended range test, though it was less general. An *injective* test was also added for the case when the subscript was a simple index array like "a(p(i))".

```
SingleSubscriptFilter
  SimpleSubscriptTestFilter
  GCDTestFilter
  SimpleoffsetLengthTestFilter (newly added)
  InjectiveTestFilter (newly added)
  BaseRangeTestFilter (extended)
    RangeTestFilter (extended)
    DVRangeTestFilter (extended)
```

Figure 5.2: Hierarchy of extended and newly added data dependence test

The hierarchy of the extended single subscript filter class is shown in Figure 5.2. All the extended and newly added tests need property analysis of index arrays.

## 5.2 Experimental Results

### 5.2.1 Overview

Table 5.1 shows the five programs used in our experiments. TRFD, BDNA and DYFESM are from the PERFECT Benchmark suite. P3M is a particle-mesh program from NCSA. TREE is a Barnes-Hut N-body program from the University of Hawaii [10]. The compilation time of the programs using Polaris is listed in column four. Array property analysis increases the compilation time by 4.5% to 10.9%[1]. These data were measured on a Sun Enterprise 4250 Server with four 248MHz UltraSPARC-II processors. The sequential execution time (measured on an SGI Origin2000 with fifty six 195MHz R10k processors) of these programs is listed in column three.

Table 5.2 shows the results of the analysis. Column two shows the loops with irregular array accesses that can be analyzed by Polaris now. The loops with a "*" are the newly parallelized loops. The loops without a "*" are not parallel, but their analysis results are used to help parallelize the loops with a "*". The properties of the irregular array accesses are listed in columns five and eight. Column nine shows the tests that were used as the query generators in array property analysis. Column ten shows the percentage of total sequential program execution time (on the Origin2000) accountable to the loops in column two. And, column eleven shows the percentage of total parallel program execution time accountable to these loops if the loops are not parallelized (the number

[1]The data of P3M is for subroutine PP only.

| | Lines of Codes | Sequential Program Execution Time (SGI Origin2000) | Polaris Execution Time (Sun 4250) | | |
|---|---|---|---|---|---|
| | | | Whole Program | Array Property Analysis | % |
| TRFD | 380 | 4.4s | 181.3s | 8.1s | 4.5% |
| DYFESM | 7650 | 3.2s | 302.3s | 19.4s | 6.4% |
| BDNA | 4896 | 9.7s | 465.7s | 31.2s | 6.7% |
| P3M* | 2414 | 355.8s | 73.1s | 8.0s | 10.9% |
| TREE | 1553 | 8.3s | 25.7 | 17.1 | 6.7% |

Table 5.1: Compilation time using Polaris. The fourth column shows the whole program compilation time. The fifth column is the time spent in irregular array access analysis.

| Program | Loops | Single-indexed Access | | | Indirectly Array Access | | | | $\%_{seq}$ | $\%_{par}$ |
|---|---|---|---|---|---|---|---|---|---|---|
| | | Array | Index | Property | Host | Index | Property | Test | | |
| TRFD | INTGRL/do_140* | - | - | - | x | ia | CFV | DD | 5% | $24\%_{32}$ |
| DYFESM | SOLXDD/do_4*<br>SOLXDD/do_10*<br>SOLXDD/do_30*<br>SOLXDD/do_50*<br>HOP/do_20* | - | - | - | xdd, z<br>r, y<br>z<br>xdd<br>xdplus,<br>xplus, xd | pptr,<br>iblen | CFD | DD | 20% | $7\%_{8}$ |
| BDNA | ACTFOR/do_240* | - | - | - | xdt | ind | CFB | PRIV | 32% | $63\%_{32}$ |
| | ACTFOR/do_236 | ind | l | CW | - | - | - | - | - | - |
| P3M | PP/do_100* | - | - | - | x0, ind0<br>r2, ind | jpr | CFB | PRIV | 74% | $76\%_{8}$ |
| | PP/goto_10<br>PP/do_50<br>PP/do_57 | ind0, x0<br>ind0, x0<br>jpr | np0<br>np0<br>npr | CW<br>CW<br>CW | - | - | - | - | - | - |
| TREE | ACCEL/do_10* | stack | sptr | STACK | - | - | - | - | 90% | $90\%_{32}$ |

Table 5.2: Programs used in our experiment. CW - consecutively written, STACK - stack access, CFV - closed-form value, CFB - closed-form bound, CFD - closed-form distance, PRIV - privatization test, DD - data dependence test.

after the % sign is the number of processors used). One to thirty-two processors were used.

Figure 5.3 shows the speedups of these programs. We compare the speedups of the programs generated by our Polaris parallelizing compiler, with and without irregular array access analysis, and the programs compiled using the automatic parallelizer provided by SGI. The experiments were performed on an SGI Origin2000 machine with 56 195MHz R10000 processors (32KB instruction case, 32KB data cache, 4MB secondary unified level cache) and 14GB memory running IRIX64 6.5. One to thirty-two processors were used for BDNA and TREE. One to eight processors were used for P3M. "APO" means using the "-apo" option when compiling the programs. This option invokes the SGI automatic parallelizer. "Polaris without IAA" means using the Polaris compiler without irregular array access analysis. "Polaris with IAA" means using the Polaris compiler with irregular array access analysis. For all five codes, the speedups of the versions in which irregular array access analysis had been used are much better than those of the other versions. As we have not yet implemented array stack analysis in our Polaris compiler, for TREE we show the results of manual parallelization.

### 5.2.2 TRFD

TRFD is a kernel simulating the computational aspects of a two-electron integral transformation [61].

Loop INTGRL/do_140, which is inherently parallel, occupies about 5% of sequential execution time. But, if it is not parallelized, it will occupy about 24% of the parallel execution time when thirty-two processors are used.

A simplified version of loop INTGRL/do_140 is shown in Figure 5.4. It is almost identical to the loop in Example 4.3. The only difference is that the `len(i)` is replaced with `i`. When irregular array access analysis is enabled, the extended range test generates two queries: one checks whether array `ia()` has a closed form distance of `i`, and the other checks whether array `ia()` has a closed form value of `(i*(i-1))/2`, where `i` is the array index. If any query returns *true*, then the dependence is broken.

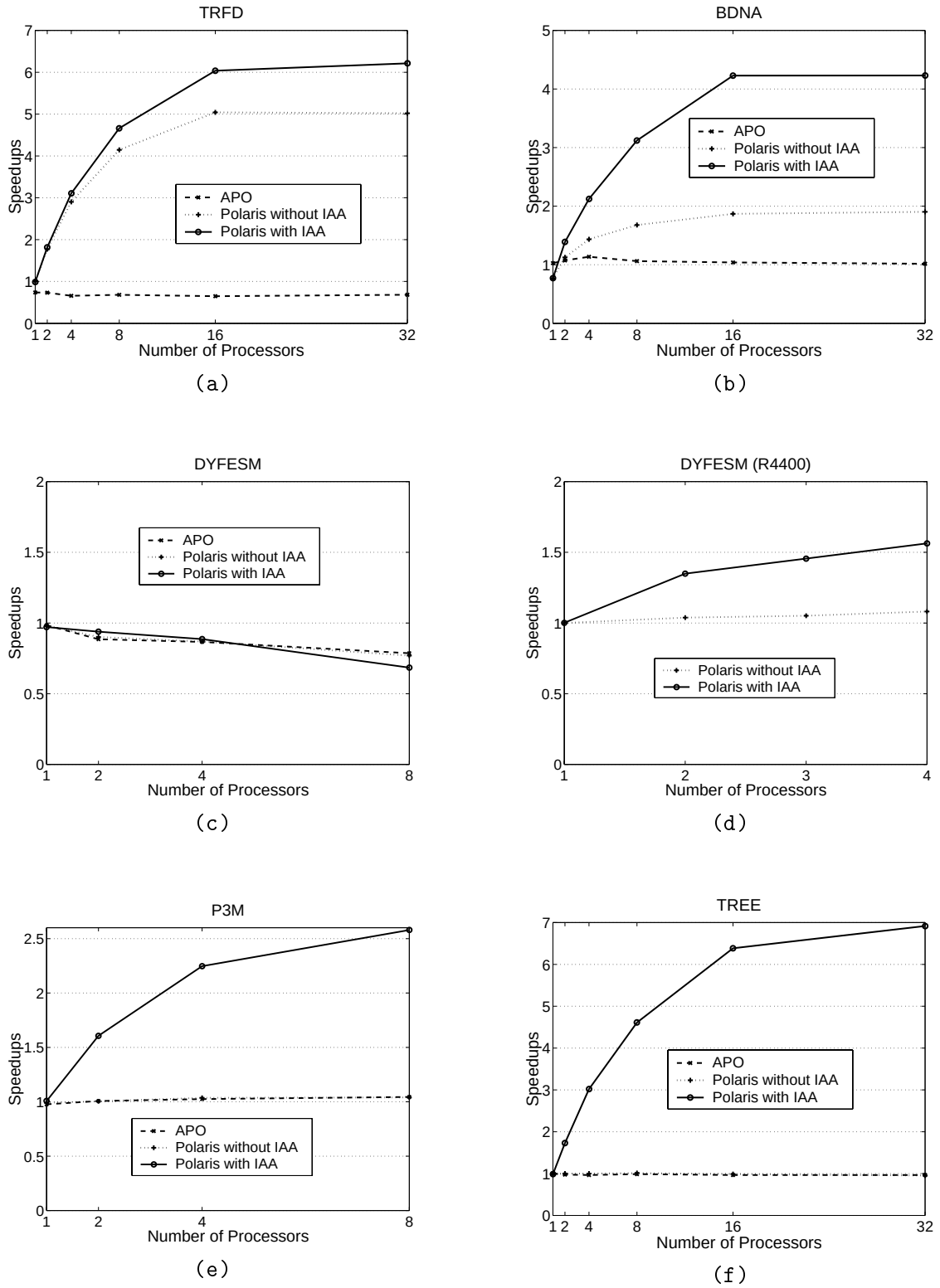



Figure 5.3: Speedups: IAA - irregular array access analysis, APO - using the automatic parallelization option in the SGI F77 compiler

```
do i=1, n
   ia(i) = (i*(i-1))/2
end do

call intgrl


subroutine intgrl

do i=1, n       <- do 140
   do j=1, i
      ij = ia(i) + j
      do k=1, i
         do l=1, k
            kl = ia(k)+l
            x(ij,kl) = ..
            x(kl,ij) = ..
         end do
      end do
   end do
end do
```

Figure 5.4: Loop INTGRL/do_140 in TRFD

### 5.2.3 BDNA

BDNA is a molecular dynamics simulation code from the PERFECT benchmark suite [27].

The `do 240` loop in subroutine $ACTFOR$ is a loop that computes the interaction of biomolecules in water. It occupies about 31% of total computation time. The main structure of this loop is outlined in Figure 5.5

Consecutively written array analysis is used in the `do j2` loop to find that elements in $[1, k]$ of $ind()$ are written in this loop. Furthermore, this loop is recognized as an index gathering loop; thus, the values of the elements in $ind[1, k]$ defined in this loop are bounded by $[1, i-1]$. This information is used to privatize array $ind()$ and $xdt()$ in the `do i` loop, which is then determined to be parallel.

### 5.2.4 DYFESM

DYFESM is a two-dimensional finite element code from the PERFECT benchmark suite [61].

DYFESM uses a compact data structure, similar to the compress column/row storage format for

```
do i = 2, n
   do j1 = 1, i-1
      xdt(j1) = ..
   end do
   k = 0
   do j2 = 1, i-1
      if (..) then
         k = k+1
         ind(k) = j2
      end if
   end do
   do j3 = 1, k
      .. = xdt(ind(j3))
   end do
end do
```

Figure 5.5: Loop ACTFOR/do_240 in BDNA

sparse matrices, to store displacements and stresses for nodes in a grid. Almost all the computations are performed on indirectly accessed arrays. Among the computations, all vector addition loops, all SAXPY operation loops, all mixed SAXPY and vector inner product operation loops, and a preconditioning loop in a conjugate gradient algorithm can be found parallel by using the offset-length data dependence test.

The loops involved are SOLXDD/do_4, SOLXDD/do_10, SOLXDD/do_30, SOLXDD/do_50 and HOP/do_20. Loops SOLXDD/do_30, SOLXDD/do_50 and HOP/do_20 have a simple offset/length pattern, as shown in Figure 5.6. Loops SOLXDD/do_4 and SOLXDD/do_10 involve subroutine calls. After procedure inlining, they have a pattern similar to, but more complicated than, the one shown in Figure 4.18 in Example 4.2.

```
do iblock = 1, nblock
   iloc = pptr(iblock)
   do i = 1, iblen(iblock)
                  = x(iloc+i-1)
      x(iloc+i-1) = ..
   end do
end do
```

Figure 5.6: Pattern of some loops in DYFESM

The relationship between $pptr()$ and $iblen()$ can be verified statically since $pptr()$ is defined in

terms of $iblen()$ in subroutine $SETHM$. $iblen()$ is read from the input file. We manually inserted the run-time test code because our implementation cannot generate run-time tests yet.

DYFESM used a tiny input data set and suffered from the overhead introduced by parallelization. The performance of all three versions worsened when multiple processors were used (Figure 5.3.(c)). We also measured its speedups on a slower SGI Challenge machine (four 200MHz R4400 Processors) and got a speedup of 1.6 (four processors) when the extra loops were parallelized (Figure 5.3.(d)).

### 5.2.5 P3M

P3M is an N-body code that uses the particle-mesh method. This code is from NCSA.

Most of the computation time (about 88% after using vendor provided FFT library) is spent in subroutine *pp* and *subpp*, whose structures are very similar. The core is a three-perfect-loop nest, which can be parallelized. Before parallelization, several single-indexed arrays in the loop must be privatized. The outline of the core loops is shown in Figure 5.7. The simplified loop pattern is similar to that in Figure 5.5. The difference is that both $x()$ and $ind()$ are consecutively written arrays here. Therefore, the consecutively written array analysis is used twice.

### 5.2.6 Barnes & Hut TREE code

The TREE code [10] is a program that implements the hierarchical N-body method for simulating the evolution of collisionless systems [9].

The core of the program is a time-centered leap-frog loop, which is inherently sequential. At each time step, it computes the force on each body and updates the velocities and positions. About 70% of the program execution time is spent in the force calculation loop. Each iteration of the force calculation loop computes the gravitational force on a single body $p$ using a tree walk method that is illustrated in Figure 5.8.

In the tree walk code, single-indexed array *stack* is used as a stack to store tree nodes yet to be visited. Variable *sptr* is used as the stack pointer. As discussed in Sect.3.3.2, array *stack* can be privatized for the force calculation loop. As there is no other data dependence in the loop, the loop can be parallelized (i.e., the force calculation of the n bodies can be performed in parallel).

```
do i1 = 1, n
   do i2 = 1, n
      do i3 = 1, n
         p = 0
         repeat
            p = p+1
            x(p) = ..
         until (..)
         k = 0
         do j2 = 1, p
            if (..) then
               k = k+1
              ind(k) = j2
            end if
         end do
         do j3 = 1, k
            .. = x(ind(j3))
         end do
      end do
   end do
end do
```

Figure 5.7: Pattern of loop in subroutine *pp* and *subpp* in P3M

```
sptr = 1
stack(sptr) = root
while (sptr .gt. 0) do
   q = stack(sptr)
   sptr = sptr - 1
   if (q is a body) then
      process body-body interaction
   elseif (q is far enough from p) then
      process body-cell interaction
   else
      do k = 1, nsubc
         if (subp(q,k) .ne. null) then
            sptr = sptr + 1
            stack(sptr) = subp(q,k)
         end if
      end do
   end if
end while
```

Figure 5.8: Kernel `while` loop in TREE

# Chapter 6

# PARALLELIZATION OF IRREGULAR REDUCTION LOOPS ON SHARED MEMORY MACHINES

## 6.1 Introduction

Irregular reductions refer to reduction operations on some elements of an array within a loop, where the access pattern of the array in the loop may be irregular. The following loop shows a basic form of the irregular reduction.

```
integer pos(1:n)
real    data(1:m)
do i=1, n
   data(pos(i)) = data(pos(i)) + expr    (1)
end do
```

In this loop, the value of array `data()` is modified by an addition, which is a reduction operation in statement (1). Here, we assume that the expression `expr` does not contain any references to array `data()`. The array `pos()` specifies which element of array `data()` to be modified in each iteration. The access pattern of `data()` in this loop may be irregular, depending on the value of `pos()`.

Irregular reduction loops are frequently found in the kernel of many large scientific and engineering applications. In order to speedup the execution of these applications by parallelization, the irregular reduction loops must be written in or transformed to a form that can be executed

efficiently on target parallel machines. Several researchers have previously proposed different methods for parallel irregular reductions and demonstrated their effectiveness in their own experimental settings.

The goal of the work described in this chapter is to search for the "best" parallel irregular reduction method. We compare, in a uniform framework, five different parallel irregular reduction methods on shared memory machines. The methods considered can be applied automatically by a parallelizing compiler. We compared the applicability, the resource requirement, and the performance of these methods. To measure the performance, we conducted an experiment using five different applications with each application implemented by using each of the five different methods.

The rest of the chapter is organized as follows. In Section 6.2, we describe the five different methods in detail and the difficulties in applying them automatically. In Section 6.3, we explain our experimental results and compare the performance. And, we present our conclusion in Section 6.4.

## 6.2 Parallelization Methods

### 6.2.1 Program Patterns

Irregular reduction loops can take forms other than the simple one shown at the beginning of this chapter. Depending on computation problems being solved and data structures being used, irregular reduction loops often fit one of the five program patterns shown in Figure 6.1:

1. single loop with one access pattern,
2. two loops perfectly nested with one access pattern,
3. single loop with two access patterns,
4. two loops perfectly nested with two access patterns, and
5. two loops not perfectly nested with two access patterns.

```
do i=1, n
   data(pos(i)) = data(pos(i)) + expr
end do
```

(1)

```
do i=1, n
   do j=low(i), up(i)
      data(pos(i,j)) = data(pos(i,j)) + expr
   end do
end do
```

(2)

```
do i=1, n
   data(pos1(i)) = data(pos1(i)) + expr1
   data(pos2(i)) = data(pos2(i)) + expr2
end do
```

(3)

```
do i=1, n
   do j=low(i), up(i)
      data(pos1(i,j)) = data(pos1(i,j)) + expr1
      data(pos2(i,j)) = data(pos2(i,j)) + expr2
   end do
end do
```

(4)

```
do i=1, n
   do j=low(i), up(i)
      data(pos1(i,j)) = data(pos1(i,j)) + expr1
   end do
   data(pos2(i)) = data(pos2(i)) + expr2
end do
```

(5)

Figure 6.1: Five common program patterns for irregular reduction loops

### 6.2.2 Data Domain and Iteration Domain

An execution of an irregular reduction loop involves two domains, namely data domain and iteration domain. The data domain is associated with the ranges of the subscripts of reduction arrays in the loop, and the iteration domain is associated with the iteration space. Take, for example, the simple loop at the beginning of this chapter. The data domain is $[1 : m]$ and the iteration domain is $[1 : n]$. In the case of multiple-nested loops where the iteration space is multi-dimensional, we coalesce the iteration space into one-dimension [81]. If the reduction array has multiple dimensions, we can linearize the subscripts and get a one-dimensional data domain. This linearization of arrays is, however, rarely needed in practice. All the multi-dimensional reduction arrays we found had loop variant subscripts in only one dimension, such as the array `md()` in the following loop.

```
do i=1, n
   md(1,pos(i)) = md(1,pos(i)) + ...
   md(2,pos(i)) = md(2,pos(i)) + ...
end do
```

Notice that `md()` can be treated as two one-dimensional arrays (i.e., `md1()`and `md2()`), and the loop can be transformed into

```
do i=1,n
   md1(pos(i)) = md1(pos(i)) + ...
   md2(pos(i)) = md2(pos(i)) + ...
end do
```

provided that array $md()$ is always used in this way throughout the whole program.

The access patterns of reduction arrays in irregular reduction loops can be illustrated by using *access pattern images*. Suppose the iteration domain is $[1:n]$ and the data domain is $[1:m]$. The access pattern image is a $n \times m$ grid. The color of a node in the grid can be either white or black. A node at position $(i, j)$ is black if and only if the $j$-th element of the reduction array is modified in the $i$-th iteration. Otherwise, it is white. Figure 6.2.(a) shows the access pattern image for array `data()` in the loop in the same figure.

When a reduction loop has several reduction statements or several reduction arrays, we can choose to use one access pattern image for each reduction statement or reduction array, or combine some of them. The access pattern image in Figure 6.2.(a) shows the access pattern for both statements (1) and (2) in the loop. There are at most two black nodes in each row because, in each iteration, at most two elements of array `data()` are modified. Rows seven and seventeen have only one black node because $pos1(7) = pos2(7)$ and $pos1(17) = pos2(17)$.

Parallelization methods for irregular reduction loops can be put into two categories: iteration domain decomposition methods and data domain decomposition methods.

### 6.2.3 Iteration Domain Decomposition Methods

Iteration domain decomposition methods divide the iteration space into several groups, with each group being assigned to one processor, as illustrated in Figure 6.2.(b). In this chapter, we discuss three parallel irregular reduction methods that fall into this category. They differentiate in how they avoid updating the same reduction array element by two different processors at the same time.

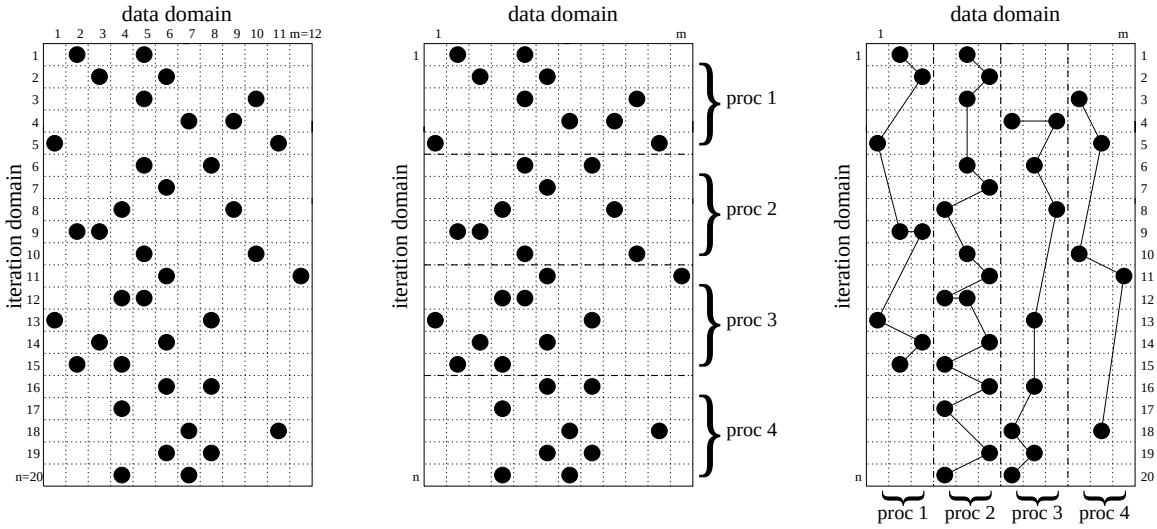
#### Critical section method

In this method, illustrated in Figure 6.3.(a), the accesses and updates of reduction array elements are enclosed by a lock/unlock pair. This method ensures that one processor does not enter the critical section if another processor is updating the same array element.

The critical section method is a simple and general method. Of all the parallelization methods, it requires the fewest modifications to the original program. It is especially useful when the irregular reduction loop has subroutine calls and complicated control flows.

```
do i=1, n
   expr = compute(i)                          (0)
   data(pos1(i)) = data(pos1(i)) + expr       (1)
   data(pos2(i)) = data(pos2(i)) - expr       (2)
end do
```

| i | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| pos1() | 2 | 6 | 5 | 7 | 1 | 8 | 6 | 9 | 2 | 5 | 6 | 5 | 1 | 3 | 4 | 6 | 4 | 7 | 6 | 4 |
| pos2() | 5 | 3 | 10 | 9 | 11 | 5 | 6 | 4 | 3 | 10 | 12 | 4 | 8 | 6 | 2 | 8 | 4 | 11 | 8 | 6 |




(a) iteration domain and data domain

(b) iteration domain decomposition

(c) data domain decomposition

Figure 6.2: Data domain vs. iteration domain

There are two kinds of overhead involved in using the critical section method: lock/unlock cost and lock waiting cost. The lock/unlock cost is high if no fast synchronization mechanism is available. The lock waiting cost depends on the granularity of the critical section and the number of conflicts, which varies with the distribution of input data.

An approach that may improve the performance of the critical section method is *statement splitting*. Suppose the `compute(i)` in Figure 6.3.(a) is a function call that takes a considerable amount of execution time. By splitting the reduction statement into two statements and hoisting the `compute(i)` out of the critical section, as shown in Figure 6.3.(b), we can overlap the executions of the expensive function calls in iterations assigned to different processors and thereby reduce the total execution time.

Another approach that may improve the performance is elimination of locks. For example, the two critical sections in the loop in Figure 6.3.(c) can be merged into one critical section, as shown in Figure 6.3.(d), thereby reducing the lock/unlock cost. The elimination of locks is especially useful in the parallel reduction loops generated automatically by compilers. This approach, however, may increase the lock waiting cost. Therefore, its impact on the total execution time depends on the access patterns, which are usually determined by the input data. P. Diniz and M. Rinard proposed a *dynamic feedback* method [25] that alternately performs *sampling* phases and *production* phases. Their method chooses the best version of the critical sections for the *production phase* based on the measurement of overhead cost in the *sampling phase*.

**Replicated copy method**

In this method, each processor uses a private copy of the whole reduction array. The parallelized code has three phases. All the private copies are initialized to the reduction identity in the first phase. In the second phase, all processors execute the reduction operation on their private copies in parallel. The last phase does the cross-processor reduction. Figure 6.4.(a) illustrates the three phases.

The biggest advantage of this method is that there are no synchronization points in the codes, except for the barriers at the end of each phase. Therefore, each phase is fully parallel. The second phase scales well when the number of processors used increases.

```
integer pos(1:n)
real    data(1:m)
lock    mylock(1:m)
doall i=1, n
   lock(mylock(pos(i)))
   data(pos(i)) = data(pos(i)) + compute(i)
   unlock(mylock(pos(i))
end do
```

(a)

```
integer pos(1:n)
real    data(1:m)
lock    mylock(1:m)
doall i=1, n
   expr = compute(i)
   lock(mylock(pos(i)))
   data(pos(i)) = data(pos(i)) + expr
   unlock(mylock(pos(i))
end do
```

(b)

```
integer pos(1:n)
real    data1(1:m)
real    data2(1:m)
lock    mylock(1:m)
doall i=1, n
   lock(mylock(pos(i)))
   data1(pos(i)) = data1(pos(i)) + expr
   unlock(mylock(pos(i))
   lock(mylock(pos(i)))
   data2(pos(i)) = data2(pos(i)) - expr
   unlock(mylock(pos(i))
end do
```

(c)

```
integer pos(1:n)
real    data1(1:m)
real    data2(1:m)
lock    mylock(1:m)
doall i=1, n
   lock(mylock(pos(i)))
   data1(pos(i)) = data1(pos(i)) + expr
   data2(pos(i)) = data2(pos(i)) - expr
   unlock(mylock(pos(i))
end do
```

(d)

Figure 6.3: Critical section method

```
integer pos(1:n)
real    data(1:m)
real    priv_data(1:m,1:num_of_proc)

// phase 1
doall i = 1, num_of_proc
   do j = 1, m
      priv_data(j,i) = 0
   end do
end do

// phase 2
doall i = 1, n
   priv_data(pos(i), proc_id)
     = priv_data(pos(i), proc_id) + expr
end do

// phase 3
doall i = 1, m
   do j = 1, num_of_proc
      data(i) = data(i) + priv_data(i,j)
   end do
end do
```

(a)

```
do i=1, n
   data(pos1(i)) = data(pos1(i)) + expr1(i)
end do
do i=1, n
   data(pos2(i)) = data(pos2(i)) + expr2(i)
end do
```

(b)

```
do i=1, n
   data(pos1(i)) = data(pos1(i)) + expr1(i)
   data(pos2(i)) = data(pos2(i)) + expr2(i)
end do
```

(c)

Figure 6.4: Replicated copy method

The replicated copy method also is simple to implement. In fact, some research parallelizing compilers, such as Polaris [14] and SUIF [37], use this method to automatically parallelize irregular reduction loops.

The biggest disadvantage of this approach is having to keep multiple copies of the whole reduction array. This not only increases the memory requirement, but also makes the execution time of the first initialization phase and the last cross-processor reduction phase proportional to the size of the reduction array. The first phase and the last phase become the bottleneck of performance when the size of the reduction array is big and a large number of processors are used. In practice, because of the parallelization overhead, the execution time of the two phases usually increases as the number of processors being used increases.

Figure 6.4.(b) shows a case where the replicated copy method can be optimized. In this case, two consecutive loops have the same iteration space and work on the same reduction array. Instead of applying the replicated copy method to each of the two loops, we can first *fuse* the two loops into one loop, as the one shown in Figure 6.4.(c), and then use the replicated copy method on the fused loop. In this way, we eliminate one cross-processor reduction phase and one initialization phase, and thereby reduce the total execution time.

### Reduction table method

In this method, each processor employs a private storage that is used as a table. The number of entries in the tables is independent of the size of the reduction array. Each entry in the table has two fields: index and value. We use $table(i).entry(j).index$ and $table(i).entry(j).value$ to represent the index field and value field in the $j$-th entry of processor $i$'s private table, respectively. The $table(i).entry(j).value$ stores the partial reduction result for the $table(i).entry(j).index$-th array element computed by processor $i$.

Similar to the replicated copy method, the reduction table method has three phases, illustrated in Figure 6.6. In the first phase, all entries in the reduction tables are initialized to the reduction identity. In the second phase, all processors execute the reduction operation in parallel. When a reduction is to be performed, the reduction table is looked up to find an entry whose index filed has the same value as the array index or, if no such entry exists, an empty entry. This lookup

process can be implemented by using a hash function. If the entry is available, then the reduction operation is performed on the value field of the entry. If the entry is not available, which means the table is full, the reduction operation is performed directly on the shared reduction array in a critical section. In the third phase, all entries that are not empty are flushed out to the shared array in critical sections.

Figure 6.5 shows an example of two processors executing a irregular reduction loop using reduction tables. Processor one executes iterations 1 through 4, and processor two executes iterations 5 through 8.

The reduction table method is a hybrid of the critical section method and the replicated copy method. Like the critical section method, it ensures the atomic operation on global shared variables; and, like the replicated copy method, each processor can accumulate its partial reduction result for some array elements before updating the shared copy. The reduction table method does not have to keep private copies of the whole reduction array. It can be set to be proportional to the reciprocal of the number of processors used so that the total size of the extra private memory is fixed. This method trades the cost of hash table calculations and the cost of critical section methods with the cost of memory operations and memory storage.

### 6.2.4 Data Domain Decomposition Method

The data domain decomposition method, as illustrated in Figure 6.2.(c), divides the data domain into several partitions and assigns one partition to each processor. A processor is responsible for updating the reduction array elements within its assigned partition.

The data domain decomposition method is similar to the *owner-computes* rule used in distributed memory programming. A processor 'owns' the array elements within the group assigned to it. Once the ownership is set, the next step is to determine which iterations to be executed on each processor. There are two possible ways to solve this scheduling problem: on-the-fly scheduling and pre-scheduling.

```
do i=1, 8
   data(pos(i)) = data(pos(i)) + expr(i)
end do
```

| i | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|
| pos(i) | 3 | 5 | 3 | 7 | 8 | 8 | 3 | 8 |

| x | 3 | 5 | 7 | 8 |
|---|---|---|---|---|
| hash(x) | 4 | 5 | 3 | 2 |

| index | | value |
|---|---|---|
| 1 | / | 0 |
| 2 | / | 0 |
| 3 | 7 | expr(4) |
| 4 | 3 | expr(1)+expr(3) |
| 5 | 5 | expr(2) |

reduction table for processor 1

| index | | value |
|---|---|---|
| 1 | / | 0 |
| 2 | 8 | expr(5)+expr(6)+expr(8) |
| 3 | / | 0 |
| 4 | 3 | expr(7) |
| 5 | / | 0 |

reduction table for processor 2

Figure 6.5: Two processors executing an irregular reduction loop by using the reduction table method.

```
// phase 1
doall i=1, num_of_proc
   do j=1, table_size
      table(i).entry(j).index = NULL
      table(i).entry(j).value = 0
   end do
end do

// phase 2
doall i=1, n
   j = hash(pos(i))
   index = table(proc_id).entry(j).index
   if (index == pos(i)) then
      table(proc_id).entry(j).value =
        table(proc_id).entry(j).value + expr
   else if (index == NULL) then
      table(proc_id).entry(j).value = data(pos(i))
      table(proc_id).entry(j).index = pos(i)
   else
      lock(my_lock(pos(i))
      data(pos(i)) = data(pos(i)) + expr
      unlock(my_lock(pos(i))
   end if
end do

// phase 3
doall i=1, num_of_proc
   do j=1, table_size
      k = table(i).entry(j).index
      if ( k != NULL ) then
         lock(my_lock(k))
         data(k) =
           data(k) + table(i).entry(j).value
         unlock(my_lock(k))
      end if
   end do
end do
```

Figure 6.6: Reduction table method

### On-the-fly scheduling method

In this method, each processor traverses all the iterations and checks whether it owns the reduction array element referenced in the current iteration. If true, the processor executes the operation; otherwise, it skips the operation. Figure 6.7.(a) shows the most simple case of using this method. In this case, the loop contains only one statement, which is a reduction statement.

In the general case where the loop body contains multiple reduction statements and multiple access patterns, the *program slicing* method is used to find statements that are in the same slice with each of the reduction statements [79, 71]. All statements in the same slice are then guarded by the same ownership checking. Statements that are in multiple slices are guarded by the conjunction of their corresponding ownership checkings[1]. For example, in the reduction loop in Figure 6.7.(b), statements (1) and (4) are found in the same slice; statements (2) and (5) are in the same slice; and statement (3) is in both slices. Figure 6.7.(c) shows the parallel version. Another approach is to distribute the loop into several loops with each loop containing one slice in the original loop body.

The on-the-fly scheduling method is easy to understand. It also is easy to use when the reduction loop is simple. The transformation becomes complicated, however, when the loop body contains multiple access patterns, and therefore, multiple slices. The use of *if* branches also complicates the instruction scheduling for the backend compiler, and is a primary performance limiter for processors with pipelined functional units. Another disadvantage of the on-the-fly scheduling method is that each processor has to go through the whole iteration space and test ownership in every iteration. It is inefficient when the iteration domain is large.

### Pre-scheduling method

The pre-scheduling method has two phases: the scheduling phase and the execution phase, as shown in Figure 6.8.(a). In the scheduling phase, the sets of iterations that modify the reduction array elements owned by each processor are collected and put into *schedule lists*. In the execution phase, all processors execute in parallel, with each processor executing the iterations on its own schedule list.

[1] In some cases, it would be more efficient to replicate the computation instead of using an *if* branch because, in some architectures, the penalty of branch misprediction is more costly than computation.

```
Sequential Version:
do i=1, n
   data(pos(i)) = data(pos(i)) + expr
end do
```

```
do i=1, n
   t1 = a(i)**2                                  (1)
   t2 = b(i)**2                                  (2)
   t3 = c(i)**2                                  (3)
   data(pos1(i)) = data(pos1(i)) + t1 + t3       (4)
   data(pos2(i)) = data(pos2(i)) + t2 + t3       (5)
end do
```

(b)

```
Parallel Version:
doall i=1, number_of_processors
   do j=1, n
      if (own(pos(j),i)) then
         data(pos(j)) = data(pos(j)) + expr
      end if
   end do
end do
```

(a)

```
doall p=1, number_of_processors
   do i=1, n
      if (own(pos1(j),i) or own(pos2(j),i)) then
         t3 = c(i)**2
      end if
      if (own(pos1(j),i)) then
         t1 = a(i)**2
         data(pos1(j)) = data(pos1(j)) + t1 + t3
      end if
      if (own(pos2(j),i)) then
         t2 = b(i)**2
         data(pos2(j)) = data(pos2(j)) + t1 + t3
      end if
   end do
end do
```

(c)

Figure 6.7: On-the-fly scheduling method

```
pre_schedule(pos, schedule_list)

doall i=1, number_of_processors
      do j in the schedule_list(i)
         data(pos(j)) = data(pos(j)) + expr
      end do
end do
```

(a)

```
integer start(1:number_of_processors)
integer count(1:number_of_processors)
integer list(1:n)


doall i=1, number_of_processors
      k = start(i)
      do j=1, count(i)
         data(pos(k)) = data(pos(k)) + expr
         k = list(k)
      end do
end do
```

(b)

```
nump = number_of_processors
start(1:nump) = 0
count(1:nump) = 0
prev(1:nump) = 0
p_start(1:nump,1:nump) = 0
p_count(1:nump,1:nump) = 0
p_end(1:nump,1:nump) = 0

doall i=1, n
      p = owner(pos(i))
      c = p_count(p,proc_id)
      if ( c == 0 ) then
         p_start(p,proc_id) = i
      else
         list(p_end(p,proc_id)) = i
      end if
      p_end(p,proc_id) = i
      p_count(p,proc_id) = c+1
end do

doall i=1, nump
      do j=1, nump
         if (p_start(i,j) != 0) then
            if (start(i) == 0) then
               start(i) = p_start(i,j)
            else
               list(prev(i)) = p_start(i,j)
            end if
            prev(i) = p_end(i,j)
         end if
         count(i) = count(i) + p_count(i,j)
      end do
end do
```

(c)

Figure 6.8: Pre-scheduling method

Here, we present a method to construct the scheduling lists, which is based on the *loop index prefetching* technique proposed by E. Gutierrez, O. Plata, and E. L. Zapata [36]. In this method, the scheduling lists are represented by three arrays: `list()`, `start()`, and `count()`. The `list()` is an array used as a linked list that stores the scheduled iteration numbers for all processors. `start(p)` gives the first iteration to be executed by the $p$-th processor. And, `count(p)` is the number of iterations to be executed by the $p$-th processor. An example of the execution phase that uses these three arrays is shown in Figure 6.8.(b).

The pre-scheduling phase is shown in Figure 6.8.(c). This phase has three steps, all of which are fully parallel. In the first step, the schedule list and the partial schedule lists used by each processor are initialized. In the second phase, all processors work in parallel to construct the partial schedule lists. And, in the last step, the partial schedule lists are combined to form the final schedule list. The asymptotic execution time of the pre-scheduling phase is $O(\frac{n}{p} + p)$, where $n$ is the size of the iteration domain and $p$ is the number of processors used.

In many cases where the dynamic nature of the problem changes slowly, the array $pos()$ does not change for several invocations of the reduction loop. In these cases, the same schedule list can be reused, and, therefore, the cost of the pre-scheduling phase is amortized.

The execution phase has good data locality if block partitioning is used to decompose the data domain. Load balancing, however, can be a problem if the partition is made without considering the distribution of the reduction array. For example, for the partition in Figure 6.2.(c), processor 4 executes only four iterations, while processor 2 has to execute fifteen iterations. H. Han and C. W. Tseng suggest that the recursive coordinate bisection (RCB) algorithm be used to partition data domain in this case [40].

The pre-scheduling method also is complicated by another issue: multiple reductions. Consider the irregular reduction loop in Figure 6.2. Suppose the data domain is partitioned as shown in Figure 6.2.(c). Then, both statements (1) and (2) can be executed by processor 1 in iteration 9, by processor 2 in iteration 7, 12 and 17, and by processor 3 in iteration 4. For other cases, however, statement (1) or (2) can be executed by only one processor. For example, although both processors 1 and 2 will execute iteration 1, statement (1) in iteration 1 should be executed only by processor 1 and not by processor 2. And, statement (2) should be executed only by processor 2 and not

by processor 1 because `data(2)`, which is modified in statement (1), is owned by processor 1, and `data(5)`, which is modified in statement (2), is owned by processor 2.

There are three ways to solve this problem. The first one is to put ownership checkings before statement (1) and before statement (2). An iteration is in a processor's schedule list if the processor owns the array element modified by either statement (1) or (2) in that iteration. In the execution phase, a processor will skip statement (1) or (2) according to the result of its ownership checking. As in the on-the-fly scheduling method, the use of *if* branches here hampers the performance of the loop body. Another method is to use three schedule lists for each processor: one for iterations in which statement (1) should be executed; another for iterations in which statement (2) should be executed; and, yet another for iterations in which both statements should be executed. The shortcoming of this method is that the number of schedule lists increases exponentially with the number of reductions in the loop body. The last method is to distribute the loop. This is not always legal and it works well only when the loop body can be easily sliced. Whatever method is used, the overhead of redundant computation cannot be avoided. For example, the instance of statement (0) in iteration 1 is executed twice, once by processor 1 and once by processor 2.

Parallelizing multiple nested reduction loops by using the pre-scheduling method also is complicated because the iteration domain becomes multi-dimensional in this case. One way to handle this problem is to coalesce the multiple nested loops into a single level loop. Imperfectly nested loops are first transformed to perfectly nested loops by using loop distribution or statement sinking [81].

The pre-scheduling method should not be confused with the *inspector/executor* model used on distributed memory machines, which was pioneered by the CHAOS runtime system [67]. Their approach is an iteration domain decomposition method. The inspector identifies the nonlocal data needed by each processor and generates a communication schedule. The executor gathers nonlocal data to local buffers using the communication schedule, performs the computation on the local buffers, and then scatters the results to other processors. In our pre-scheduling method, no data gathering or scattering is needed. Computation is always performed on the "local" data.

| | Memory Requirement | Execution Time |
|---|---|---|
| Critical Section | $O(m)$<br>for lock variables | Best: $(C_{cal} + C_{lock}) \times \frac{n}{p}$<br>Worst: $(C_{cal} + C_{lock}) \times n$ |
| Replicated Copy | $O(m \times p)$<br>for replicated copies | $(C_{init} + C_{merge}) \times m + C_{cal} \times \frac{n}{p}$ |
| Reduction Table | $O(n_{entry} \times p)$<br>for reduction tables | Best: $(C_{init} + C_{merge}) \times n_{entry} + (C_{cal} + C_{tab}) \times \frac{n}{p}$<br>Worst: $C_{init} \times n_{entry} + C_{merge} \times n_{entry} \times p + C_{cal} \times n$ |
| On-the-fly Scheduling | None | Best: $C_{chk} \times n + C_{cal} \times \frac{n}{p}$<br>Worst: $C_{chk} \times n + C_{cal} \times n$ |
| Pre-scheduling | $O(n + 2p)$<br>for schedule lists | Best: $\frac{V_{sched}}{n_{reuse}} + C_{cal} \times \frac{n}{p}$<br>Worst: $\frac{V_{sched}}{n_{reuse}} + C_{cal} \times n$ |

Table 6.1: Memory requirements and execution times of the five parallel irregular reduction methods

### 6.2.5 Performance Analysis

In this subsection, we discuss the theoretical performance of each parallel irregular reduction method. To be concise, we consider only the loop pattern (1) in Figure 6.1.

We assume the size of iteration domain is $n$, the size of data domain is $m$, and the execution time of the loop body is $C_{cal}$. As a result, the execution time of the sequential version is $C_{cal} \times n$.

Table 6.1 shows the memory requirements and the execution times of each method. We assume $p$ processors are used. In Table 6.1, $C_{lock}$ is the cost of a pair of lock/unlock operations; $C_{init}$ is the cost of setting the value of a reduction array element to the reduction identity; $C_{merge}$ is the cost of adding the partial sum of a reduction array element computed by one processor to the global copy of the reduction array; $n_{entry}$ is the number of entries in a reduction table; $C_{tab}$ is the cost of the reduction table lookup operation; $C_{chk}$ is the cost of checking the ownership of a reduction array element; $V_{sched}$ is the cost of pre-scheduling cost whose asymptotic execution time is $O(\frac{n}{p} + p)$; and $n_{reuse}$ is the number of times a schedule is reused in the pre-scheduling method.

The on-the-fly-scheduling requires no extra memory space. The size of memory space used in the pre-scheduling method is proportional to the size of the iteration domain. For the critical section method and replicated copy method, the size is proportional to the size of the data domain. For the reduction table method, as well as the replicated copy method, the size is proportional to the number of processors used. In most real programs, the replicated copy method requires the largest amount of memory space among all the methods.

Except for the replicated copy method, we presented the best and worst possible execution times for each method. The reason is that, except for the replicated copy method, the actual execution time of the methods we discussed depend on the distribution of the index arrays. Also note that other effects, such as cache behavior, have not been taken into account as we assume a perfect memory system.

The execution time in Table 6.1 should be read carefully. Comparing the best or worst execution time of two methods is not useful because, given an input data set, two methods seldom can both achieve their own best or worst execution time. The best and worst times are listed here so that we can see the dominating factor in the execution time.

Ideally, we would like to compare the average execution time of these methods. Such a quantitative comparison also is not useful, however, unless we know the distribution of the characteristics of the input data commonly used in the real world. There is no perfect answer. In our work, we took another approach. We took several programs and the associated data from some other researchers who had studied the irregular reduction problem, parallelized these problems using the five different methods, and then compared the execution times on a parallel machine. The conclusion of our study is based on our experiments. Despite all its caveats, our experiments gave us an important qualitative view of which method was better under which condition. The experiments are discussed in the next section.

## 6.3 Experiments

### 6.3.1 Experimental Setting

We evaluated the five different parallel irregular reduction methods on a SGI Origin2000 using one to thirty-two MIPS R10000 processors (195MHz, 32KB instruction cache, 32KB data cache, 4MB 2nd cache, 14GB memory, IRIX64 6.5).

All programs were written in FORTRAN 77 with SGI multi-processor directives inserted. The default *first touch* policy was used for the memory placement scheme. The programs were compiled using SGI MIPSPro F77 compiler (version 7.3.1.m) with '-mp' and '-O2' options.

We tested five different applications that have irregular reduction cores. Each application fits

| Data Set | Size of Data Domain | Size of Iteration Domain | Iteration / Data Ratio | Description | Source | Used in Test |
|---|---|---|---|---|---|---|
| 50k | 50,000 | 500,000 | 100 | Randomly generated by a linear congruential generator | Randomly generated | SIR |
| 5k | 5,000 | 500,000 | 10 | Randomly generated by a linear congruential generator | Randomly generated | SIR |
| BCSSTK30 | 28,914 | 1,036,208 | 36 | BCS structural engineering matrix (large eigenvalue problem) | Harwell-Boeing collection | SIR, VetMat |
| PSMIGR3 | 3,140 | 543,162 | 173 | Inter-country migration (doubly stochastic) | Harwell-Boeing collection | SIR, VecMat |
| 891rs | 891,900 | 7,831,490 | 9 | Irregular mesh generated by EULER (sorted) | University of Malaga | EULER |
| 891 | 891,900 | 7,831,490 | 9 | Irregular mesh generated by EULER (colored) | University of Malaga | EULER |
| 1161rs | 1,161,981 | 10,163,580 | 9 | Irregular mesh generated by EULER (sorted) | University of Malaga | EULER |
| 1161 | 1,161,981 | 10,163,580 | 9 | Irregular mesh generated by EULER (colored) | University of Malaga | EULER |
| NBF100 | 32,000 | 3,200,000 | 100 | Mesh generated by NBF | University of Maryland | NBF |
| NBF50 | 32,000 | 1,600,000 | 50 | Mesh generated by NBF | University of Maryland | NBF |
| SF10 | 7,294 | 52,216 | 7 | Unstructured 3D finite element model | Carnegie Mellon University | Spark98 |
| SF5 | 30,169 | 220,546 | 7 | Unstructured 3D finite element model | Carnegie Mellon University | Spark98 |

Table 6.2: Test data sets

one of the program patterns described in Section 6.2.1. All five parallel irregular reduction methods are tested for each application.

The input data sets used in the tests are listed in Table 6.2. At least two of them were used for each application. The property of each input data set is described in the following subsections when the programs that use the data sets are explained.

The block partitioning method was used in both the iteration domain decomposition and the data domain decomposition, as shown in Figure 6.2.(b) and (c). We measured the number of iterations executed by each processor and found the load was even among all processors.

For the reduction table method, the number of entries in a reduction table was set to 4096. The hash function was $hash(k) = k \ mod \ 4096$.

### 6.3.2 Simple Irregular Reduction (SIR)

This program is a straightforward implementation of the simple irregular reduction loop of pattern (a) in Section 6.2.1. The kernel of this program is shown in Figure 6.9. The `comput()` is a function call, the execution time (granularity) of which can be controlled by an input parameter. In our experiments, we chose the parameters to be 2, 10, and 20, which corresponded to the granularity of 0.7 micro-seconds, 4.8 micro-seconds, and 18.5 micro-seconds, respectively. Four input data sets, namely 50k, 5k, BCSSTK30, and PSMIGR3, were used in this test.

#### Data set: 50k

Data set 50k is generated by a linear congruential generator. The values of `pos()` are randomly distributed in the range of 1 to 50,000. The access pattern image of data set 50k in the simple irregular reduction loop is shown in Figure 6.14.(50k). This access pattern image should not be confused with the structure image of the sparse matrix. The size of the access pattern image shown is not proportional to the sizes of the iteration domain and data domain. And, there is only one black node in each row, as one iteration accesses only one element of $data()$.

The experimental results are shown in Figure 6.15. Subfigure (a1) compares the execution time of the five parallel irregular reduction methods when the granularity is 0.7 micro-seconds. The different execution time for one to thirty-two processors are listed from the left to the right. The horizontal line marks the execution time of the sequential version of the program. Subfigure (b1) shows the breakdown of execution time for each method. The three parts (from bottom to top) in each bar of the replicated copy method represent the three phases in this method. The three parts in each bar of the reduction table method also represent the three phases in the reduction table method. The lower part in each bar of the pre-scheduling method is for the scheduling phase, and the upper part is for the execution phase. Subfigures (a2)/(b2) and (a3)/(b3) are the corresponding results for the cases when the granularity is 4.8 micro-seconds and 18.5 micro-seconds, respectively.

The replicated copy method is the best in all tested cases. However, the cost of the first and third phases actually increases with the number of processors used and dominates the execution time when 32 processors are used and the granularity of loop body is small, as shown in subfigure (b1).

The performance of the critical section method and the reduction table method are almost the same. This is due to the fact that data set 50k has very low data locality and many lock contentions occurred during the execution.

Both the pre-scheduling phase and the execution phase of the pre-scheduling method scale very well. This method and the on-the-fly scheduling method have the second and third best performance in this test, respectively.

All methods perform equally well when the granularity of loop body is large, as shown in Figure 6.15.(a3).

**Data set: 5k**

Data set 5k is generated by the same linear congruential generator that generates data set 50k. In 5k, values of `pos()` are randomly distributed in the range of 1 to 5,000. The access pattern image is shown in Figure 6.14.(5k).

The experimental results are shown in Figure 6.16. The most obvious difference between the result of this test and that of the 50k is that the performance of the reduction table method is much better in this test, although the distribution of the data in both data sets is almost the same. The reason is simple: the size of the data domain for data set 5k is 5,000, which is very close to the number entries (4,096) in the reduction table. In this case, the reduction table works almost the same as the replicated copy method.

**Data set: BCSSTK30**

BCSSTK30 is a sparse matrix from the Harwell-Boeing collection. Most of the nonzero elements are near the diagonal of the matrix. BCSSTK30 is stored in the *sorted compressed row storage* (SCRS) format. When a sparse matrix is stored in the SCRS format, all the nonzero elements of the matrix are stored in a one-dimensional data array row by row, from top to bottom. And, within a row, the nonzero elements are stored one by one, from left to right. Here, we use $matdata(1:n)$ to denote the one-dimensional data array, where $n$ is the number of nonzero elements in the sparse matrix. The SCRS format is a special form of the *compressed row storage* (CRS) format, in which no order is assumed between the elements within a row of the sparse matrix stored in the $matdata()$

array.

In this test, the loop traverses all elements of $matdata(1:n)$ from $matdata(1)$ to $matdata(n)$. $pos(i)$ gives the column number of element $matdata(i)$. Figure 6.14.(BCSSTK30) shows the image of the access pattern of $data()$ in the simple irregular reduction loop.

The experimental results are shown in Figure 6.17. In this test, the performance of the critical section method is much better than those in the previous two tests. It is even better than the two data domain decomposition methods in most cases. As shown in Figure 6.14.(BCSSTK30), the access pattern has good data locality. Very few lock contentions are expected. The reduction table method also performs well for the same reason.

#### Data set: PSMIGR3

PSMIGR3 is another sparse matrix for the Harwell-Boeing collection. It is also stored in the SCRS format, and the loop traverses all its nonzero elements. The access pattern image is shown in Figure 6.14.(PSMIGR3).

The experimental results are shown in Figure 6.18. As the property of this data set is similar to that of data set 5k, the behaviors of all the methods are similar to those in the test using data set 5k.

### 6.3.3 VecMat

VecMat implements a multiplication of a full row vector and a sparse matrix stored in the CRS format. The product is stored in another full row vector.

The reduction kernel is shown in Figure 6.10. The loop_do_i scans the $n$ rows of the matrix. The loop_do_j scans all the nonzero elements of each row. The multiplier $vec(i)$ is common to all elements in row $i$ of the matrix. The results of the partial inner products are accumulated in $c(pos(j))$. This program fits pattern (2) in Section 6.2.1. The two nested loops are coalesced when the pre-scheduling method is used.

Two data sets, namely BCSSTK30 and PSMIGR3, are used. The access pattern images are shown in Figure 6.14.(BCSSTK30) and (PSMIGR3). The experimental results are shown in Figure 6.19 and Figure 6.20, respectively.

```
do i=1, n
   value = comput(i)
   data(pos(i)) = data(pos(i)) + value
end do
```

Figure 6.9: Simple irregular reduction

```
do i=1, n
   do j=ptr(i), ptr(i+1)-1
      c(pos(j)) = c(pos(j)) + vec(i)*mat(j)
   end do
end do
```

Figure 6.10: VecMat

```
do i=1,numEdges
   n1 = edge(1,i)
   n2 = edge(2,i)
   r1 = func1(i,n1,n2)
   r2 = func2(i,n1,n2)
   vd(1,n1) = vd(1,n1) + r1
   vd(2,n1) = vd(2,n1) + r2
   vd(3,n1) = vd(3,n1) + r3
   vd(1,n2) = vd(1,n2) - r1
   vd(2,n2) = vd(2,n2) - r2
   vd(3,n2) = vd(3,n2) - r3
enddo
```

Figure 6.11: Kernel of EULER

```
do i = 1, natoms
   do p = 1, inb(i)
      j = partners(p, i)
      force = ((x(i) - x(j)) ** (-6)) / 1000
      f(i) = f(i) + force                          (1)
      f(j) = f(j) - force                          (2)
   end do
end do
```

Figure 6.12: NBF

```
do i=1, nodes
   Anext = matrixptr(i)
   Alast = matrixptr(i+1)

   sum1=k(1,1,Anext)*v(1,i)+k(2,1,Anext)*v(2,i)+k(3,1,Anext)*v(3,i)
   sum2=k(1,2,Anext)*v(1,i)+k(2,2,Anext)*v(2,i)+k(3,2,Anext)*v(3,i)
   sum3=k(1,3,Anext)*v(1,i)+k(2,3,Anext)*v(2,i)+k(3,3,Anext)*v(3,i)

   do j=Anext+1, Alast-1
      col = matrixcol(j)

      sum1=sum1+k(1,1,j)*v(1,col)+k(2,1,j)*v(2,col)+k(3,1,j)*v(3,col)
      sum2=sum2+k(1,2,j)*v(1,col)+k(2,2,j)*v(2,col)+k(3,2,j)*v(3,col)
      sum3=sum3+k(1,3,j)*v(1,col)+k(2,3,j)*v(2,col)+k(3,3,j)*v(3,col)

      tmp1=k(1,1,j)*v(1,i)+k(1,2,j)*v(2,i)+k(1,3,j)*v(3,i)
      tmp2=k(2,1,j)*v(1,i)+k(2,2,j)*v(2,i)+k(2,3,j)*v(3,i)
      tmp3=k(3,1,j)*v(1,i)+k(3,2,j)*v(2,i)+k(3,3,j)*v(3,i)
      w(1,col) = w(1,col)+tmp1
      w(2,col) = w(2,col)+tmp2
      w(3,col) = w(3,col)+tmp3
   end do

   w(1,i) = w(1,i)+sum1
   w(2,i) = w(2,i)+sum2
   w(3,i) = w(3,i)+sum3
end do
```

Figure 6.13: Spark98

The granularity of the inner loop body is very small: about 0.08 micro-seconds. Therefore, the overhead of the parallel methods becomes significant. For both data sets, when fewer than four processors are used, none of the five parallel versions have shorter execution times than the sequential version. In BCSSTK30, the parallel versions, except for the one for on-the-fly scheduling, finally outperform the sequential version when thirty-two processors are used. In PSMIGR3, however, only the replicated copy version does better than the sequential version when thirty-two processors are used.

This experiment also demonstrates the worst scenario for the on-the-fly scheduling method: the overhead introduced by on-the-fly ownership checking is larger than the execution time of the original computation itself. Since each processor sweeps through all iterations, the total execution time will not go down by using multiple processors.

Similar to the test case of SIR, the good data locality in data set BCSSTK30 makes the critical section method perform better in BCSSTK30 than in PSMIGR3.

### 6.3.4 EULER

EULER, from the HPF-2 motivating application suite, solves the differential EULER equations on irregular meshes. The code used in this experiment is a simplified version used in [36]. The kernel, shown in Figure 6.11, is a loop that computes physical magnitudes over the mesh edges, each of which is defined by two nodes. The reduction array `vd()` is a two-dimensional array which can be treated as three one-dimensional arrays. There are two access patterns of the reduction array in the loop: one defined by `edge(1,:)` and the other defined by `edge(2,:)`. Three scheduling lists are used in the pre-scheduling method. The reduction loop fits pattern (3) in Section 6.2.1.

The performance is tested on two irregular meshes, one with 891,900 nodes and one with 1,161,981 nodes. Both meshes were generated by the original EULER code. The nodes of the input mesh are considered as particles, and the edges are the neighboring interactions between them. Both meshes have a connectivity *(number of edges/number of nodes)* of 8. Two versions of each mesh were used: colored and sorted. We use 891 and 1161 to represent the two colored versions, and 891rs and 1161rs to represent the two sorted versions. The sorted version has higher data locality than the colored version. The access pattern images shown in Figure 6.14.(891rs),

(1161rs), (891), and (1161) illustrate the difference.

The experimental results are shown in Figures 6.21, 6.22, 6.23 and 6.24. The difference in size between the two meshes does not make much difference in the relative performance of the five methods. This can be seen by comparing Figures 6.21/6.22 with Figures 6.23/6.24. The data locality makes the largest difference. First, in the sorted version, the critical section method and reduction table method are among the best, especially when sixteen or thirty-two processors are used. In the colored version, they are the worst. Second, for both data sets, the execution time of the sequential colored version is almost five times as long as that of the sequential sorted version.

As in the VectMat test, the small granularity of the loop body in the sorted version leads to a speedup of 1 in the on-the-fly scheduling method.

### 6.3.5 NBF

NBF, provided by H. Han and C. W. Tseng [40], is a kernel abstracted from the GROMOS molecular dynamics benchmark [77]. In NBF, each molecule has a list of partners. NBF uses two nested loops shown in Figure 6.12. The outer loop iterates each molecule, and the inner loop goes over each partner of this molecule. The program fits pattern (4) in Section 6.2.1.

In the pre-scheduling method, the loop nest was first distributed into two loop nests, one with statement (1) and the other with statement (2), and the latter then was coalesced into a single level loop, as shown below.

```
do i = 1, natoms
   do p = 1, inb(i)
      j = partners(p,i)
      force = ((x(i) - x(j)) ** (-6)) / 1000
      f(i) = f(i) - force
   end do
end do

do k = 1, all
   i = geti(k)
   p = getp(k)
   j = partners(p,i)
   force = ((x(i) - x(j)) ** (-6)) /1000
   f(j) = f(j) - force
end do
```

In the above loop `do k`, array $geti()$ and array $getp()$ are used to store loop indices $i$ and $j$ in

the original loop. Also note that the above loop `do i` is not an irregular reduction loop, which can be parallelized easily.

We use two data sets, one with 100 partners for each molecule and the other with 50 partners for each molecule. Both have a total of 32000 molecules. Given the parameters, NBF generated these two data sets automatically. The access pattern images are shown in Figure 6.14.(NBF50) and (NBF100).

The experimental results are shown in Figures 6.25 and 6.26. The replicated copy method has the best performance among the five parallel reduction methods in both tests.

In both tests, the cost of the pre-scheduling phase of the pre-scheduling method is so large that the pre-scheduling method has the worst performance in almost all test cases. The exceptions are the tests using 16 and 32 processors with data set NBF50, where the pre-scheduling method has the second worst performance.

The difference between the two results is that, in the test using the data set NBF50, the performance of the critical section method and the reduction table method are better than those in the test using the data set NBF100. The reason is that the access of the NBF50 has better data locality than the access of the NBF100.

### 6.3.6 Spark98

Spark98 is a collection of sparse matrix-vector product kernels for shared memory and message passing machines [57]. It was built as a tool for understanding the performance of irregular codes on different parallel systems. It consists of five versions of a sparse matrix-vector multiplication program: one sequential version, two shared memory versions, one message passing version, and one hybrid shared memory and message passing version. One of the shared memory versions uses the critical section method, and the other uses the replicated copy method. The original codes were written in C. We rewrote the sequential version in FORTRAN 77 and parallelized it using the five different methods discussed in this chapter.

Spark98 uses two input data sets. The two meshes, called sf10 and sf5, were two unstructured three-dimensional finite element models of the earth underneath the San Fernando Valley. Each mesh is represented by a *stiffness matrix*, which contains a $3 \times 3$ submatrix for each pair of nodes

connected by each edge of the mesh (including self-edges). The stiffness matrix has the dimension $3n \times 3n$ if the mesh has $n$ nodes. The stiffness matrix is stored in a symmetric CRS format.

The reduction loop of Spark98 is shown in Figure 6.13. The sparse matrix-vector product computation involves irregular reductions because Spark98 exploits symmetry and stores only the upper part of the stiffness matrix. This loop fits pattern (5) in Section 6.2.1. We distribute the loop into two loops before applying the pre-scheduling method. The access pattern images are shown in Figure 6.14.(SF10) and (SF5).

The experimental results are shown in Figures 6.27 and 6.28. Because both sf10 and sf5 have good data locality, the critical section method and reduction table method are the two best ones.

For the replicated copy method, the cost of the initialization phase and the cross-processor reduction phase is so high that the replicated copy method has the worst performance when sixteen or thirty-two processors were used. Both data set SF10 and data set SF5 have an *iteration domain/data domain* ratio of 7, which is the smallest of all the data sets used in the experiments. Everything else being equal, the smaller the *iteration domain/data domain* ratio, the higher the overhead cost of the initialization phase and the cross-processor reduction phase. The same situation also was observed in the test of EULER with data sets 891rs and 1161rs, which have good data locality and a low *iteration domain/data domain* ratio of 9.

### 6.3.7 Summary of Experiments

The results of the experiments can be summarized as follows:

1. The program model has an influence on how easy it is to apply the parallelization methods, but it usually causes little difference in the relative performance of the different parallelization methods.

2. The access pattern and the distribution of the reduction array affects the relative performance.

3. The critical section method and the reduction table method have very good performance when the access pattern has good data locality and few lock contentions are met. And, in our test, we found both methods performed equally well in this case, although the critical section method is simpler to implement than the reduction table method.

4. The replicated copy method has good performance in most cases. In our experiment, we found the cost of the initialization phase and the cross-processor reduction phase increased as more processors were used. The performance decreases when the *iteration domain / data domain* ratio is small.

5. The reduction table method may have performance similar to the replicated copy method when the table size is almost as large as the reduction array.

6. The speedups of the on-the-fly scheduling method increased as the number of processors used increased from two to eight. The speedups flattened when more processors were used because of the non-scalable overhead of real-time checking.

7. When the data domain is partitioned in block, the pre-scheduling method has good data locality, and the execution phase usually has good performance and scales very well with the number of processors used. The pre-scheduling method is a good choice if the schedule can be reused.

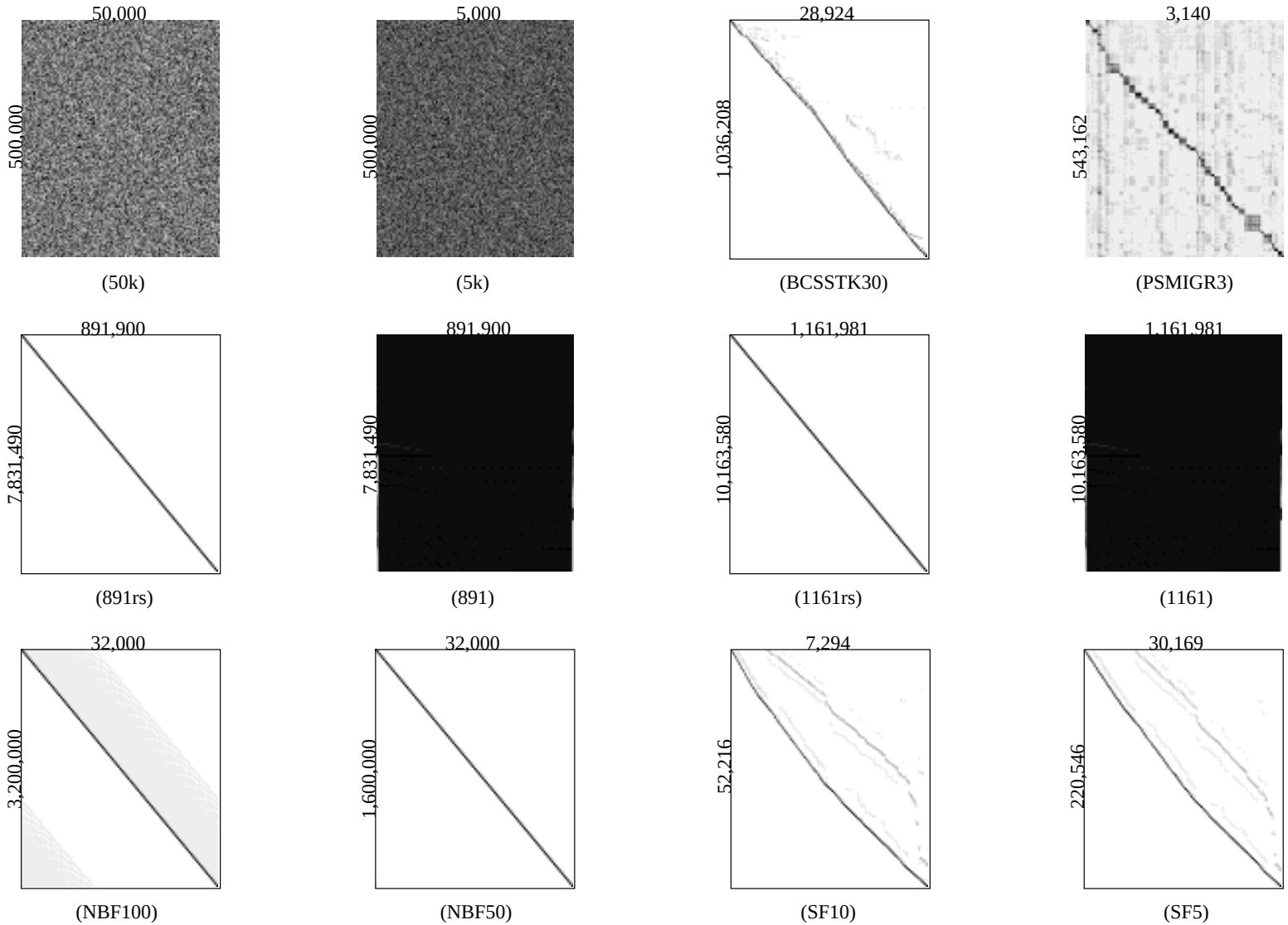
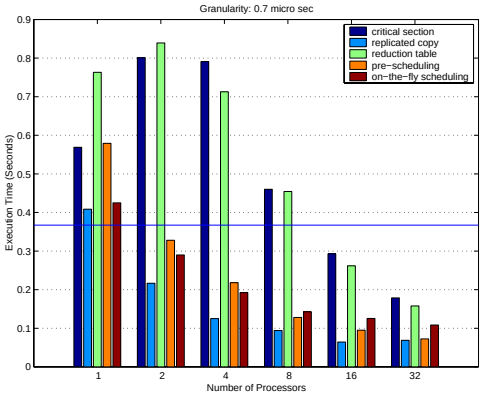
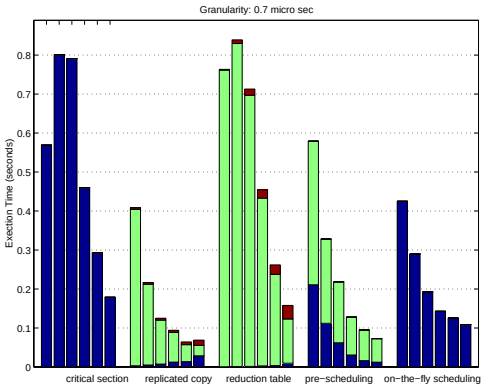


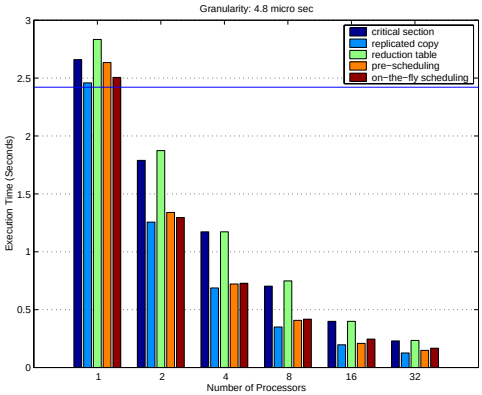

Figure 6.14: Access pattern images



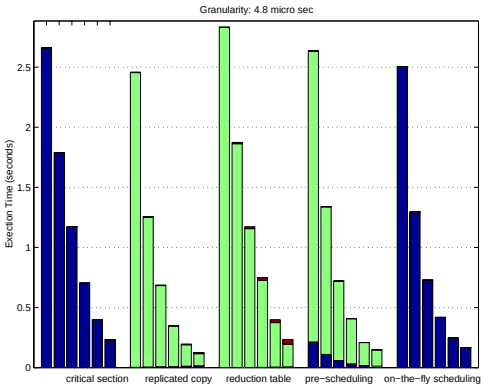

(a1)



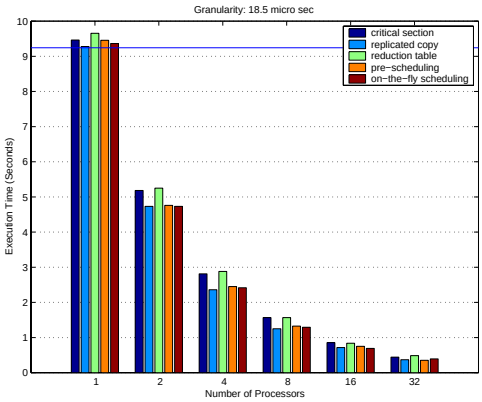

(b1)



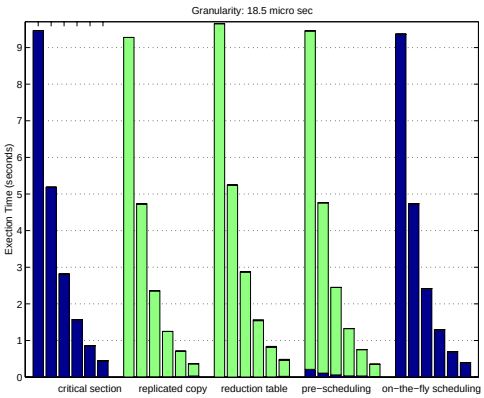

(a2)




(b2)

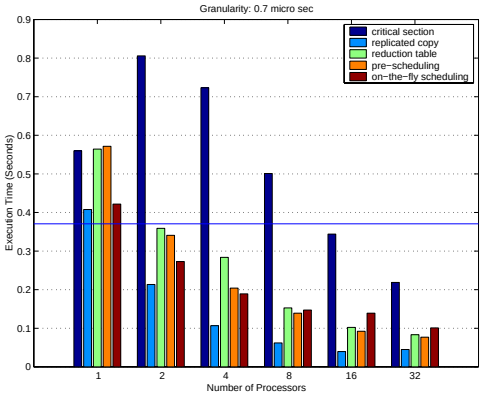



(a3)

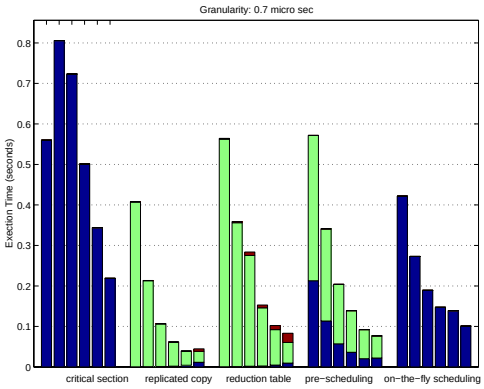



(b3)

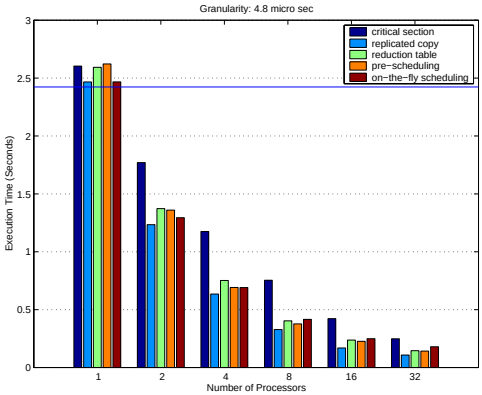
Figure 6.15: Simple: 50k



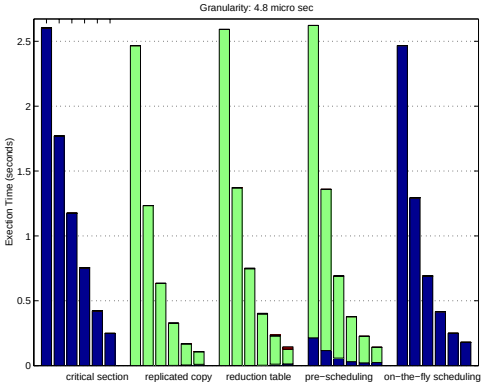

(a1)



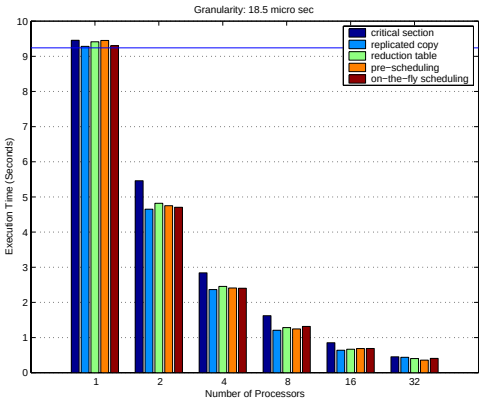

(b1)



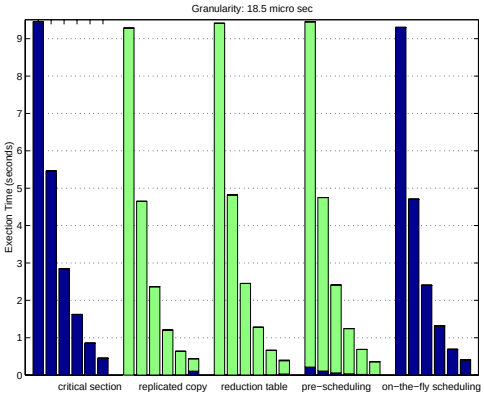

(a2)




(b2)

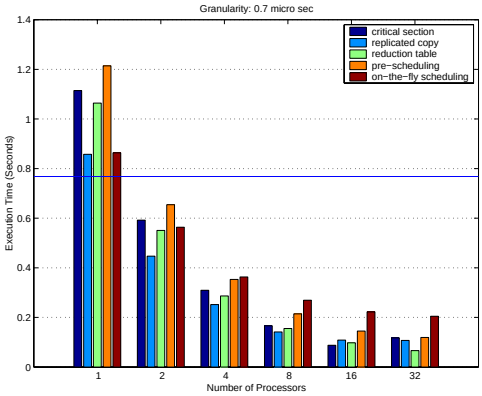



(a3)

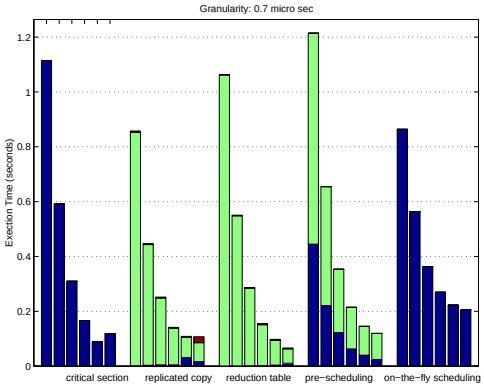



(b3)

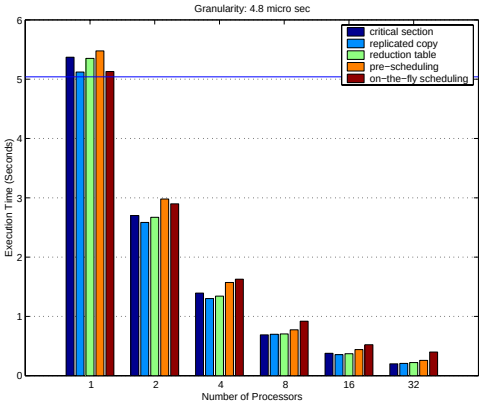
Figure 6.16: Simple: 5k



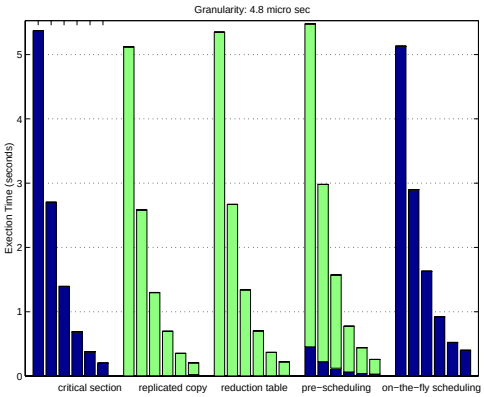

(a1)



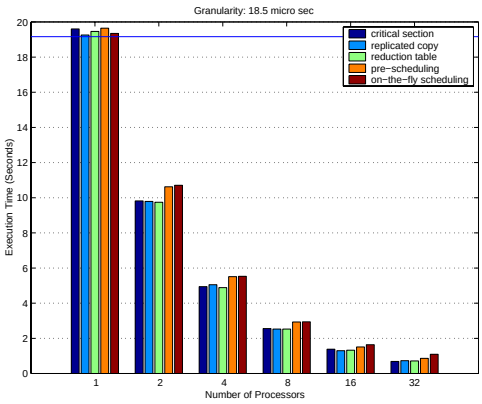

(b1)



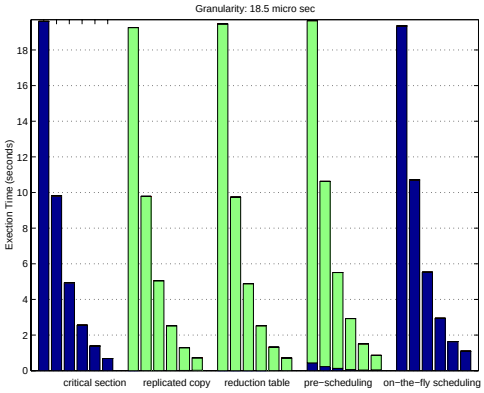

(a2)




(b2)

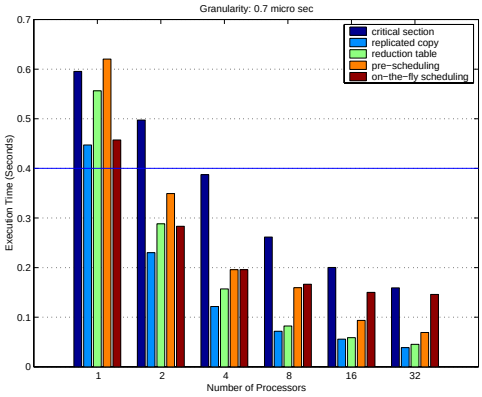



(a3)

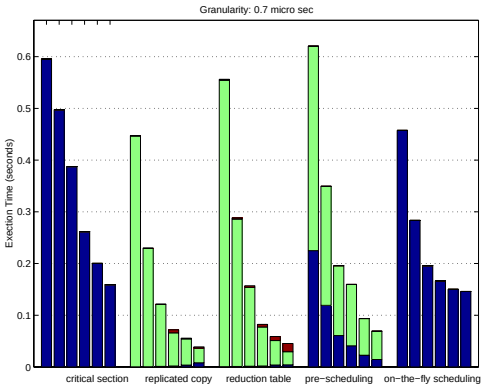



(b3)

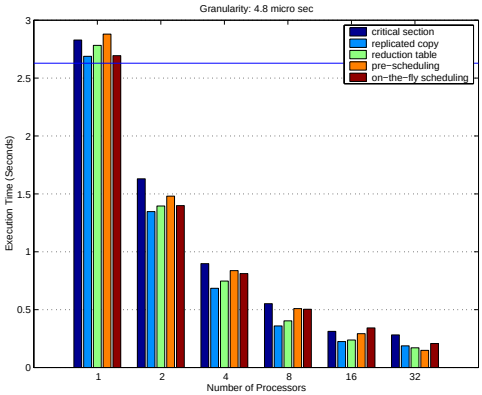
Figure 6.17: Simple: BCSSTK30



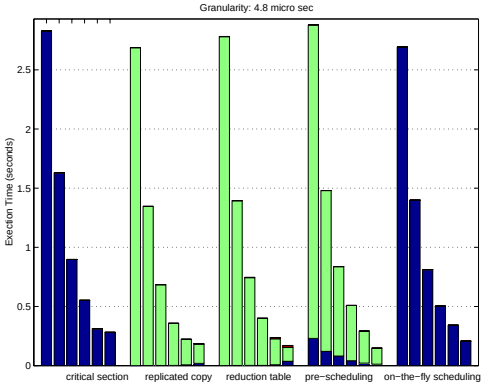

(a1)



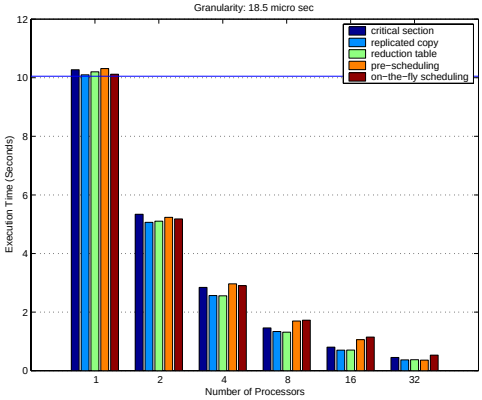

(b1)



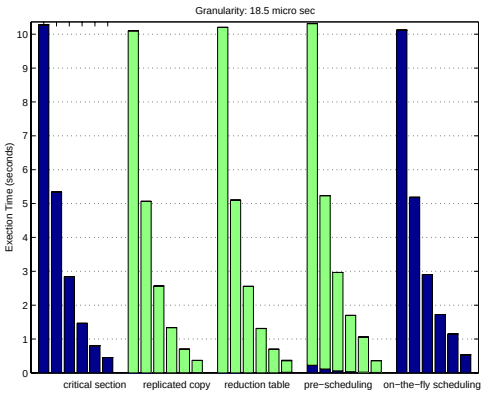

(a2)




(b2)

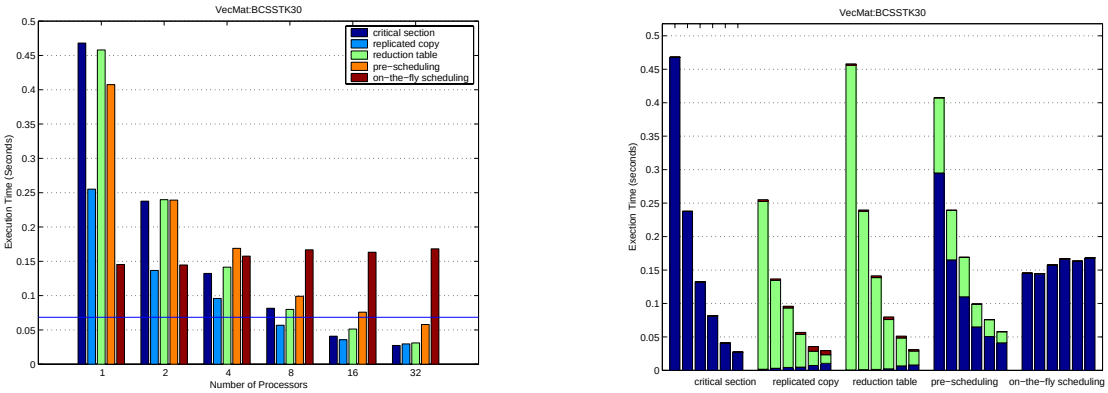



(a3)



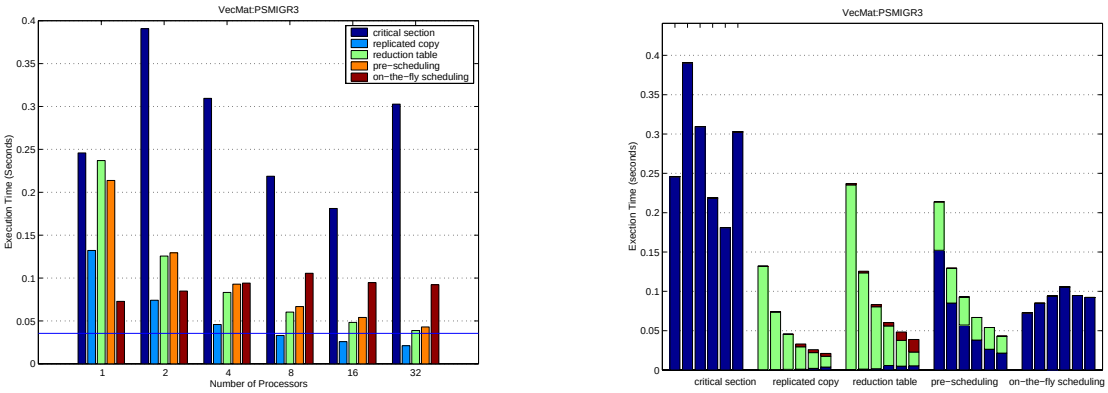

(b3)

Figure 6.18: Simple: PSMIGR3



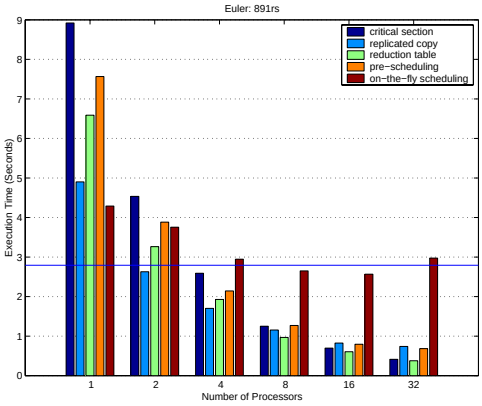

(a) (b)

Figure 6.19: VecMat: BCSSTK30

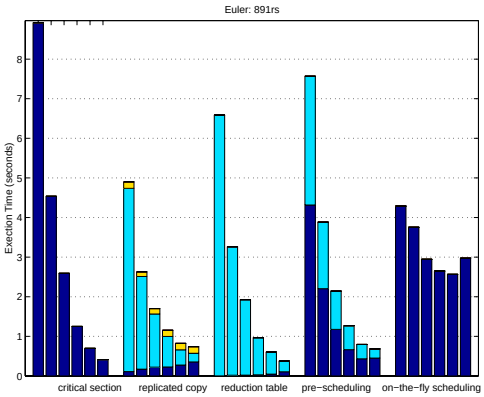



(a) (b)

Figure 6.20: VecMat: PSMIGR3

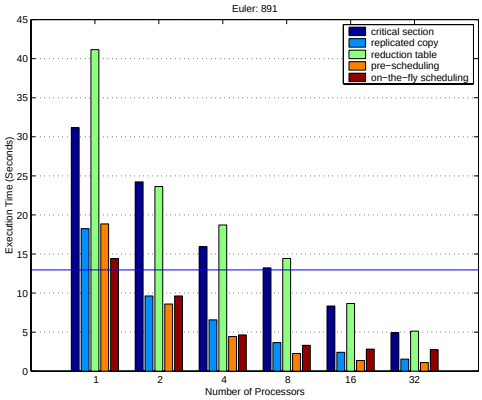



(a)

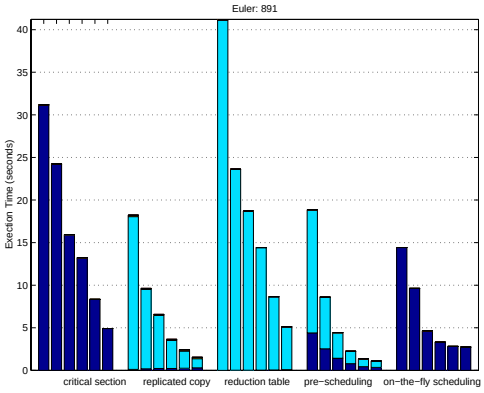



(b)

Figure 6.21: EULER: 891rs

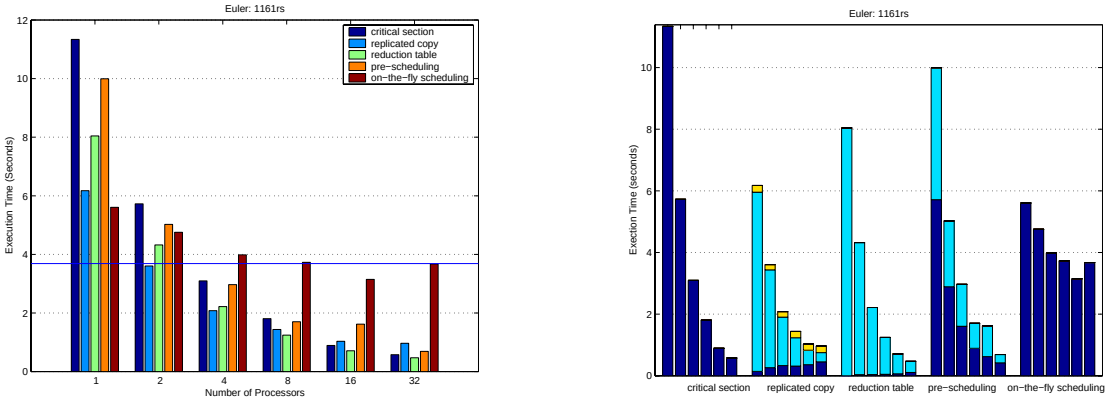



(a)



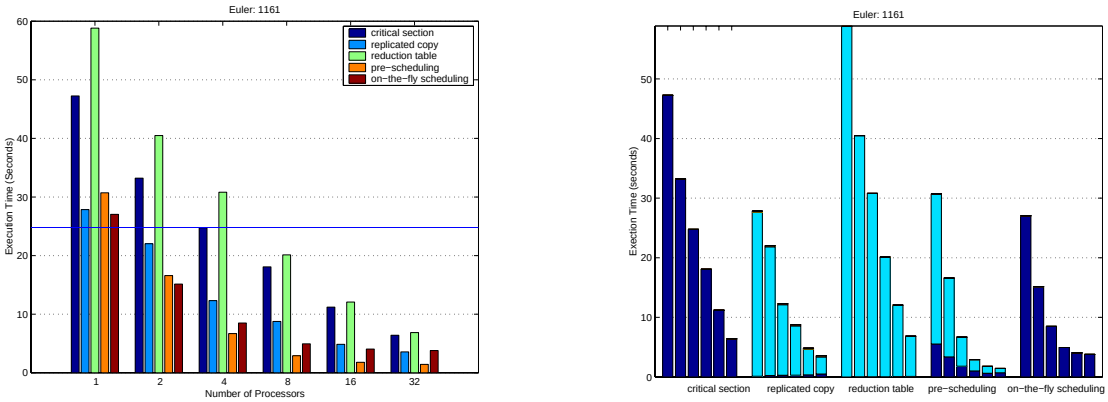

(b)

Figure 6.22: EULER: 891



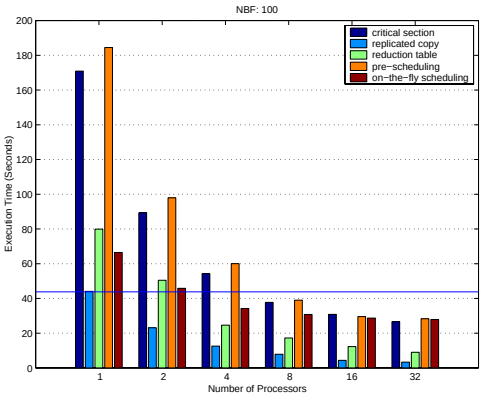

(a) (b)

Figure 6.23: EULER: 1161rs

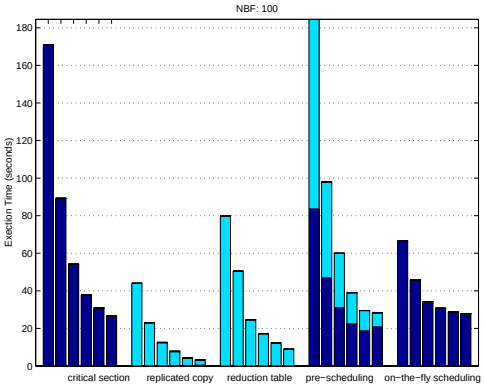



(a) (b)

Figure 6.24: EULER: 1161

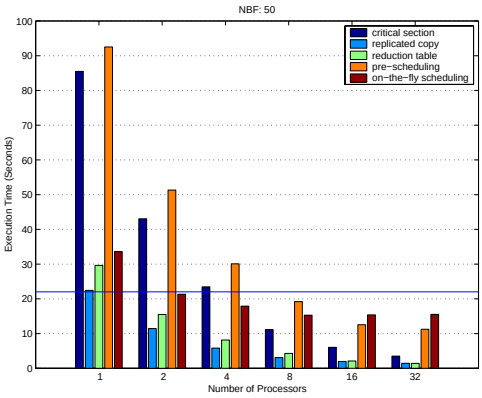



(a)

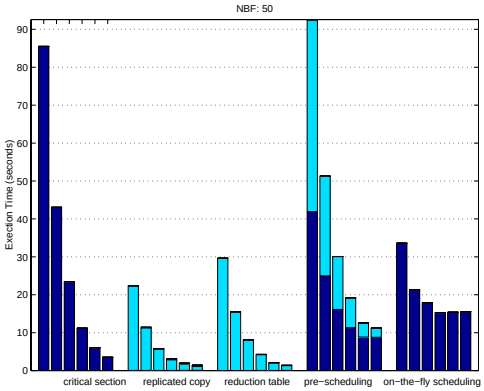



(b)

Figure 6.25: NBF: 100

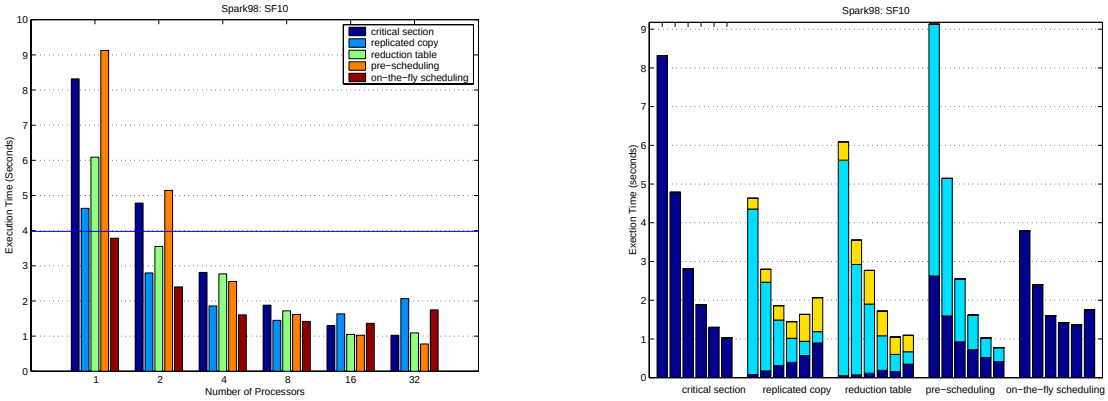



(a)



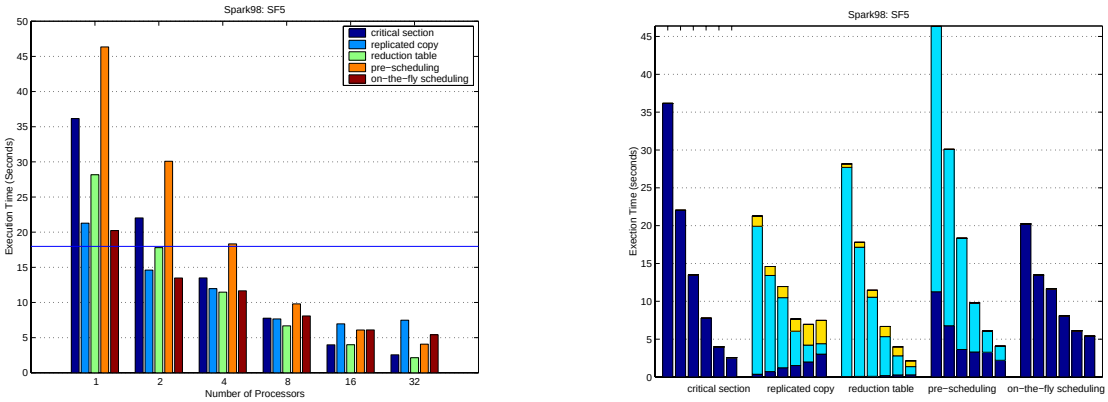

(b)

Figure 6.26: NBF: 50




(a) (b)

Figure 6.27: Spark98: SF10




(a) (b)

Figure 6.28: Spark98: SF5

## 6.4 Conclusion

Irregular reduction loops appear frequently in many sparse and irregular programs. An irregular reduction loop can be parallelized by using different methods. In this chapter, we described five different methods that can be applied automatically by a parallelizing compiler. We addressed the difficulties of using these methods, and compared the performance of the five methods in experiments with five different applications.

We found that there does not exist one single best parallel irregular reduction method. An ideal solution would be for the compiler to be capable of applying all the methods and choosing one according to the pattern of the loop, the characteristic of the input data, the resource restriction, and the performance requirements.

Our experiments did, however, reveal some heuristic guidelines for choosing a parallelization method.

The five methods can be put into two classes: the iteration domain decomposition methods and the data domain decomposition methods. In general, the iteration domain decomposition methods (i.e., critical section method, replicated copy method, and reduction table method) are versatile and easier for a compiler to apply automatically. The data domain decomposition methods (i.e., on-the-fly-scheduling method and pre-scheduling method) work for simple loops with simple loop bodies, but require the support of other compiler techniques, (e.g., loop distribution, loop coalescing, or program slicing) to handle more complicated cases, such as nested loops and loops with multiple irregular access patterns.

Among the three iteration domain decomposition methods, the critical section method is the most general one. It is a good choice when the reduction loop involves subroutine calls. The replicated copy performs well in most cases. Its overhead (in both memory requirement and execution time) becomes large when a large number of processors (thirty-two in our experiments) are used. When the access pattern has good locality, the critical section method and the reduction table method have good and scalable performance. A possible solution is to always use the reduction table method to parallelize the loop at compile-time, and dynamically set the size of the reduction table at run-time according to the characteristic of the input data. For example, when the size of the data domain is small, the size of the reduction table should be set to the size of the data domain

so that the reduction table method performs similarly to the replicated copy method. When the access pattern has good locality and the size of the data domain is large, the size of the reduction table should be set to the size of the data domain divided by the number of processor used in order to take advantage of the locality and avoid the overheads.

If memory restriction is the major concern, then the on-the-fly-scheduling method is the best since it requires no extra memory space. However, its performance is mediocre and it does not scale well. The replicated copy method should be avoided when the size of the data domain is large or if many processors are to be used because the amount of extra memory space required is proportional to the product of the size of the data domain and the number of processors used.

If the access pattern of the reduction array remains constant during several invocations of the reduction loop or several reduction loops share the same access pattern, and the reduction loop contains at most two different access patterns, then the pre-scheduling method is the best. The execution phase can have very good data locality and scales very well with the number of processors used.

# Chapter 7

# CONCLUSIONS AND FUTURE WORK

## 7.1 Conclusion

In this dissertation, we have discussed our efforts to develop compiler analysis techniques for sparse and irregular applications, with a focus on automatic parallelization.

We first tried to understand the difficulties in applying traditional compiler analysis techniques and parallelization techniques to sparse and irregular applications. By studying a collection of FORTRAN 77 programs with sparse and irregular data access patterns, we have identified several important issues that must be addressed. They are: new ways to detect privatizable arrays, analysis of irregular array accesses, overhead elimination of run-time tests, methods for efficient parallel irregular reductions, and parallelization of premature exit loops. For each of the problems, we also briefly discussed its compiler solutions. In the empirical study, we found that the techniques we proposed improved the effectiveness of parallelizing compilers.

Among the key issues identified, the two most important ones are the compiler analysis of irregular array accesses and the parallelization of irregular reduction loops. In the rest of this dissertation, we described, in detail, our efforts to solve these two problems.

We have developed compiler analysis techniques for the two most common irregular array accesses: single-indexed array accesses and simple indirect array accesses.

For single-indexed array accesses, we used a simple bounded depth-first search method to check how the value of the index variable changes between the array accesses. We can verify two kinds of common access patterns: consecutively-written access and array stack access. We also described

how to use this pattern information to enhance compiler optimizations, such as array privatization, elimination of array bounds checking, and data dependence tests.

For simple indirect array access, we developed an array property analysis technique. This technique can be used to find index array property information which can, in turn, be used in the analysis of its host array. In order to make this method efficient and useful for real programs, we have designed it to be interprocedural and demand-driven.

We have implemented these two techniques in Polaris, a parallelizing compiler developed at the University of Illinois at Urbana-Champaign. We also extended the original data dependence tests and the array privatization tests implemented in Polaris so that they could use our techniques on demand. Our experiments with five programs have demonstrated that a parallelizing compiler enhanced with these techniques can detect more inherently parallel loops than traditional compilers, especially in sparse and irregular applications.

We have studied five different possible parallelization methods for irregular reduction loops, all of which can be applied automatically by a compiler. We compared their ease of use, applicability, supporting compiler techniques required, run-time resource requirements and, most importantly, run-time performance. Not surprisingly, there does not exist one single best method. Our analysis and experiments revealed a general guideline of choosing a parallel irregular reduction method that satisfies given requirements.

## 7.2 Future Work

Our work revealed that there is much room for improvement.

### 7.2.1 Further Study of Irregular Applications

The importance of the key techniques we identified is based on the empirical study we did as the first step of a program collection. The effectiveness of the techniques we have developed and implemented are also demonstrated through experiments with five programs from the same program collection. Although we tried to use programs from different disciplines, they are by no means comprehensive. It would be desirable to extend our study to other application codes.

A particularly interesting direction is to study C/C++ programs. In our work, we studied

FORTRAN programs in which the basic data type was the array. Our techniques work well for FORTRAN codes because they take advantage of the common program patterns people follow when they use arrays to represent complicated and irregular data structures. In C/C++ programs, we face a different, yet well-known problem: pointer analyses. There already has been much research done in this area. We believe it would be beneficial to follow the same philosophy we used in our work: study real programs and find the common program patterns.

### 7.2.2 Integrated Compile-time and Run-time Optimization

In Section 4.5, we briefly described a mechanism to generate test codes at compile-time and to use run-time tests to verify array properties. In Section 6.4, we also mentioned using run-time feedback to dynamically adjust the size of reduction tables in irregular reduction loops. When an optimization that is input dependent needs to be performed, a run-time method is required.

Using run-time methods does not mean less work for compilers. On the contrary, to use run-time methods effectively and efficiently, one needs strong compiler support. For example, in Section 2.6, we described several compile-time techniques to reduce the overhead of run-time tests. The technique to hoist loop invariant test codes is only meaningful in the context of the run-time test. Optimizing compiler-generated run-time test codes pose new challenges to compiler techniques.

Currently, we know very little about how to integrate compile-time optimizations and run-time optimizations. It should be a very interesting research topic. Just as compiler optimization provides a higher level of abstraction to the programmers than direct code generation does, run-time optimization will provide an even higher level of abstraction and will further liberate programmers.

# REFERENCES


[1] Harold Abelson, Gerald Jay Sussman, and Julie Sussman. *Structure and Interpretation of Computer Programs*. The MIT Press, San Francisco, California, 1985.

[2] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley Publishing Company, 1986.

[3] John R. Allen and Ken Kennedy. Automatic loop interchange. In *Proceedings of the SIGPLAN '84 Symposium on Compiler Construction*, pages 233–246, New York, NY 10036, USA, 1984. ACM Press, ACM Press.

[4] Zahira Ammarguellat and W. L. Harrison, III. Automatic recognition of induction variables and recurrence relations by abstract interpretation. In Mark Scott Johnson, editor, *Proceedings of the ACM SIGPLAN '90 Conference on Programming Language Design and Implementation (SIGPLAN '90)*, pages 283–295, White Plains, NY, USA, June 1990. ACM Press.

[5] R. Asenjo, E. Gutierrez, Y. Lin, D. Padua, B. Pottenger, and E. Zapata. On the automatic parallelization of sparse and irregular fortran codes. Technical Report CSRD-TR-1512, Dept. of Computer Science, University of Illinois at Urbana Champaign, December 1996.

[6] V. Balasundaram and K. Kennedy. A technique for summarizing data access and its use in parallelism-enhancing transformations. In *Proceedings of the ACM SIGPLAN'89 Conference on Programming Language Design and Implementation (PLDI)*, Portland, OR, June 1989.

[7] U. Banerjee. Data dependence in ordinary programs. Master's thesis, Dept. of Computer Science, University of Illinois at Urbana-Champaign, November 1976. Report No. 76-837.

[8] Utpal Banerjee. *Dependence Analysis*. Kluwer Academic Publishers, 1997.

[9] J. Barnes and P. Hut. A hierarchical o(nlogn) force calculation algorithm. *Nature*, 324(4):446–449, 1986.

[10] Joshua E. Barnes. ftp://hubble.ifa.hawaii.edu/pub/barnes/treecode/. Technical report, Institute for Astronomy, University of Hawaii, 1994.

[11] A. J. Bernstein. Analysis of programs for parallel processing. *IEEE Transactions on Electronic Computers*, 15:757–762, October 1966.

[12] Aart J. C. Bik and Harry A. G. Wijshoff. Automatic data structure selection and transformation for sparse matrix computations. *IEEE Transactions on Parallel and Distributed Systems*, 7(2):109–126, February 1996.

[13] W. Blume and R. Eigenmann. An overview of symbolic analysis techniques needed for the effective parallelization of the perfect benchmarks. In K. C. Tai, editor, *Proceedings of the 23rd International Conference on Parallel Processing. Volume 2: Software*, pages 233–238, Boca Raton, FL, USA, August 1994. CRC Press.

[14] William Blume, Ramon Doallo, Rudolf Eigenmann, John Grout, Jay Hoeflinger, Thomas Lawrence, Jaejin Lee, David Padua, Yunheung Paek, Bill Pottenger, Lawrence Rauchwerger, and Peng Tu. Parallel programming with polaris. *IEEE Computer*, 29(12):78–82, December 1996.

[15] William Blume and Rudolf Eigenmann. The range test: A dependence test for symbolic, non-linear expressions. In *Proceedings of the Conference on Supercomputing*, pages 528–537, Los Alamitos, November 1994. IEEE Computer Society Press.

[16] William Blume, Rudolf Eigenmann, Keith Faigin, John Grout, Jay Hoeflinger, David Padua, Paul Petersen, William Pottenger, Lawrence Rauchwerger, Peng Tu, and Stephen Weatherford. Polaris: Improving the effectiveness of parallelizing compilers. In Keshav Pingali, Uptal Banerjee, David Gelernter, Alex Nicolau, and David Padua, editors, *Proceedings of the 7th International Workshop on Languages and Compilers for Parallel Computing*, Lecture Notes in Computer Science, pages 141–154, Ithaca, New York, August 8–10, 1994. Springer-Verlag.

[17] William Joseph Blume. *Symbolic analysis techniques for effective automatic parallelization.* PhD thesis, University of Illinois at Urbana-Champaign, June 1995.

[18] Preston Briggs and Keith D. Cooper. Effective partial redundancy elimination. In *Proceedings of the ACM SIGPLAN '94 Conference on Programming Language Design and Implementation*, pages 159–170, 1994.

[19] D. Callahan and K. Kennedy. Analysis of interprocedural side effects in a parallel programming environment. *Journal of Parallel and Distributed Computing*, 5(5):517–550, October 1988.

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

[21] Cliff Click. Global code motion/global value numbering. In *Proceedings of the ACM SIGPLAN'95 Conference on Programming Language Design and Implementation (PLDI)*, pages 246–257, La Jolla, California, 18–21 June 1995.

[22] Béatrice Creusillet and Francois Irigoin. Interprocedural array region analysis. In *Eighth International Workshop on Languages and Compilers for Parallel Computing (LCPC'95)*, pages 4–1 to 4–15. Ohio State University, Colombus (Ohio), August 1995.

[23] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark K. Wegman, and F. Kenneth Zadeck. An efficient method of computing static single assignment form. In *Proceedings of 16th Annual ACM Symposium on Principles of Programming Languages*, pages 25–35, 1989.

[24] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transaction on Programming Languages and Systems*, 13(4):451–490, October 1991.

[25] Pedro C. Diniz and Martin C. Rinard. Dynamic feedback: An effective technique for adaptive computing. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI-97)*, volume 32, 5 of *ACM SIGPLAN Notices*, pages 71–84, New York, June 15–18 1997. ACM Press.

[26] Evelyn Duesterwald, Rajiv Gupta, and Mary Lou Soffa. A practical framework for demand-driven interprocedural data flow analysis. *ACM Transactions on Programming Languages and Systems*, 19(6):992–1030, November 1997.

[27] Mike Berry et.al. The perfect club benchmarks: Effective performance evaluation of supecomputers. *International Journal of Supercomputer Applications*, 3(3):5–40, Fall 1989.

[28] P. Feautrier. Array expansion. In *Proceedings of the Second International Conference on Supercomputing*, St. Malo, France, July 1988.

[29] P. Feautrier. Dataflow analysis of scalar and array references. *International Journal of Parallel Programming*, 20(1):23–52, February 1991.

[30] A. Ferreira and J. D. P. Rolim. *Parallel Algorithms for Irregular Problems: State of the Art.* Kluwer Academic Publishers, 1995.

[31] Ian Foster, Rob Schreiber, and Paul Havlak. Hpf-2 scope of activities and motivating applications. Technical Report CRPC-TR94492, Rice University, November 1994.

[32] Michael P. Gerlek, Eric Stoltz, and Michael Wolfe. Beyond induction variables: Detecting and classifying sequences using a demand-driven SSA form. *ACM Transactions on Programming Languages and Systems*, 17(1):85–122, January 1995.

[33] Thomas Gross and Peter Steenkiste. Structured dataflow analysis for arrays and its use in an optimizing compiler. *Software Practice and Experience*, 20(2):133–155, February 1990.

[34] Junjie Gu, Zhiyuan Li, and Gyungho Lee. Symbolic array dataflow analysis for array privatization and program parallelization. In Sidney Karin, editor, *Proceedings of the 1995 ACM/IEEE Supercomputing Conference, December 3–8, 1995, San Diego Convention Center, San Diego, CA, USA*, New York, NY 10036, USA, 1995. ACM Press and IEEE Computer Society Press.

[35] Rajiv Gupta. Optimizing array bound checks using flow analysis. *ACM Letters on Programming Languages and Systems*, 2(1–4):135–150, March–December 1993.

[36] E. Gutierrez, O. Plata, and E.L. Zapata. On automatic parallelization of irregular reductions on scalable shared memory systems. In *Proceedings of the 5th Int'l. Euro-Par Conference*, August 1999.

[37] M. W. Hall, J. M. Anderson, S. P. Amarasinghe, B. R. Murphy, S. W. Liao, E. Bugnion, and M. S. Lam. Maximizing multiprocessor performance with the suif compiler. *IEEE Computer*, 29(12):84–89, December 1996.

[38] M. W. Hall, B. R. Murphy, and S. P. Amarasinghe. Interprocedural analysis for parallelization: Design and experience. In *SIAM Conference on Parallel Processing for Scientific Computing*, February 1995.

[39] Mary W. Hall, Saman P. Amarasinghe, Brian R. Murphy, Shih-Wei Liao, and Monica S. Lam. Detecting coarse-grain parallelism using an interprocedural parallelizing compiler. In Sidney Karin, editor, *Proceedings of the 1995 ACM/IEEE Supercomputing Conference, December 3–8, 1995, San Diego Convention Center, San Diego, CA, USA*, pages ??–??, New York, NY 10036, USA and 1109 Spring Street, Suite 300, Silver Spring, MD 20910, USA, 1995. ACM Press and IEEE Computer Society Press.

[40] H. Han and C.W. Tseng. improving compiler and run-time support for adaptive irregular codes. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT'98)*, Paris, France, October 1998.

[41] P. Havlak. *Interprocedural Symbolic analysis*. PhD thesis, Rice University, May 1994.

[42] Paul Havlak and Ken Kennedy. Experience with interprocedural analysis of array side effects. *IEEE Transactions on Parallel and Distributed Systems*, 2(3), July 1991.

[43] Michael T. Heath, Esmond Ng, and Barry W. Peyton. Parallel algorithms for sparse linear systems. *SIAM Review*, 33(3):420–460, September 1991.

[44] K. Kennedy, K. S. McKinley, and C.-W. Tseng. Analysis and transformation in the ParaScope Editor. In *Proceedings of the 1991 ACM International Conference on Supercomputing*, Cologne, Germany, June 1991.

[45] Priyadarshan Kolte and Michael Wolfe. Elimination of redundant array subscript range checks. *ACM SIGPLAN Notices PLDI 1995*, 30(6):270–278, June 1995.

[46] Vladimir Kotlyar, Keshav Pingali, and Paul Vinson Stodghill. Compiling parallel code for sparse matrix applications. In *SC'97: High Performance Networking and Computing: Proceedings of the 1997 ACM/IEEE SC97 Conference: November 15–21, 1997, San Jose, California, USA.*, New York, NY 10036, USA and 1109 Spring Street, Suite 300, Silver Spring, MD 20910, USA, 1997. ACM Press and IEEE Computer Society Press.

[47] Z. Li. Array privatization for parallel execution of loops. In *Proceedings of 1992 International Conference on Supercomputing, July 19–23, 1992, Washington, DC*, pages 313–322, New York, NY 10036, USA, 1992. ACM Press.

[48] Zhiyuan Li and Pen-Chung Yew. Interprocedural analysis for parallel computing. In *1988 International Conference on Parallel Processing*, volume II, pages 221–228, St. Charles, Ill., August 1988.

[49] Y. Lin and D. Padua. On the automatic parallelization of sparse and irregular fortran programs. In *Proc. of 4th Workshop on Languages, Compilers, and Run-time Systems for Scalable Computers (LCR98)*, volume 1511 of *Lecture Notes in Computer Science*, pages 41–56. Springer-Verlag, Pittsburgh, PA, 1998.

[50] Y. Lin and D. Padua. Demand-driven interprocedural array property analysis. In *Proceedings of the 12th International Workshop on Languages and Compilers for Parallel Computing*, San Diego, CA, August 1999.

[51] Joseph W. H. Liu. The role of elimination trees in sparse factorization. *SIAM Journal on Matrix Analysis and Applications*, 11(1):134–172, January 1990.

[52] Victoria Markstein, John Cocke, and Peter Markstein. Optimization of range checking. In *Proceedings of the SIGPLAN '82 Symposium on Compiler Construction*, pages 114–119. ACM, ACM, 1982.

[53] Vadim Maslov. Lazy array data-flow dependence analysis. In *Proceedings of 21st Annual*

*ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 311–325, New York, NY, USA, May 1994. ACM Press.

[54] Dror E. Maydan, Saman P. Amarasinghe, and Monica S. Lam. Array-data flow analysis and its use in array privatization. In *Proceedings of the Twentieth Annual ACM Symposium on Principles of Programming Languages, Charleston, South Carolina, January 10–13, 1993*, pages 2–15, New York, NY, USA, 1993. ACM Press.

[55] Kathryn S. McKinley. Dependence analysis of arrays subscripted by index arrays. Technical Report TR91-162, Dept. of Computer Science, Rice University, June 1991.

[56] S. S. Mukherjee, S. D. Sharma, M. D. Hill, J. R. Larus, A. Rogers, and J. Saltz. Efficient support for irregular applications on distributed-memory machines. In *Proc. of the Fifth ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming (PPOPP'95)*, pages 68–79, July 1995.

[57] David R. O'Hallaron. Spark98: sparse matrix kernels for shared memory and message passing systems. Technical Report CMU-CS-97-178, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA 15213, October 1997.

[58] Yunheung Paek, Jay Hoeflinger, and David Padua. Simplification of array access patterns for compiler optimizations. In *Proceedings of the ACM SIGPLAN'98 Conference on Programming Language Design and Implementation (PLDI)*, pages 60–71, Montreal, Canada, 17–19 June 1998.

[59] Paul M. Petersen and David A. Padua. Static and dynamic evaluation of data dependence analysis techniques:. *IEEE Transactions on Parallel and Distributed Systems*, 7(11):1121–1132, November 1996.

[60] Sergio Pissanetzky. *Sparse Matrix Technology.* Academic Press, Orlando, Florida, 1984.

[61] Lynn Pointer. Perfect: performance evaluation for cost-effective transformations, report 2. Technical Report CSRD 964, University of Illinois at Urbana-Champaign, Center for Supercomputing Research and Development, Urbana, IL 61801, USA, March 1990.

[62] William Pugh. A practical algorithm for exact array dependence analysis. *Communications of the ACM*, 35(8):102–114, August 1992.

[63] William Pugh and David Wonnacott. Nonlinear array dependence analysis. Technical Report CS-TR-3372, Dept. of Computer Science, Univ. of Maryland, November 1994.

[64] William Pugh and David Wonnacott. An exact method for analysis of value-based array data dependences. In *Proceedings of the Sixth Annual Workshop on programming Languages and Compilers for Parallel Computing*, December 93.

[65] Lawrence Rauchwerger. *Run-time parallelization: a framework for parallel computation.* PhD thesis, University of Illinois at Urbana-Champaign, 1995.

[66] Y. Saad. SPARSKIT: A basic tool kit for sparse matrix computations. Technical Report 90-20, Research Institute for Advanced Computer Science, NASA Ames Research Center, Moffett Field, CA, 1990.

[67] S. D. Sharma, R. Ponnusamy, B. Moon, Y.-S. Hwang, R. Das, and J. Saltz. Run-time and compile-time support for adaptive irregular problems. In IEEE, editor, *Proceedings, Supercomputing '94: Washington, DC, November 14–18, 1994*, Supercomputing, pages 97–106, 1109 Spring Street, Suite 300, Silver Spring, MD 20910, USA, 1994. IEEE Computer Society Press.

[68] Z. Shen, Z. Li, and P. C. Yew. An empirical study on array subscripts and data dependencies. In *International Conference on Parallel Processing, Vol. 2: Software*, pages 145–152, Pennsylvania, USA, August 1989. The Pennsylvania State University.

[69] Loren Taylor Simpson. Value-driven redundancy elimination. Technical Report TR98-308, Rice University, April 6, 1998.

[70] Madalene Spezialetti and Rajiv Gupta. Loop monotonic statements. *IEEE Transactions on Software Engineering*, 21(6):497–505, June 1995.

[71] F. Tip. A survey of program slicing techniques. *Journal of programming languages*, 3:121–189, 1995. Surveys the state-of-the-art in program slicing and gives many references to the literature.

[72] R. Triolet, F. Irigoin, and P. Feautrier. Direct parallelization of call statements. In *Proceedings of the SIGPLAN'86 Symposium on Compiler Construction*, pages 176–185, Palo Alto, CA, July 1986.

[73] P. Tu and D. Padua. Gated SSA-Based demand-driven symbolic analysis for parallelizing compilers. In *Conference proceedings of the 1995 International Conference on Supercomputing, Barcelona, Spain, July 3–7, 1995*, CONFERENCE PROCEEDINGS OF THE INTERNATIONAL CONFERENCE ON SUPERCOMPUTING 1995; 9th, pages 414–423, New York, NY 10036, USA, 1995. ACM Press.

[74] Peng Tu and David Padua. Automatic array privatization. In Uptal Banerjee, David Gelernter, Alex Nicolau, and David Padua, editors, *Proceedings of the 6th International Workshop on Languages and Compilers for Parallel Computing*, Lecture Notes in Computer Science, pages 500–521, Portland, Oregon, August 12–14, 1993. Intel Corp. and the Portland Group, Inc., Springer-Verlag.

[75] M. Ujaldon, E. L. Zapata, B. M. Chapman, and H. P. Zima. Vienna-Fortran/HPF extensions for sparse and irregular problems and their compilation. *IEEE Transactions on Parallel and Distributed Systems*, 8(10):1068–1083, October 1997.

[76] Manuel Ujaldon, Emilio L. Zapata, Shamik D. Sharma, and Joel Saltz. Parallelization techniques for sparse matrix applications. *Journal of Parallel and Distributed Computing*, 38(2):256–266, November 1996.

[77] R. v. Hanxleden. Handling irregular problems with Fortran D — A preliminary report. In *Proceedings of the Fourth Workshop on Compilers for Parallel Computers*, Delft, The Netherlands, December 1993.

[78] R. von Hanxleden, K. Kennedy, C. Koelbel, R. Das, and J. Saltz. Compiler analysis for irregular problems in Fortran D. In Uptal Banerjee, David Gelernter, Alex Nicolau, and David Padua, editors, *Proceedings of the 5th International Workshop on Languages and Compilers for Parallel Computing*, Lecture Notes in Computer Science, pages 97–111, New Haven, Connecticut, August 3–5, 1992. Springer-Verlag.

[79] M. Weiser. Program slicing. *IEEE Transactions on Software Engineering*, SE-10(4):352–357, July 1984.

[80] M. J. Wolfe. *Optimizing Supercompilers for Supercomputers*. PhD thesis, Dept. of Computer Science, University of Illinois at Urbana-Champaign, October 1982.

[81] M. J. Wolfe. *High Performance Compilers for Parallel Computing*. Addison-Wesley, Redwood City, CA, 1996.

[82] Michael Wolfe. Beyond induction variables. In *Proceedings of the Conference on Programming Language Design and Implementation (PLDI)*, volume 27, pages 162–174, New York, NY, July 1992. ACM Press.

[83] Mihalis Yannakakis. Computing the minimum fill-in is NP-complete. *SIAM Journal on Algebraic and Discrete Methods*, 2(1):77–79, March 1981.

[84] B. Chapman H. Zima. *Supercompilers for Parallel and vector Computers*. ACM Frontier Series. Addison-Wesley, 1991.

# VITA

Yuan Lin was born on October 23, 1968 in Shanghai, China. In 1987, he entered Fudan University, and earned a Bachelor in Science degree in Mathematics in 1991 and a Master in Science degree in Computer Science in 1994. After working as a technical consultant in the Professional Service Organization in Hewlett-Packard China for two years, he started his graduate studies at the University of Illinois at Urbana-Champaign in August, 1996. Since then, he had been a graduate research assistant at the Computer Science Department, working under the direction of Professor David Padua. He earned his Doctor of Philosophy in Computer Science in 2000. After graduation, he will join the Star*Core design center (Motorola) in Atlanta, Georgia.