Algorithms for automatic parallelism, optimization, and optimal processor utilization

Katharine J Macke
University of Nevada, Las Vegas

Follow this and additional works at: https://digitalscholarship.unlv.edu/rtds

Repository Citation
https://digitalscholarship.unlv.edu/rtds/315

This Thesis is brought to you for free and open access by Digital Scholarship@UNLV. It has been accepted for inclusion in UNLV Retrospective Theses & Dissertations by an authorized administrator of Digital Scholarship@UNLV. For more information, please contact digitalscholarship@unlv.edu.
INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

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

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

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps. Each original is also photographed in one exposure and is included in reduced form at the back of the book.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.
Algorithms for automatic parallelism, optimization, and optimal processor utilization

Macke, Katharine J., M.S.
University of Nevada, Las Vegas, 1993
ALGORITHMS FOR AUTOMATIC PARALLELISM, OPTIMIZATION, AND OPTIMAL PROCESSOR UTILIZATION

by

Katharine J. Macke

A thesis submitted in partial fulfillment of the requirements for the degree of

Master of Science

in

Computer Science

Department of Computer Science
University of Nevada, Las Vegas
May, 1993
The thesis of Katharine J. Macke for the degree of Master of Science in Computer Science is approved.

Chairperson, E. A. Yfantis Ph.D.

Examining Committee Member, Kia Makki Ph.D.

Examining Committee Member, Ajoy Datta Ph.D.

Graduate Faculty Representative, Darrell Pepper Ph.D.

Graduate Dean, Ronald Smith Ph.D.

University of Nevada, Las Vegas
May, 1993
Abstract

In this thesis we first investigate the reaching definitions optimization. This compiler optimization collects and stores information about where a variable is defined and how long that definition of the variable stays alive before it is redefined. We compare the old iterative solution to a new algorithm that uses the partialout concept. The partialout solution decreases execution time by eliminating the multiple passes required in the iterative solution. Currently, compilers that find a data dependence between two statements in a loop do not parallelize the loop. Next we investigate automatic parallelism for these loops by breaking the loop into a set of smaller loops, each of which contains no dependencies and thus can be executed in parallel. Finally, we introduce a set of algorithms for optimal processor utilization. The algorithms split the loop into a sequential series of parallel blocks, each block executing in parallel and utilizing the optimal number of processors possible.
Contents

ABSTRACT iii

ACKNOWLEDGEMENTS viii

CHAPTER 1 INTRODUCTION 1

CHAPTER 2 REACHING DEFINITIONS 5

2.1 Analysis ................................................................. 5
2.2 Loops ................................................................. 6
2.3 Algorithm ............................................................. 7
2.4 Example .............................................................. 8
  2.4.1 Program ......................................................... 9
  2.4.2 Flow Chart ...................................................... 10
  2.4.3 Loop Structures .............................................. 10
  2.4.4 In-Out Calculations ....................................... 11
  2.4.5 Analysis ....................................................... 12

CHAPTER 3 DATA DEPENDENCE 14
Acknowledgments

I wish to thank the chairman of my committee, Dr. E. A. Yfantis for his wonderful support, ingenious ideas, and academic friendship. I also wish to thank Dr. Kia Makki who was among the very best of my teaching professors and a very kind human being. A special thanks also to Dr. Roy Ogawa who got me started at this fine institution and kept by my side until the very end. Without the help, support, and professionalism of these three gentleman, the course work and this thesis would not have been completed. I wish also to thank the rest of my committee, Dr. A. Datta and Dr. D. Pepper for their time and effort. And finally I want to thank my family and friends who remained ever so patient and totally supportive of these endeavors. Thank you Mom, Dad, Bob, and Sami.
Chapter 1

Introduction

In this research project we analyze and design compiler algorithms for automatic parallelism, optimization, and optimal processor utilization. As the speed of the hardware is increasing, more research is needed in designing software applications for compilers that will take advantage of the emerging hardware designs. In chapter 2 we investigate a current compiler optimization. Compiler optimizations use two types of analysis. The first is control-flow analysis which helps identify the loops in the flow graph of a program. The other is data-flow analysis which is a process of collecting information about the way variables are used in a program [1]. One type of data-flow analysis that is often employed in optimizing compilers is the reaching definitions analysis. In this type of analysis, information about all the definitions of a program are collected and stored in a use-definition or ud chain. The compiler collects and stores information for where each variable is defined and how far that definition of the variable reaches [6]. Reaching definitions anal-
ysis is a valuable tool in that the ud chains are used for several optimizations which include copy propagation, detection of loop-invariant computations, code motion, and detection of induction variables [1]. We present a new algorithm for defining the reaching definitions that will further enhance the speed of execution.

With the advent of massively parallel machines such as the Thinking Machine, compilers are needed that can execute existing sequential code in a parallel manner. In Chapter 3 we introduce the concept of data dependence between two statements in a loop which uses array indexing. If a dependence exists, then there is a possibility that array elements are accessing the same memory location within the area of execution [14]. Currently, if a compiler finds a data dependence between two statements in a loop, no parallelization takes place [12]. Therefore, we explore the possibility of splitting the loop into many smaller loops so that code within each smaller loop or parallel block has no dependencies and thus can be parallelized.

The automatic parallelization of code should optimally balance processor utilization, thus reducing the number of time units needed to execute the code. In Chapter 4 we implement our loop splitting concept of Chapter 3 and present a group of algorithms that can be used to optimally parallelize loops which use array-indexing. Loops have a sequential nature in that we think of a loop as a series of statements to be executed and then repeated n times. But parallel compiler writers know that if there is no dependence
between the statements of a loop body, the statements may be executed in parallel [12]. To parallelize loops which use array-indexing, data dependence tests are used to determine types of dependencies between any two statements $S$ and $S'$ in the loop. Popular data dependence tests now in use include the GCD Test, the Separability Test, the Banerjee Test, and the Omega Test [12, 16, 4, 10]. In this research we make use of the Separability Test and assume this test returns true for statements $S$ and $S'$ of a given loop that uses array-indexing. Given this assumption, we present a method of parallelization to optimally balance processor utilization given the dependencies returned by the Separability Test. Making use of the diophantine equation resulting from the dependencies of statements $S$ and $S'$, we present loop-splitting algorithms for optimal parallelization. Each loop-splitting algorithm splits the loop into smaller loops each of which has no statement dependencies and thus can be executed in parallel. The diophantine equation resulting from a dependence between $S$ and $S'$ may be expressed as $ax + a_0 = by + b_0$ [12]. The loop-splitting algorithms use the coefficients $a$ and $b$ and the constant $c$ which equals $b_0 - a_0$. The equation is rewritten as $ax = by + c$. For the parameters $a = b$ and $c = 0$ we have a trivial case that has no cross dependencies and thus each available processor is assigned the execution of one loop iteration. The next two algorithms devise a loop-splitting schema that operates on the coefficients $a = b$ and $c \neq 0$. In these algorithms, the value of $c$ determines where the loop can be optimally split. If $c = \pm 1$ we find the worst-case for
optimization with each iteration of the loop dependent on the next iteration. Another algorithm investigates the case for $a > 0$ and $b < 0$ or $a < 0$ and $b > 0$. In this situation the index conflicts are embedded from outer to inner dependence and the algorithm seeks the innermost dependence with the shortest distance and splits the loop there. The last set of algorithms deal with the coefficients $a \neq b$. Given these parameters, the algorithm uses both $a$ and $b$ and the constant $c$ to determine where the loop can be optimally split. The Separability Test returns parametric equations that determine the index conflicts or collisions of the array. We find the first collision from these parametric equations and then execute a set of comparisons to determine whether an $x$index conflict (if $b > a$) or a $y$index conflict (if $a > b$) falls in the same parallel block. If the collision does fall within the same parallel block, we split the loop thus eliminating the dependence. The algorithms presented succeed in utilizing the maximum number of processors possible given the existing dependencies and thus reduce the number of time units needed for execution.
Chapter 2

Reaching Definitions

2.1 Analysis

Currently there are two different methods for collecting the reaching definitions information. One method uses the iterative solution and the other method uses what is known as interval analysis [1]. In the iterative solution, continuous passes are made over the blocks in sequential order until all the information is collected. With interval analysis, we start with a flow graph that represents the entire program and apply transformations that reduce the flow-graph to a single node or region. Both methods use data-flow equations to compute the IN and OUT of each block. The IN represents those definitions that are alive coming into the block and the OUT represents those definitions that are alive coming out of the block [11]. The IN and OUT for each block is derived from the following equations:

\[ \text{IN}(\text{block}) = \text{union} \ \text{OUT}(P) \text{ where } P \text{ is predecessor of block } P \]
\[\text{OUT}(\text{block}) = \text{gen}(\text{block}) \cup (\text{IN}(\text{block}) - \text{kill}(\text{block}))\]

The \textit{gen} is all definitions generated by the block and not killed by that same block and the \textit{kill} is all the definitions that are killed by the block itself. The main difficulty with the iterative solution is that multiple passes are required to find the final \text{IN} and \text{OUT} of each block. This results in excess work for the compiler often causing unessential reiteration of calculations of most blocks. The major fault with interval analysis is the added complexity of the algorithm when applying the data-flow equations. It has not been firmly established whether this added complexity results in a time savings but the method has been used in some optimizing compilers [1].

2.2 Loops

A close analysis of the reaching definitions problem will show that the major difficulty arises when the programmer uses a loop structure. Loop structures cause repeated passes in the iterative solution. This is because the calculations of the \text{IN} and \text{OUT} of the loop blocks must include definitions that are generated but not killed in the loop itself. Doing a sequential first pass over the nodes in the loop will not include those definitions, thus forcing another pass. It has been shown that the upper limit for the number of iterations needed is equal to the depth of nested loops plus two [1]. In other words, if a program has a loop nested inside another loop, the upper bound for the number of iterations needed is four. The final pass is redundant in that none of
the IN and OUT definitions change. This is simply a final check to ascertain that there are no more changes. It is our contention that an algorithm which calculates a partialout for loop blocks will eliminate the multiple passes of the iterative solution and generate significantly fewer bit manipulations.

2.3 Algorithm

The algorithm presented here looks at all loops of a program and calculates what we call a partialout for each loop. When loops are nested, the partialout of the innermost loop is always calculated first. Also for nested loops, once a block or node is visited, that block is removed from all outermost loops. This way, blocks are not revisited and calculations repeated. Once the partialouts for all the loops are calculated and stored, a single sequential pass for calculating the true IN and OUT is all that is needed, thus eliminating a significant number of bit manipulations. The data structures used are generally all simple bit strings. The bit manipulations include union, complement, and intersection. The overhead or added complexity of this algorithm is minimal. A bit string that represents all the nodes of a loop must be built. Since other optimizations, including detection of loop-invariant computations, code motion, and detection of induction variables also need this information, the loop bit string is not really overhead for this algorithm. But the loop bit strings do have to be sorted from innermost to outermost which adds some complexity.
/ * dopass * /
for each block in sequential order do
  IN(block) = union pout(P) where P is predecessor
  OUT(block) = gen(block) union (pin(block) - kill(block))
/* main */
for each loop innermost to outermost do
  pout(headerblock) = empty
  nextblock
  for each block not visited in loop do
     pin(block) = union pout(P) where P is predecessor
     pout(block) = gen(block) union (pin(block) - kill(block))
  for all blocks except headerblock in loop
     mark visited in next outer loop
  dopass

A careful analysis of this algorithm will show that it forces the calculation of the partialout of the smallest or innermost region, extends out to the next larger region and calculates that partialout and continues in this manner until the partialout of the outermost loop is calculated.

2.4 Example

The example that follows is a small Pascal program that contains two loops, one of which is nested inside the other. This example includes a step-by-step calculation of the partialin and partialout of each block to emphasize the basic features of the algorithm. The program itself is not meant to do anything, but rather is intended to demonstrate the effects of the loop constructs on
the reaching definitions.

2.4.1 Program

program test

var
    a, b, c, k, d, i, x, y, z: integer;

begin
    (1) a := 5;
    (2) b := 10;
    (3) c := a + b;
    for k := 1 to 5 do begin
        (4) d := a - b;
        (5) a := 3;
        for i := 1 to 3 do begin
            (6) x := b - a;
            (7) y := c + a;
            (8) c := d;
            if (y > x) then
                (9) y := y - a;
            (10) x := y
        end;
        (11) d := x;
    end;
    (12) c := y
end;

(13) z := x + y

end.
2.4.2 Flow Chart

2.4.3 Loop Structures

\[ \text{loop}_{\text{inner}} \rightarrow 0000 \ 1011 \ 1111 \ 100 \]

\[ \text{loop}_{\text{outer}} \rightarrow 0101 \ 1111 \ 1111 \ 110 \]
2.4.4 In-Out Calculations

\[ \text{pin(07)} = \text{empty} \]
\[ \text{pin(08)} = 0000 0111 0000 0 \]
\[ \text{pin(09)} = 0000 0111 0000 0 \]
\[ \text{pin(10)} = 0000 0111 0000 0 \]
\[ \text{pin(11)} = 0000 0111 0000 0 \]
\[ \text{pin(12)} = 0000 0111 0000 0 \]
\[ \text{pin(13)} = 0000 0111 1000 0 \]
\[ \text{pin(04)} = \text{empty} \]
\[ \text{pin(05)} = 0001 1011 1100 0 \]
\[ \text{pin(06)} = 0001 1011 1100 0 \]
\[ \text{pin(14)} = 0001 1011 1100 0 \]
\[ \text{in(01)} = \text{empty} \]
\[ \text{in(02)} = 1110 1010 1111 0 \]
\[ \text{in(03)} = 1110 1010 1111 0 \]
\[ \text{in(04)} = 1110 1010 1111 0 \]
\[ \text{in(05)} = 0111 1011 1101 0 \]
\[ \text{in(06)} = 0111 1011 1101 0 \]
\[ \text{in(07)} = 0111 1011 1101 0 \]
\[ \text{in(08)} = 0101 1111 0000 0 \]

\[ \text{pout(05)} = \text{empty} \]
\[ \text{pout(07)} = 0000 0111 0000 0 \]
\[ \text{pout(08)} = 0000 0111 0000 0 \]
\[ \text{pout(09)} = 0000 0111 0000 0 \]
\[ \text{pout(10)} = 0000 0111 0000 0 \]
\[ \text{pout(11)} = 0000 0111 0000 0 \]
\[ \text{pout(12)} = 0000 0101 1000 0 \]
\[ \text{pout(13)} = 0000 0111 1000 0 \]
\[ \text{pout(02)} = \text{empty} \]
\[ \text{pout(04)} = 0001 1000 0000 0 \]
\[ \text{pout(05)} = 0001 1011 1100 0 \]
\[ \text{pout(06)} = 0001 1011 1100 0 \]
\[ \text{pout(14)} = 0000 1010 1111 0 \]
\[ \text{out(01)} = 1110 0000 0000 0 \]
\[ \text{out(02)} = 1110 1010 1111 0 \]
\[ \text{out(03)} = 1110 1010 1111 0 \]
\[ \text{out(04)} = 0111 1010 1101 0 \]
\[ \text{out(05)} = 0111 1011 1101 0 \]
\[ \text{out(06)} = 0111 1011 1101 0 \]
\[ \text{out(07)} = 0101 1111 0000 0 \]
\[ \text{out(08)} = 0101 1111 0000 0 \]
\[ in(09) = 0101 1111 0000 0 \quad out(09) = 0101 1111 0000 0 \]
\[ in(10) = 0101 1111 0000 0 \quad out(10) = 0101 1111 0000 0 \]
\[ in(11) = 0101 1111 0000 0 \quad out(11) = 0101 1111 0000 0 \]
\[ in(12) = 0101 1111 0000 0 \quad out(12) = 0101 1101 1000 0 \]
\[ in(13) = 0101 1111 1000 0 \quad out(13) = 0101 1011 1100 0 \]
\[ in(14) = 0111 1011 1101 0 \quad out(14) = 0100 1010 1111 0 \]
\[ in(15) = 1110 1010 1111 0 \quad out(15) = 1110 1000 1111 1 \]

### 2.4.5 Analysis

The above example demonstrates the efficiency of our partialout algorithm. We had a total of fifteen blocks with two nested loops. When using the iterative algorithm to determine the reaching definitions, we must make four complete passes through the blocks. Each block requires that four bit manipulations be executed to determine the IN and OUT. Therefore, the total number of bit manipulations executed by the iterative approach is 240. The partialout approach looks first at the innermost loop and calculates the partialin and partialout for only those blocks contained in the inner loop, in this example blocks 5, 7, 8, 9, 10, 11, 12, and 13. It then looks at the outermost loop and calculates the partialin and partialout for those blocks contained in the outer loop but not in the inner loop (plus the headerblock of the inner loop). In this example, the partialin and partialout are now calculated for blocks 2, 4, 5, 6, and 14. And finally, the partialout approach makes a final
pass through all the blocks in sequence, calculating the true IN and OUT of each block. The total number of bit manipulations required for the partialout approach is 104. Given these bit manipulation statistics, we find that the partialout approach for this example is approximately 56% more efficient (faster) than the iterative approach.
Chapter 3

Data Dependence

Sequential languages such as C, Fortran, and Pascal impose a linear order on statement executions. New architectures have many CPUs and provide the opportunity for parallel processing. If the user parallelizes the program so that it will run in a computer with $n$ processors, then the same program will fail to run in computers with less or more than $n$ processors. Automatic parallelism increases the portability by having the compiler parallelize a program based on the number of processors available. And automatic parallelism has the ability to relax the linear order of statement executions so that part of the program may be executed concurrently, while retaining the integrity of the program.

Data dependence analysis provides us with the necessary theory to enable us to build a compiler that divides the user program into parts that may be executed concurrently. The largest gain during the execution time is obtained by parallelizing the code within the inner loops. Often times
parallelization increases the accuracy of the results. Thus if $A(I)$ is an array of random numbers uniformly distributed between $[0,1000]$, then as the $I$ becomes relatively large the variable $S$, which accumulates the sum of the array elements, will become very large and the relatively small values of $A(I)$ do not contribute to the $S$. On the other hand if we parallelize this using, for example 10 processors, we will improve both the speed of execution and accuracy.

A great deal of research has been devoted to the idea of automatic parallelism during the past fifteen years [2, 5, 7, 13, 3]. Automatic parallelism would allow CPU bound programs dealing with such areas as neural networks [9], time series analysis [15], or image processing to be executed in a relatively small time. Currently, when subscripted variables are used within an inner loop, and dependence analysis tests show dependence between array elements defined and array elements used, then the loop is not parallelized. In this chapter we identify these dependencies and split the loop into smaller loops or parallel blocks so that the array accesses within these blocks are independent and can therefore be parallelized.

3.1 Background

We denote the induction variable controlling the loop by $I$, and let $T, U$ be integers denoting the lower and upper limits of this induction variable. We will refer to the induction variable $I$ controlling the loop as the DoVar.
We define a variable \( v \) to be admissible in a loop, if \( v \) is a DoVar, or \( v \) is a subscripted variable with subscript expressions containing no function reference and dependent only on the DoVar. Statement \( S \) at iteration \( i \), will be denoted as \( S(i) \), and statement \( S' \) at iteration \( i' \) will be denoted \( S'(i') \). A true dependence from \( S(i) \) to \( S'(i') \), denoted \( S(i) \delta_i S'(i') \), occurs when a variable \( v \) is defined in \( S(i) \) and used in \( S'(i') \). An antidependence from \( S(i) \) to \( S'(i') \), denoted \( S(i) \delta^a S'(i') \), occurs when a variable \( v \) is used in \( S(i) \) and defined in \( S'(i') \). Finally, an output dependence from \( S(i) \) to \( S'(i') \), denoted \( S(i) \delta^o S'(i') \), occurs when a variable \( v \) is defined in \( S(i) \) and redefined in \( S'(i') \). Notice that in all three cases no definition of \( v \) occurs between the statements \( S(i) \) and \( S'(i') \). A dependence direction shows the relationship between instances of each loop iteration [12, 8]. If the dependence direction is of type \( = \), then a dependence holds within the same loop iteration when the distance vector \( = 0 \) or \( i' - i = 0 \). If the dependence direction is of type \( > \), then a dependence holds from a particular loop iteration back to a previous loop iteration when the distance vector \( > 0 \) or \( i' - i > 0 \). If the dependence direction is of type \( < \), then dependence holds from a particular iteration to a future loop iteration when the distance vector \( < 0 \) or \( i' - i < 0 \). When a statement \( S(i) \) in the \( i \)th iteration is assigned a value and then this value is causing dependence by being carried from one iteration of the loop to the next, we call this dependence loop-carried dependence [16]. If \( L \) denotes a loop and \( STAT(L) \) is the set of assignment statements in \( L \), the
dependence graph of $L$, denoted by $DG(L)$, is the graph of the relation $\delta$ on $STAT(L)$.

If $\mathbb{Z}$ is the set of integers and $a, b \in \mathbb{Z}$ with $g = \gcd(a, b)$ then one could prove there exist integers $u, v \in \mathbb{Z}$ such that $g = au + bv$. A linear diophantine equation is an equation of the form $\sum_{i=1}^{n} a_i x_i = c$ where $a_j \in \mathbb{Z}$, $j = 1, \ldots, n$, $c \in \mathbb{Z}$ and $n \in \mathbb{N}$, with $n \geq 1$.

**Theorem 3.1** A linear diophantine equation $\sum_{i=1}^{n} a_i x_i = c$ with $g = \gcd(a_1, \ldots, a_n)$, has a solution if and only if $g$ divides $c$.

**Theorem 3.2** Let $ax + by = c$ denote a linear diophantine equation, assume that $g = \gcd(a, b)$, $g$ divides $c$ and that $u, v$ are integers with $g = au + bv$. Then the set of all solutions $(x_t, y_t)$, $t \in \mathbb{Z}$, of the equation is given as follows:

$$
\begin{align*}
x_t &= \frac{uc}{g} + \frac{b}{g} \\
y_t &= \frac{vc}{g} - \frac{a}{g}
\end{align*}
$$

(3.1)

Proofs to the above two theorems can be found in [16]. Let us show some examples to demonstrate how the above theorems are used. Consider the following loop:

```
DO I = 20, 60
   S : A(6*I + 1) = \cdots
   ;
   S' : \cdots = A(4*I - 6)
END DO
```
In order for the statements $S$ and $S'$ to be dependent, there must be values $x, y$ of $I$ such that the array element accessed by statement instance $S(x)$ is the same as the one accessed by statement instance $S'(y)$. This implies that there must exist a solution to the dependence equation $6x + 1 = 4y - 6$ or $6x - 4y = -7$. The greatest common divisor ($gcd$) of 6 and 4 is 2. This does not divide 7, hence statements $S$ and $S'$ are independent. Therefore, this loop can be restructured into a DoAll block as follows:

\[
\text{DOALL } I = 20, 60 \\
S : A(6 \ast I + 1) = \cdots \\
\vdots \\
S' : \cdots = A(4 \ast I - 6) \\
\text{END DOALL}
\]

This DoAll structure allows for each iteration of the loop to be executed by a different processor in parallel. Instead of using forty time units to execute this loop, we can use only one time unit and forty processors all working concurrently. Next, consider the following loop:

\[
\text{DO } I = 1, 20 \\
S : A(2 \ast I + 3) = \cdots \\
\vdots \\
S' : \cdots = A(I - 1) \\
\text{END DO}
\]

Here, our diophantine equation is $2x + 3 = y - 1$ with $a = 2$, $b = 1$, and $c = -4$. Our $g = gcd(a, b) = 1$. Since $g$ divides $c$, we know that some kind of
dependence exists between statements $S$ and $S'$. Therefore, this loop cannot be restructured into a DoAll block without further investigation.

### 3.2 Increase of Parallelism

If the diophantine equation corresponding to the arguments of the matrices that could be true dependent, antidependent, or output dependent, has a solution and the solution is within the boundaries of the loop, formerly parallelization of the loop was impossible. However, if the boundaries of the loop are known, parallelism can be achieved by splitting the loop into smaller loops or parallel blocks, each of which has no dependence. The process works as follows:

- Identify the arrays that could possibly cause dependence.
- Form the diophantine equations corresponding to the argument of the arrays identified as a possible cause of dependence.
- Check if the diophantine equations have integer solutions within the loop boundaries. If no solutions exists, then the loop can be parallelized or restructured into a single DoAll block and the process finishes. Otherwise, we find those solutions.
- The solutions enable us to find the values of the index variable $I$ for which two or more statements of the loop access the same memory location. When this happens we say that we have a collision.
• Let \( i_1 < i_2 < \ldots < i_n \) be the ordered sequence of index values for which collisions could happen. Then if \( T, U \) are the known lower and upper values of the loop respectively, with \( T \leq i_j \leq U, j = 1, 2, \ldots, n \), then we divide the loop into at most \( n + 1 \) smaller loops, each of which can be parallelized.

• If \( T < i_1 - 1 \) then the first loop has lower limit \( T \) and upper limit \( i_1 - 1 \), otherwise the code is sequential and is preceding the first loop.

The loop corresponding to the \( i_{j-1} \) and \( i_j \) values of \( I \), has lower limit \( i_{j-1} + 1 \) and upper limit \( i_j - 1 \).

• For \( I = i_1, \ldots, i_n \), the code is executed sequentially.

The following example shows how the loop splitting works:

\[
\text{DO } I = 6, 20 \\
\quad S: A(6 \times I + 3) = \cdots \\
\quad \vdots \\
\quad S': \cdots = A(3 \times I - 9) \\
\text{END DO}
\]

The diophantine equation corresponding to the array arguments of the above example is \( 6x + 3 = 3y - 9 \), with \( 6 \leq x, y \leq 20 \). A loop-independent dependency exists if \( 6x + 3 = 3x - 9 \), or \( x = -4 \). Since \(-4\) is not in the interval \([6, 20]\), there is no loop independent dependence between \( S \) and \( S' \).

A true dependence exists if the above diophantine equation has a solution \((x, y)\) with \( 6 \leq x < y \leq 20 \). The diophantine equation can be written as
$6x - 3y = -12$. Then $g = \gcd(6, -3) = 3$ and since $g = 6u - 3v$ we have $u = 1, v = 1$. Our parametric equations are $x_t = -4 - t$ and $y_t = -4 - 2t$. Thus $S$ is true dependent on $S'$ if $6 \leq -4 - t < -4 - 2t \leq 20$ which implies $-12 \leq t \leq -10$. For $t = -10$, we have $x = 6, y = 16$. A common access to location $A(39)$ causes a true dependence between $S$ and $S'$. Also, for $t = -11$, we have $x = 7, y = 18$. The common location accessed is $A(45)$. Finally, for $t = -12$ we obtain $x = 8, y = 20$ and the common location accessed is $A(51)$. In order for an antidependence to exist, the inequality $6 \leq -4 - 2t < -4 - t \leq 20$ must be satisfied. Since the above inequality is not satisfied, an antidependence does not exist. The loop can be split into the following sequence. Remember, the DoAll block means the iterations of the loop (in this case, from $I = 9, 15$) have no dependence and can be executed in parallel.

**DO I = 6, 8**

\[ S : A(6 * I + 3) = \cdots \]
\[ \vdots \]
\[ S' : \cdots = A(3 * I - 9) \]

**END DO**

**DOALL I = 9, 15**

\[ S : A(6 * I + 3) = \cdots \]
\[ \vdots \]
\[ S' : \cdots = A(3 * I - 9) \]

**END DOALL**

**DO I = 16, 20**
\[ S : A(6I + 3) = \cdots \]
\[ \vdots \]
\[ S' : \cdots = A(3I - 9) \]

\textbf{END DO}
Chapter 4

Parallelization

In the last chapter we explained what data dependence is, we discussed the three different types of dependence direction, and we showed how, with the use of diophantine equations, we could determine the index conflict areas or collisions of a loop and suggested a way of splitting the loop so that pieces of it could be done in parallel as opposed to a purely sequential execution. We split the loop at each collision assuring that no two collisions resided in the same parallel block. But our loop splitting schema did not optimally parallelize the loop. In this chapter we present algorithms to optimally split a loop into a sequence of parallel blocks. By optimally we mean using the greatest number of processors and the least number of time units possible. Each iteration of a parallel block may be executed simultaneously, whereas the parallel blocks themselves must be executed sequentially. These algorithms insure that no two members of an index conflict or collision reside in the same parallel block. Consequently, these algorithms guarantee that the integrity of the original
program is maintained and the results from the loop-splitting algorithms is the same as the results from the original program.

We used a loop restructuring tool called tiny which was written by M. Wolfe of the Oregon Graduate Institute. Tiny, given a specially formatted program with loops and statements within the loop body attempts to restructure a loop so that the loop can be parallelized into a DoAll block. A DoAll block signifies that each iteration of the loop is independent and consequently can be executed in parallel. If tiny is presented with two statements of a loop which have a loop-carried dependence, tiny will return the following message: Unable to parallelize due to data dependence. At this point in the tiny application, we include our procedure that contains the algorithms for loop-splitting and optimal parallelization. Our procedure writes out to a file the sequential series of parallel blocks for loop-splitting. The procedure will ask for a lower and upper bound (the real values of $T$ and $U$, respectively) for run-time and split the loop into sequential parallel blocks, giving the optimal parallelizations we seek.

We have subdivided the parameters of the diophantine equations, namely the values of $a$, $b$, and $c$ into several cases. We first present the logic for each case, followed by the algorithm itself, and then an example. The first and most trivial case occurs when $a = b$ and $c = 0$. We know from the Theorem 3.1 that a dependence can exist between two statements in a loop $S$ and $S'$ only if $g = \text{gcd}(a, b)$ divides $c$. Since, in our trivial case, $c = 0$, $g$ divides $c$ and
since $a = b$, we have a dependence direction of type $=$ or a loop-independent dependence. Consequently, given these parameters, we know that there is no dependence across loop iterations. The dependence between statements will fall within each loop iteration. Therefore the DoAll block will suffice to maximally parallelize any loop with these parameters. For an example, consider the following loop:

```
DO I = 1, 5
  S : A(2 * I + 3) = \ldots
  \ldots
S' : \ldots = A(2 * I + 3)
END DO
```

Here, $a = 2$, $b = 2$, and $c = 0$. Therefore, our diophantine equation is $2x = 2y$ and the lower bound is $T = 1$ with an upper bound of $U = 5$. If we unroll this loop it would look like the following:

```
S_1 : A(5) = \ldots
  \ldots
S_1' : \ldots = A(5)
S_2 : A(7) = \ldots
  \ldots
S_2' : \ldots = A(7)
S_3 : A(9) = \ldots
  \ldots
S_3' : \ldots = A(9)
S_4 : A(11) = \ldots
```
As is evident, the dependence falls within each loop iteration. Tiny, the restructuring tool we used, is able to determine that the dependence falls within each loop iteration. Consequently, given these parameters, tiny will restructure the loop into a DoAll block that looks like the following:

\begin{verbatim}
DOALL I = 1, 5
  S : A(2*I + 3) = ···
  ···
  S' : ··· = A(2*I + 3)
END DOALL
\end{verbatim}

Given this DoAll block, each iteration of the loop will be executed by a different processor or in parallel. In the following sections, we will review all the cases that exist for our diophantine equation parameters and present algorithms that will split the loop into a series of DoAll blocks. These sequential DoAll blocks will permit maximum parallelization and optimal processor utilization.
4.1 Case 1

4.1.1 Parameters

Let $a = b$ and $c > 0$.

4.1.2 Logic

Let $g = \text{gcd}(a, b) = a = b$. If $g$ does not divide $c$ there is no dependence and tiny will restructure the loop into a DoAll block. If $g$ does divide $c$, we have a dependence direction of type $\text{<}$. The GCD Test is not an exact test, but rather 'provides a sufficient and necessary condition for the existence of a solution to the dependence equation.' (Zima). So we explore in more depth. We wish to find the first indexes for $x$ and $y$ which results in the array elements accessing the same memory location. When array elements access the same memory location, we call this a collision. Let $s$ equal $c$ divided by $g$. We will denote this as $s = c/g$. Our first collision will be at $yindex = T$ and $xindex = T + s$ where $T$ is the lower limit. Given the dependence direction of type $\text{<}$, we must be assured that the $yindex$ memory access is executed before the $xindex$ memory access takes place. Thus we must place each member of this collision in a separate parallel block with the $yindex$ in a parallel block that is sequentially executed before the parallel block which contains the $xindex$. We can accomplish this by splitting the loop between $xindex - 1$ and $xindex$. If $xindex \geq U$ where $U$ is our upper bound, the entire loop can be put into a single DoAll block as all collisions
reside outside of our region of execution. Otherwise, we must continue to find the conflicts within our region of execution, until \( x_{\text{index}} \geq U \). We do this by letting \( x_{\text{index}} = x_{\text{index}} + s \) and splitting the loop again between \( x_{\text{index}} - 1 \) and \( x_{\text{index}} \). Since \( s = c/g \) represents the distance between collisions, by adding \( s \) to \( x_{\text{index}} \) and splitting the loop between \( x_{\text{index}} - 1 \) and \( x_{\text{index}} \) we are guaranteed that all members of the collisions are in separate parallel blocks.

### 4.1.3 Algorithm

The algorithms presented in this chapter are written in pseudocode that resembles the C programming language. The purpose of the pseudocode is to make a transition from the logic and theory to the programming concept. A full set of routines have been implemented in our restructuring tool, tiny. The choice of using pseudocode here is mainly for clarification purposes. Certain routines which perform a specific function are called by these algorithms. First, a routine called `printdoall()` is used. The `printdoall` routine prints the `DoAll` block from a specified lower bound to a specified upper bound replicating statements \( S \) and \( S' \). For example, if the diophantine equation is \( x = y + 1 \), a call to `printdoall(2, 5)` will produce the following output:

\[
\text{DOALL } I = 2, 5 \\
S : A(I) = \cdots \\
:\ldots \\
S' : \cdots = A(I + 1)
\]
This routine is used to print the set of sequential parallel blocks. Another routine, \texttt{findgcd()} calculates $g$, the greatest common divisor of $a, b$ of our diophantine equation and finds the parameters $u$ and $v$ for the equation $g = au - bv$.

/* Case 1 and Case 2
   \[ a = b \text{ and } c \neq 0 \]
   \textit{global variables}
   \[ \text{int } a, b, a_0, b_0, T, U \]
/*

\texttt{main()}
{
  \text{int } c, s, index_1, index_2
  \[ c = \text{abs}(b_0 - a_0) \]
  \[ s = c / a \]
  \[ index_1 = T \]
  \[ index_2 = index_1 + s - 1 \]
  \texttt{while}(index_2 < U)\
  \{ \texttt{printdoall(index_1, index_2)}
  \[ index_1 = index_2 + 1 \]
  \[ index_2 = index_2 + s \]
  \}
  \texttt{printdoall(index_1, U)}
}
4.1.4 Example

Consider the following loop:

\[
\begin{align*}
\text{DO } I &= 1, 20 \\
S : A(I) &= \cdots \\
\vdots & \\
S' : \cdots &= A(I + 4)
\end{align*}
\]

END DO

Our diophantine equation is \(x = y + 4\) where \(a = 1, b = 1,\) and \(c = 4.\) The \(gcd(a, b) = 1\) or \(g = 1.\) Therefore, \(s = c/g = 4.\) Our first collision is at \(x_{index} = T + s = 5\) and \(y_{index} = T = 1.\) Thus our first loop split is between \(x_{index} = 4\) and \(x_{index} = 5.\) Our algorithm for this example prints the following set of sequential parallel blocks:

\[
\begin{align*}
\text{DOALL } I &= 1, 4 \\
S : A(I) &= \cdots \\
\vdots & \\
S' : \cdots &= A(I + 4)
\end{align*}
\]

END DOALL

\[
\begin{align*}
\text{DOALL } I &= 5, 8 \\
S : A(I) &= \cdots \\
\vdots & \\
S' : \cdots &= A(I + 4)
\end{align*}
\]

END DOALL

\[
\begin{align*}
\text{DOALL } I &= 9, 12 \\
S : A(I) &= \cdots
\end{align*}
\]
\[ S' : \cdots = A(I + 4) \]

**END DOALL**

**DOALL I = 13, 16**

\[ S : A(I) = \cdots \]

\[ S' : \cdots = A(I + 4) \]

**END DOALL**

**DOALL I = 17, 20**

\[ S : A(I) = \cdots \]

\[ S' : \cdots = A(I + 4) \]

**END DOALL**

### 4.1.5 Time Factor

We need some method for comparing the number of sequential DoAll blocks executed with the parallel algorithm to the number of sequential loop iterations executed for when no parallelization is done. If we consider each iteration of a loop a single time unit when executed sequentially and each DoAll block a single time unit when executed in parallel, we can compare the two and determine the decrease in execution time brought about by the parallel algorithm. The number of sequential DoAll blocks executed is dependent on \( s = c/g \). If we let \( t = [(U - T + 1)/s] \), \( t \) will signify the total number of sequential DoAll blocks or time units needed to execute the code.
$U - T + 1$ represents the number of time units needed to execute the code in a purely sequential manner. By comparing $t$ to $U - T + 1$, we can get an idea of the decrease in execution time or how much faster the parallel algorithm will execute. In the example of this section, $t = 5$ and $U - T + 1 = 20$. Thus we find a 5:20 ratio which signifies that the parallel algorithm in this example can execute the code 4 times as fast as the sequential code.

4.2 Case 2

4.2.1 Parameters

Let $a = b$ and $c < 0$.

4.2.2 Logic

Let $g = \gcd(a, b) = a = b$. If $g$ does not divide $c$ there is no dependence and tiny will restructure the loop into a DoAll block. If $g$ does divide $c$, then we have a dependence direction of type $>$. Again we wish to find the first indexes for $x$ and $y$ which results in the array elements accessing the same memory location in our given region of execution. Given these parameters, let $c = |c|$ (we want distance, and are not concerned about direction). Then let $s = c/g$. Now our first collision is when $xindex = T$ and $yindex = T + s$. This case is similar to section 4.1/ In section 4.1 the right side of the diophantine equation is always greater then the left side and the dependence direction is always of type $<$. In this case, with $c < 0$, ...
the left side of the diophantine equation is always greater than the right side and the dependence direction is always of type \(>\). Given this dependence direction, we must be assured that the \(x_{\text{index}}\) memory access is executed before the \(y_{\text{index}}\) memory access takes place. Consequently, we place each member of this collision in a separate parallel block with the \(x_{\text{index}}\) in a parallel block that is sequentially executed before the parallel block which contains the \(y_{\text{index}}\). We can accomplish this by splitting the loop between \(y_{\text{index}} - 1\) and \(y_{\text{index}}\). If \(y_{\text{index}} \geq U\) then the loop can be executed as a \texttt{DoAll} block as all conflicts reside outside of our region of execution. While \(y_{\text{index}} < U\), we must continue to search for all conflicts that reside in the region of execution. If we let \(y_{\text{index}} = y_{\text{index}} + s\) and split the loop between \(y_{\text{index}} - 1\) and \(y_{\text{index}}\), we are assured that all members of the collisions will reside in unique parallel blocks. We repeat this procedure until \(y_{\text{index}} \geq U\).

### 4.2.3 Algorithm

See section 4.1.3

### 4.2.4 Example

Consider the following loop:

\[
\text{DO } I = 4, 22 \\
S : A(3 \ast I) = \cdots \\
\vdots \\
S' : \cdots = A(3 \ast I - 6)
\]
Our diophantine equation is $3x = 3y - 6$ where $a = 3$, $b = 3$, and $c = -6$. Let $c = |c|$ so that $c = 6$. The $gcd(a, b) = 3$ or $g = 3$. Thus, $s = c/g = 2$. Our first collision is at $xindex = T = 4$ and $yindex = T + s = 6$. In other words, when $I = 4$, $S$ accesses memory location $A(12)$ and when $I = 6$, $S'$ accesses the same memory location. Consequently, our first loop split is between $yindex - 1 = 5$ and $yindex = 6$. Our algorithm for this example prints the following sets of sequential parallel blocks:

**DOALL $I = 4, 5$**

$S : A(3 * I) = \cdots$

\vdots

$S' : \cdots = A(3 * I - 6)$

**END DOALL**

**DOALL $I = 6, 7$**

$S : A(3 * I) = \cdots$

\vdots

$S' : \cdots = A(3 * I - 6)$

**END DOALL**

**DOALL $I = 8, 9$**

$S : A(3 * I) = \cdots$

\vdots

$S' : \cdots = A(3 * I - 6)$

**END DOALL**

**DOALL $I = 10, 11$**
\begin{verbatim}
  S : A(3* I) = ⋯
  ⋮
  S' : ⋯ = A(3* I - 6)
END DOALL
DOALL I = 12, 13
  S : A(3* I) = ⋯
  ⋮
  S' : ⋯ = A(3* I - 6)
END DOALL
DOALL I = 14, 15
  S : A(3* I) = ⋯
  ⋮
  S' : ⋯ = A(3* I - 6)
END DOALL
DOALL I = 16, 17
  S : A(3* I) = ⋯
  ⋮
  S' : ⋯ = A(3* I - 6)
END DOALL
DOALL I = 18, 19
  S : A(3* I) = ⋯
  ⋮
  S' : ⋯ = A(3* I - 6)
END DOALL
DOALL I = 20, 21
  S : A(3* I) = ⋯
\end{verbatim}
36

\[ S' : \cdots = A(3 \cdot I - 6) \]

\textbf{END \textsc{DOALL}}

\textbf{DOALL} \textsc{I} = 22

\[ S : A(3 \cdot I) = \cdots \]

\[ S' : \cdots = A(3 \cdot I - 6) \]

\textbf{END \textsc{DOALL}}

\section*{4.2.5 Time Factor}

The number of sequential \textsc{DoAll} blocks executed using the parallel algorithm is again dependent on \( s = \frac{c}{g} \) where \( c = |c| \). In this example, our \( t = \left\lfloor \frac{(U - T + 1)}{s} \right\rfloor = 10 \) and \( U - T + 1 = 19 \) for a ratio of 10:19. This ratio implies that the parallel algorithm is approximately 2 (exactly 1.9) times faster than the sequential execution.

\section*{4.3 Case 3}

\subsection*{4.3.1 Parameters}

Let \( a = b \) and \( s = \frac{c}{g} = \pm 1 \).

\subsection*{4.3.2 Logic}

Let \( g = \gcd(a, b) = a = b \). If \( g \) does not divide \( c \) there is no dependence and \texttt{tiny} will restructure the loop into a single \texttt{DoAll} block. If \( g \) does divide \( c \)
and $s = \pm 1$ we have a dependence direction of type $>$ or $<$ depending on the sign of $s$ with a collision distance of 1. Examples of diophantine equations that fall into this case are $x = y + 1$, $4x = 4y - 4$, and $3x + 1 = 3y + 4$. This is the worst case situation for parallelizing a sequential loop. Since the distance between dependencies is only 1, each iteration of the loop is dependent upon the next or previous iteration depending on the sign of $s$. Loops with these parameters cannot be parallelized and must be executed sequentially to maintain the integrity of the original program.

4.3.3 Example

Consider the following loop:

```
DO I = 5, 10
    S : A(3*I + 1) = \cdots
    \vdots
    S': \cdots = A(3*I + 4)
END DO
```

Our diophantine equation is $3x + 1 = 3y + 4$ where $a = 3$, $b = 3$, and $c = 3$. The $gcd(a, b) = 3$ or $g = 3$. Thus, $s = c/g = 1$. If we unroll this loop and plot the memory access locations we will find the following:

$S_5 : A(16) = \cdots$

$\vdots$

$S'_5 : \cdots = A(19)$

$S_6 : A(19) = \cdots$
As we can see, with a dependence distance of 1, each iteration of the loop must be executed sequentially to insure that the access to the memory location made by the statement \( S_i \) is executed prior to the reassignment of the memory location made by the statement \( S_i \).

### 4.4 Case 4

#### 4.4.1 Parameters

Let \( a > 0, b < 0 \) or \( a < 0, b > 0 \).
4.4.2 Logic

Let $g = \gcd(a, b)$. To avoid accessing negative memory locations, we assume the diophantine equation is such that $c > by$ or in the second case $c > ax$. Examples of diophantine equations that fall under this case are $2x + 3 = 51 - 3y$ or $15 - 2x = 2y + 3$. Our restructuring tool, tiny, will normalize any loop that accesses negative memory locations thus reassigning all memory locations to positive numbers. Therefore, we need not be concerned with negative memory locations or is that the intent of this research. If $g$ does not divide $c$ then tiny will restructure the loop into a DoAll block, since we have no data dependence. If $g|c$, then given the parameters $a > 0, b < 0$ we note that the left side of the diophantine equation increases and the right side decreases as the induction variable increases. Consequently, at some point as the left side increases and the right side decreases, we have a change of direction (cod) point or a point where the left side becomes greater than the right side. Given the other parameter $a < 0, b > 0$ we have the same situation but reversed with the left side of the diophantine equation decreasing as the right side increases. And again we recognize a cod point.

In order to optimally parallelize this loop, we want to split the loop at the point of this direction change. Referring again to the first set of parameters, if we let $w = |a| + |b|$, we calculate $z = (\text{int})c/w$. Our $z$ now represents the last index of $x$ where the left side of the diophantine equation is smaller than the right side or the point just before the direction change. Therefore
we split the loop between \( x_{\text{index}} = z \) and \( x_{\text{index}} = z + 1 \). Considering the second set of parameters \( a < 0, b > 0 \) our \( z \) represents the last index of \( x \) where the right side of the diophantine equations is smaller than the left side. Splitting the loop at the cod point guarantees that no two elements of any of the conflicts share the same parallel block.

4.4.3 Algorithm

/ * Case 4 

\( a > 0, b < 0 \) or \( a < 0, b > 0 \)

global variables:

\( \text{int } a, b, a_0, b_0, T, U \)

\( c = b_0 - a_0 \)

\( w = \text{abs}(a) + \text{abs}(b) \)

\( z = \lfloor c/w \rfloor \)

if\( (T > z) \)

printdoall\( (T, U) \)

else{

printdoall\( (T, z) \)

printdoall\( (z + 1, U) \)

}

4.4.4 Example

Consider the following loop:

\[
\text{DO } I = 1, 15 \\
S : A(2 \ast I + 3) = \cdots
\]
Our diophantine equation is $2x + 3 = 51 - 3y$ with $a = 2$, $b = -3$, and $c = 48$. These parameters coincide with our first set of parameters of section 4.4.2. Let $w = |a| + |b| = 5$. Then $z = (int)c/w = 9$. Now we split the loop between $x_{index} = z = 9$ and $x_{index} = z + 1 = 10$. Our algorithm for this example prints the following two sequential parallel blocks:

```
DOALL I = 1, 9
  S : A(2*I + 3) = \cdots

S' : \cdots = A(51 - 3*I)

END DOALL
```

```
DOALL I = 10, 15
  S : A(2*I + 3) = \cdots

S' : \cdots = A(51 - 3*I)

END DOALL
```

### 4.4.5 Time Factor

With these parameters, the number of sequential **DoAll** blocks it takes to execute the code is always two. We simply split the loop one time at the **cod** point and this insures no two members of a collision reside in the same parallel block. Therefore, $t = 2$. Consequently, in our example, we have
$t = 2$ and $U - T + 1 = 15$. Thus our ratio is $2:15$ and we find that the parallel algorithm is $7.5$ times faster than the sequential execution. As the value of $U - T + 1$ becomes quite large (we have many iterations of the loop) the decrease in execution time for the parallel algorithm becomes quite significant.

4.5 Case 5

4.5.1 Parameters

Let $a > b$ and $c = 0$.

4.5.2 Logic

Let $g = \gcd(a, b)$. One solution set of the diophantine equation is given by $x_0 = u \times c/g$ and $y_0 = v \times c/g$ where $u$ and $v$ are derived from the equation $g = au - bv$ [16]. Since $c = 0$, we know one solution set of the diophantine equation is $x_0 = 0, y_0 = 0$ or $(0,0)$. Given the parameter $a > b$, we have a dependence direction of type $>$. Consequently, if we let $z = \lfloor T \times g/b \rfloor$, then our first conflict is at $x_{index} = b/g \times z$ and $y_{index} = a/g \times z$. To insure that no two members of a collision reside in the same parallel block, we split the loop between $y_{index} - 1$ and $y_{index}$. To find the next parallel block, we increment $z$, let $oldy_{index} = y_{index}$, and again calculate $x_{index} = b/g \times z$ and $y_{index} = a/g \times z$. We could at this point split the loop between $y_{index} - 1$ and $y_{index}$ which guarantees that no elements of a collision reside
in the same parallel block. However, since the left side of the diophantine equation is increasing more rapidly than the right side, it is not necessary to split the loop at every increment of \( z \). Instead, at each increment of \( z \), if \( x_{\text{index}} \geq \text{oldy}_{\text{index}} \) we split the loop between \( y_{\text{index}} - 1 \) and \( y_{\text{index}} \). Otherwise, we do not split the loop but instead only increment \( z \) again and recalculate \( x_{\text{index}} \) and \( y_{\text{index}} \). As soon as \( x_{\text{index}} \geq \text{oldy}_{\text{index}} \), we again split the loop between \( y_{\text{index}} - 1 \) and \( y_{\text{index}} \). Continuing in this manner until \( y_{\text{index}} \geq U \), we have optimally parallelized the loop.

### 4.5.3 Algorithm

```c
/* Case 5 and Case 6
   a > b and c = 0 or b > a and c = 0
   global variables
   int a, b, a0, b0, T, U */

main()
{
  int m, index1, index2, flag, A
  float z1, m1
  findgcd()
  if a > b
    z1 = T * g/b
  else
    z1 = T * g/a
  m = (int)z1
  m1 = (float)m
```
if \( z_1 == m_1 \)
\[ z = z_1 \]
else
\[ z = z_1 + 1 \]
if \( a > b \) {
\[ \text{index}_1 = b/g * z \]
\[ \text{index}_2 = a/g * z \]
}
else {
\[ \text{index}_1 = a/g * z \]
\[ \text{index}_2 = b/g * z \]
}
flag = 1
A = T
while \( \text{index}_2 \leq U \) {
if (flag) {
printdoall \((A, \text{index}_2 - 1)\)
oldindex = \text{index}_2
A = \text{index}_2
}
flag = 1
z++
if \( a > b \) {
\[ \text{index}_1 = b/g * z \]
\[ \text{index}_2 = a/g * z \]
}
else {
\[ \text{index}_1 = a/g * z \]
\begin{verbatim}
index_2 = b/g * z

} else
flag = 0

} printdoall (A, U)

\end{verbatim}

4.5.4 Example

Consider the following loop:
\begin{verbatim}
DO I = 1, 20
   S: A(3*I - 2) = ...
   ...
   S': ... = A(2*I - 2)
END DO
\end{verbatim}

Our diophantine equation is \(3x - 2 = 2y - 2\) with \(a = 3, b = 2,\) and \(c = 0.\) Then \(z = [T*g/b] = 1.\) Our first collision is at \(xindex = b/g * z = 2\) and \(yindex = a/g * z = 3.\) Thus our first split is between \(yindex - 1 = 2\) and \(yindex = 3.\) Our algorithm for this example will print out the following set of sequential parallel blocks:
\begin{verbatim}
DOALL I = 1, 2
   S: A(3*I - 2) = ...
   ...
\end{verbatim}
\[ S' : \cdots = A(2 \times I - 2) \]

END DOALL

DOALL \( I = 3, 5 \)
\[ S : A(3 \times I - 2) = \cdots \]
\[ \vdots \]
\[ S' : \cdots = A(2 \times I - 2) \]

END DOALL

DOALL \( I = 6, 8 \)
\[ S : A(3 \times I - 2) = \cdots \]
\[ \vdots \]
\[ S' : \cdots = A(2 \times I - 2) \]

END DOALL

DOALL \( I = 9, 14 \)
\[ S : A(3 \times I - 2) = \cdots \]
\[ \vdots \]
\[ S' : \cdots = A(2 \times I - 2) \]

END DOALL

DOALL \( I = 15, 20 \)
\[ S : A(3 \times I - 2) = \cdots \]
\[ \vdots \]
\[ S' : \cdots = A(2 \times I - 2) \]

END DOALL
4.5.5 Time Factor

To find the number of sequential DoAll blocks, given the parameters \( a > b \) and \( c = 0 \), we must first determine the number of collisions that occur between a given \( T \) and \( U \). We know that \( g = \gcd(a, b) \) and we can rewrite our diophantine equation such that \( ax - by = c \). Since \( g = au - bv \), we can solve for \( u \) and \( v \). Then using equation 3.2, we can find the parametric equations \( x_t \) and \( y_t \). Now we must solve the inequality \( T < x_t < y_t < U \) such that \( T' < t < U' \). Let \( n \) denote the maximum number of loop splits required. Then \( n = U' - T' + 1 \). The maximum number of loop splits would insure that no two members of a collision reside in the same parallel block. But since \( a > b \), we know that the left side of the diophantine equation is increasing more rapidly than the right side and therefore we may not need to split the loop for every collision (or for every increment of \( t \)). Our algorithm checks if \( xindex < oldyindex \) to determine if we need to split the loop for the respective collision. If we keep a counter, denoted \( k \), and increase \( k \) each time \( xindex < oldyindex \), then \( k \) will represent the number of times we do not need to split the loop since the members of each collision are already in a unique parallel block. Now the total number of splits needed, denoted \( l \), is \( n - k \). Our \( l \) now represents the optimal number of splits required. Since the number of sequential DoAll blocks is one greater that the number of optimal splits, we have \( t = l + 1 \). Now by comparing \( t \) to \( U - T + 1 \), we can represent our ratio as in the cases above. For this example, our diophantine equation
is $3x - 2 = 2y - 2$. Our $g = \gcd(a, b) = 1$. We rewrite the diophantine equation such that $3x - 2y = 0$. Since $g = au - bv$, our $u = 1$ and $v = 1$. Then $x_t = -2t$ and $y_t = -3t$. Now our inequality is $1 \leq -2t < -3t \leq 20$ such that $-\frac{20}{3} \leq t \leq -\frac{1}{2}$. Since we are only interested in integer solutions we rewrite the inequality as $-6 \leq t \leq -1$. Then $n = U' - T' + 1 = 6$. We find that $k = 2$, so that $l = n - k = 4$ and thus $t = 5$. Then our ratio is 5:20 so that our parallel algorithm for this example executes the loop 4 times faster than the sequential execution.

4.6 Case 6

4.6.1 Parameters

Let $b > a$ and $c = 0$.

4.6.2 Logic

This is a similar situation to section 4.5 except that with these parameters, the right side of the diophantine equation is greater and increases more rapidly than the left side. Remember that in section 4.5, the left side was greater than the right and increased more rapidly. Since $c = 0$, we know one solution set to our diophantine equation is $(0, 0)$. Given the current parameters ($b > a$) we have a dependence direction of type $<$. Consequently, if we let $z = \lceil T * g/a \rceil$, then our first collision is at $x_{\text{index}} = b/g * z$ and $y_{\text{index}} = a/g * z$. To insure that no two members of a collision reside in the
same parallel block, we split the loop between \( x_{\text{index}} - 1 \) and \( x_{\text{index}} \). To find
the next parallel block, we increment \( z \), let \( \text{old}x_{\text{index}} = x_{\text{index}} \), and again calculate \( x_{\text{index}} = b/g \times x \) and \( y_{\text{index}} = a/g \times z \). As before, we could at this
point split the loop at \( x_{\text{index}} - 1 \) and \( x_{\text{index}} \) which would guarantee that
no elements of the collision reside in the same parallel block. However, since
the right side of the equation is increasing more rapidly than the left side, it
is not necessary to split the loop at every increment of \( z \). Instead, for each
increment of \( z \), if \( y_{\text{index}} \geq \text{old}x_{\text{index}} \) we split the loop between \( x_{\text{index}} - 1 \)
and \( x_{\text{index}} \). Otherwise, we need not split the loop since the elements of the
collision do not reside in the same parallel block. Thus, we just increment \( z \)
and recalculate \( x_{\text{index}} \) and \( y_{\text{index}} \). Again, we continue in this manner until
\( x_{\text{index}} \geq U \). At this time we have an optimally parallelized loop.

4.6.3 Algorithm

See section 4.5.3.

4.6.4 Example

Consider the following loop:

\[
\text{DO I = 1, 25} \\
\quad S : A(3 \times I) = \cdots \\
\quad \vdots \\
\quad S' : \cdots = A(4 \times I) \\
\text{END DO}
\]
Our diophantine equation is $3x = 4y$ with $a = 3$, $b = 4$, and $c = 0$. Then $z = \lceil T \ast g/a \rceil = 1$. Our first collision is at $x_{index} = b/g \ast z = 4$ and $y_{index} = a/g \ast z = 3$. So we split the loop between $x_{index} - 1 = 3$ and $x_{index} = 4$. Our algorithm for this example prints the following set of sequential parallel blocks:

**DOALL** I = 1, 3

\[
S : A(3 \ast I) = \cdots
\]

\[
\vdots
\]

\[
S' : \cdots = A(4 \ast I)
\]

**END DOALL**

**DOALL** I = 4, 7

\[
S : A(3 \ast I) = \cdots
\]

\[
\vdots
\]

\[
S' : \cdots = A(4 \ast I)
\]

**END DOALL**

**DOALL** I = 8, 11

\[
S : A(3 \ast I) = \cdots
\]

\[
\vdots
\]

\[
S' : \cdots = A(4 \ast I)
\]

**END DOALL**

**DOALL** I = 12, 15

\[
S : A(3 \ast I) = \cdots
\]

\[
\vdots
\]

\[
S' : \cdots = A(4 \ast I)
\]

**END DOALL**
DOALL I = 16, 23
   S : A(3 * I) = ⋯
   :
   S' : ⋯ = A(4 * I)
END DOALL

DOALL I = 24, 25
   S : A(3 * I) = ⋯
   :
   S' : ⋯ = A(4 * I)
END DOALL

4.6.5 Time Factor

We follow the same process here as in section 4.5.5 to determine the total number of sequential DoAll blocks. Except in this case, \( b > a \) so that now the right side of the diophantine equation is increasing more rapidly than the left side and therefore we may not need to split the loop for every collision. In this case our algorithm checks if \( yindex < oldxindex \) to determine if we need to split the loop for the respective collision. Therefore, our \( k \) will now be incremented each time \( yindex < oldxindex \). For this example, our diophantine equation is \( 3x - 4y \). Our \( g = gcd(a, b) = 1 \). We rewrite the diophantine equation such that \( 3x - 4y = 0 \). Since \( g = au - bv \), our \( u = 1 \) and \( v = 1 \). Then \( x_t = -4t \) and \( y_t = -3t \). Now our inequality is \( 1 \leq -3t < -4t \leq 25 \) such that \( -\frac{25}{4} \leq t < -\frac{1}{3} \). Since we are only interested in integer solutions we rewrite the inequality as \( -6 \leq t \leq -1 \). Then \( n = U'-T'+1 = 6 \). We find
that \( k = 1 \), so that \( l = n - k = 5 \) and thus \( t = 6 \). Then our ratio is 6:25 so that our parallel algorithm for this example executes the loop approximately 4 times faster than the sequential execution.

4.7 Case 7

4.7.1 Parameters

Let \( b > a \) and \( c > 0 \).

4.7.2 Logic

Given these parameters, we know that the right side of the diophantine equation is always greater than the left side and increases more rapidly. Let \( g = \gcd(a, b) \). A solution set to the diophantine equation is \( x_0 = u \cdot c/g \) and \( y_0 = v \cdot c/g \) where \( u \) and \( v \) are derived by the equation \( g = au - bv \).

With these parameters, we have a dependence direction of type \(<\). If we let \( y_1 = (T - y_0) \cdot g/a \), we can successfully find the lower bound for the derived parametric equations. If \( y_1 \in \mathbb{Z} \), then \( lbw = y_1 \). If \( y_1 \in \mathbb{R} \), then \( lbw = y_1 + 1 \). The \( lbw \) now indicates the first \( t \) of the parametric equations that results in a collision. This first collision is then calculated at \( x_{index} = x_0 + b/g \cdot lbw \) and \( y_{index} = y_0 + a/g \cdot lbw \). Hence, we split the loop between \( x_{index} - 1 \) and \( x_{index} \). This split will insure that each member of the collision resides in a separate parallel block. As in the above sections, we could increment \( lbw \) by 1, recalculate \( x_{index} \) and \( y_{index} \) and again split the loop between
$xindex - 1$ and $xindex$. However, since the right side of the diophantine equation is increasing more rapidly than the left side it is not necessary to split the loop for every increment of the $lwb$. Instead, we let $oldxindex = xindex$ and compare $oldxindex$ with $yindex$. If $yindex \geq oldxindex$ we split the loop between $xindex - 1$ and $xindex$. Otherwise, we just increment $lwb$ and recalculate $xindex$ and $yindex$. Continuing in this manner until $yindex > U$, we have optimally parallelized the loop.

4.7.3 Algorithm

/* Case 7 and Case 8 
   $b > a$ and $c > 0$ or $a > b$ and $c < 0$
   global variables:
   int $a, b, a_0, b_0, T, U$
*/
main()
{
   int $x_0, y_0, m, lwb, flag,$
   int $index_1, index_2, oldindex$
   float $z_1, m_1,$
   $x_0 = u * (b_0 - a_0)/g$
   $y_0 = v * (b_0 - a_0)/g$
   if ($b > a$)
      $z_1 = (T - y_0) * g/a$
   else
      $z_1 = (T - x_0) * g/b$
   $m = (int)z_1$
\[ m_1 = (\text{float})m \]

\[ \text{if } (z_1 == m_1) \]
\[ lw = z_1 \]

\[ \text{else} \]
\[ lw = z_1 + 1 \]

\[ \text{if } (b > a) \{ \]
\[ \text{index}_1 = x_0 + b/g * lw \]
\[ \text{index}_2 = y_0 + a/g * lw \]
\[ \} \]

\[ \text{else} \{ \]
\[ \text{index}_1 = y_0 + a/g * lw \]
\[ \text{index}_2 = x_0 + b/g * lw \]
\[ \} \]

\[ \text{flag} = 1 \]

\[ A = t \]

\[ \text{while } (\text{index}_1 \leq U) \{ \]
\[ \text{if } (\text{flag}) \{ \]
\[ \text{printdoall } (A, \text{index}_1 - 1) \]
\[ \text{oldindex} = \text{index}_1 \]
\[ A = \text{index}_1 \]
\[ \} \]

\[ \text{flag} = 1 \]

\[ lw = ++ \]

\[ \text{if } (b > a) \{ \]
\[ \text{index}_1 = x_0 + b/g * lw \]
\[ \text{index}_2 = y_0 + a/g * lw \]
\[ \} \]

\[ \text{else} \{ \]
\[ \text{index}_1 = y_0 + a/g \times lwb \]
\[ \text{index}_2 = x_0 + b/g \times lwb \]

\)
\[ \text{if} \ (\text{index}_2 \geq \text{oldindex}) \]
\[ \text{continue} \]
\[ \text{else} \]
\[ \text{flag} = 0 \]
\]
\[ \text{printdoall} \ (A, U) \]

4.7.4 Example

Consider the following loop:

\[ \text{DO I = 1, 20} \]
\[ S : A(3 \times I + 2) = \cdots \]
\[ ; \]
\[ S' : \cdots = A(5 \times I + 6) \]
\[ \text{END DO} \]

Our diophantine equation is \(3x + 2 = 5y + 6\) with \(a = 3, b = 5,\) and \(c = 4.\) Let \(g = \gcd(a, b) = 1,\) then derive \(g = au - bv\) for \(u\) and \(v.\) A solution set to the diophantine equation is \(x_0 = u \times c/g = 8\) and \(y_0 = v \times c/g = 4.\) To find the lower bound or \(lwb,\) we let \(y_1 = (T - y_0) \times g/a = -1.\) Since \(|y_1| \in \mathbb{Z}, lwb = y_1 = -1.\) Our first collision is at \(\text{xindex} = x_0 + b/g \times lwb = 3\) and \(\text{yindex} = y_0 + a/g \times lwb = 1.\) So our first loop split will be between \(\text{xindex} - 1 = 2\) and \(\text{xindex} = 3.\) Our algorithm for this example prints the
following set of sequential parallel blocks:

**DOALL I = 1, 2**

\[ S : A(3 \cdot I + 2) = \cdots \]

\[ S' : \cdots = A(5 \cdot I + 6) \]

**END DOALL**

**DOALL I = 3, 7**

\[ S : A(3 \cdot I + 2) = \cdots \]

\[ S' : \cdots = A(5 \cdot I + 6) \]

**END DOALL**

**DOALL I = 8, 17**

\[ S : A(3 \cdot I + 2) = \cdots \]

\[ S' : \cdots = A(5 \cdot I + 6) \]

**END DOALL**

**DOALL I = 18, 20**

\[ S : A(3 \cdot I + 2) = \cdots \]

\[ S' : \cdots = A(5 \cdot I + 6) \]

**END DOALL**

### 4.7.5 Time Factor

This case is similar to section 4.6.5 except now with \( c > 0 \), the right side of the diophantine equation is increasing even more rapidly than the left
side. So again, our algorithm checks if \( y_{\text{index}} < \text{oldx}_{\text{index}} \) to determine if we need to split the loop for the respective collision. Thus our \( k \) will now be incremented each time \( y_{\text{index}} < \text{oldx}_{\text{index}} \). For this example, our diophantine equation is \( 3x + 2 = 5y + 6 \). Our \( g = \gcd(a, b) = 1 \). We rewrite the diophantine equation such that \( 3x - 5y = 4 \). Since \( g = au - bv \), our \( u = 2 \) and \( v = 1 \). Then \( x_t = 8 - 5t \) and \( y_t = 4 - 3t \). Now our inequality is \( 1 \leq 4 - 3t < 8 - 5t \leq 20 \). Then \( n = U' - T' + 1 = 4 \). We find that \( k = 1 \), so that \( l = n - k = 3 \) and thus \( t = 4 \). Then our ratio is \( 4:20 \) so that our parallel algorithm for this example executes the loop 5 times faster than the sequential execution.

4.8 Case 8

4.8.1 Parameters

Let \( a > b \) and \( c < 0 \).

4.8.2 Logic

This case is very similar to section 4.7 except that here the left side of the diophantine equation is always greater and increases more rapidly than the right side. Let \( g = \gcd(a, b) \). Again, a solution set to the diophantine equation is \( x_0 = u*c/g \) and \( y_0 = v*c/g \). With the given parameters, we have a dependence direction of type \( > \). So in this case we let \( x_1 = (T - x_0) * g/b \) to locate the lower bound. If \( x_1 \in \mathbb{Z} \), then \( l_{\text{wb}} = x_1 \). If \( x_1 \in \mathbb{R} \), then \( l_{\text{wb}} = \).
\( x_1 + 1 \). Now our first collision is calculated at \( x_{\text{index}} = x_0 + b/g \times lwb \) and \( y_{\text{index}} = y_0 + a/g \times lwb \). Therefore, we split the loop between \( y_{\text{index}} - 1 \) and \( y_{\text{index}} \). Again, this split guarantees that each member of the first collision resides in a different parallel block. To continue, we could increment \( lwb \) by 1, recalculate the \( x_{\text{index}} \) and \( y_{\text{index}} \) and again split the loop between \( y_{\text{index}} - 1 \) and \( y_{\text{index}} \). But given these parameters we know that the left side of the diophantine equation is increasing more rapidly then the right side and therefore it is not necessary to split the loop after every increment of \( lwb \). Instead, set \( \text{oldy}_{\text{index}} = y_{\text{index}} \) and compare \( \text{oldy}_{\text{index}} \) to \( x_{\text{index}} \).

If \( x_{\text{index}} \geq \text{oldy}_{\text{index}} \) we split the loop between \( y_{\text{index}} - 1 \) and \( y_{\text{index}} \). If \( x_{\text{index}} < \text{oldy}_{\text{index}} \), simply increment \( lwb \) and recalculate \( x_{\text{index}} \) and \( y_{\text{index}} \). Continuing in like manner until \( x_{\text{index}} > U \), we have optimally parallelized our loop.

### 4.8.3 Algorithm

See section 4.7.3

### 4.8.4 Example

Consider the following loop:

```plaintext
DO I = 1, 20
  S : A(7*I + 3) = ···
  ···
S' : ··· = A(3*I - 2)
```
Our diophantine equation is \[7x + 3 = 3y - 2\] with \(a = 7\), \(b = 3\), and \(c = -5\). Since \(g = \gcd(a, b) = 1\) and \(g = au - bv\), we find that \(u = 1\) and \(v = 2\). Then a solution set to our diophantine equation is \(x_0 = u * c/g = -5\) and \(y_0 = v * c/g = -10\). We next calculate \(x_1 = (T - x_0) * g/b = 2\). Since \(2 \in \mathbb{Z}\), \(lwb = 2\). Our first collision will be at \(x_{\text{index}} = x_0 + b/g * lwb = 1\) and \(y_{\text{index}} = y_0 + a/g * lwb = 4\). So our first split will be between \(y_{\text{index}} - 1 = 3\) and \(y_{\text{index}} = 4\). Our algorithm for this example prints the following set of sequential parallel blocks:

\[\text{DOALL } I = 1, 3\]
\[S : A(7 * I + 3) = \cdots\]
\[\vdots\]
\[S' : \cdots = A(3 * I - 2)\]

\text{END DOALL}

\[\text{DOALL } I = 4, 10\]
\[S : A(7 * I + 3) = \cdots\]
\[\vdots\]
\[S' : \cdots = A(3 * I - 2)\]

\text{END DOALL}

\[\text{DOALL } I = 11, 20\]
\[S : A(7 * I + 3) = \cdots\]
\[\vdots\]
\[S' : \cdots = A(3 * I - 2)\]

\text{END DOALL}
4.8.5 Time Factor

This case is similar to section 4.5.5 except now with \( c < 0 \) the left side of the diophantine equation is increasing even more rapidly than the right side. Thus our algorithm checks if \( x_{\text{index}} < y_{\text{oldindex}} \) to determine if we need to split the loop for the respective collision. So our \( k \) will now be incremented each time \( x_{\text{index}} < y_{\text{oldindex}} \). For this example, our diophantine equation is \( 7x + 3 = 3y - 2 \). Our \( g = \gcd(a, b) = 1 \). We rewrite the diophantine equation such that \( 7x - 3y = -5 \). Since \( g = au - bv \), our \( u = 1 \) and \( v = 2 \). Then \( x_t = -5 - 3t \) and \( y_t = -10 - 7t \). Now our inequality is \( 1 \leq -5 - 3t < -10 - 7t \leq 20 \) such that \( -\frac{30}{7} \leq t \leq -2 \). Since we are only interested in integer solutions, we rewrite the inequality as \( -4 \leq t \leq 2 \). Then \( n = U' - T' + 1 = 3 \). We find that \( k = 1 \), so that \( l = n - k = 2 \) and thus \( t = 3 \). Then our ratio is 3:20 so that our parallel algorithm for this example executes the loop approximately 7 times faster than the sequential execution.

4.9 Case 9

4.9.1 Parameters

Let \( a > b \) and \( c > 0 \).

4.9.2 Logic

Given these parameters, we know that the left side of the diophantine equation is increasing more rapidly than the right side, but we do not know which
side of the diophantine equation is actually greater at the beginning of the loop. For example, $c$ may be large enough so that the right side is greater. Given the diophantine equation $2x = y + 5$ and initializing the loop at a lower bound of 1, $x_{index} = 2$ and $y_{index} = 6$ or the right side of the diophantine equation is greater than the left. But since the left side is increasing more rapidly, when the index reaches 5, $x_{index} = 10$ and $y_{index} = 10$. At this point the left side and right side are equal and if we increment the index again then $x_{index} = 12$ and $y_{index} = 11$. Now the left side is greater than the right. We have experienced a direction change at index 5, with a dependence direction of type $>$ up to 5, a dependence direction of type $=$ at 5, and a dependence direction of type $<$ past 5. In order to split this loop for optimal parallelization, we need to check for the type of dependence direction and act accordingly. Let $g = \text{gcd}(a, b)$. Then a solution set to the diophantine equation is $x_0 = u*c/g$ and $y_0 = v*c/g$ where $g = au - bv$. Since dependence direction of type $<$ will dominate this loop, we let $y_1 = (T - y_0) * g/a$ and if $y_1 \in \mathbb{Z}$, the lower bound $lwb = y_1$ or if $y_1 \in \mathbb{R}$, $lwb = y_1 + 1$. To find the first collision, we let $x_{index} = x_0 + b/g * lwb$ and $y_{index} = y_0 + a/g * lwb$. Now we check for the type of dependence direction. If $x_{index} > y_{index}$ (we have a dependence direction of type $>$) we must split the loop between $x_{index} - 1$ and $x_{index}$, set $oldx_{index} = x_{index}$, increment $lwb$, and recalculate $x_{index}$ and $y_{index}$. If $x_{index} = y_{index}$ (we have a dependence direction of type $=$) we need not split the loop as we have a dependence within the loop itera-
tion, not across iterations. So we increment \( lwb \) and recalculate \( xindex \) and \( yindex \). If \( xindex < yindex \) (we have a dependence direction of type \(<\) ) we split the loop between \( yindex - 1 \) and \( yindex \), set \( oldyindex = yindex \), increment \( lwb \), and recalculate \( xindex \) and \( yindex \). We continue in this manner until \( yindex > U \). At this point, we have parallelized our loop. As with the other examples, we need not always split the loop at every increment of \( lwb \). We need only split the loop if two members of a collision fall in the same parallel block. Therefore, if the dependence direction is of type \( > \), we check if \( yindex \geq oldxindex \). If this is the case, we split the loop, otherwise we do not split. In a similar manner, if the dependence direction is of type \( < \), we check if \( xindex \geq oldyindex \) and if this is true we split the loop, otherwise we do not. Now we have optimally parallelized the loop.

4.9.3 Algorithm

```c
/* Case 9 and Case 10
   a > b, c > 0 or b > a, c < 0
  global variables:
   int a, b, a0, b0, T, U
*/
main()
{
    int x0, y0, m, lwb, k
    int index1, index2, oldindex, flag
    float z1, m1
    \( x_0 = u * (b_0 - a_0)/g \)
```
\[ y_0 = v \times (b_0 = a_0)/g \]

\[ \text{if } (a > b) \]
\[ \quad z_1 = (T - y_0) \times g/a \]

\[ \text{else} \]
\[ \quad z_1 = (T - x_0) \times g/b \]

\[ m = (\text{int})z_1 \]

\[ m_1 = (\text{float})m \]

\[ \text{if } (z_1 == m_1) \]
\[ \quad Iwb = z_1 \]

\[ \text{else} \]
\[ \quad Iwb = z_1 + 1 \]

\[ \text{if } (a > b) \{ \]
\[ \quad \text{index}_1 = x_0 + b/g \times Iwb \]
\[ \quad \text{index}_2 = y_0 + a/g \times Iwb \]
\[ \} \]

\[ \text{else} \{ \]
\[ \quad \text{index}_1 = y_0 + a/g \times Iwb \]
\[ \quad \text{index}_2 = x_0 + b/g \times Iwb \]
\[ \} \]

\[ \text{flag} = 1 \]

\[ A = T \]

\[ \text{while } (\text{index}_2 \leq U) \{ \]
\[ \quad \text{if } (\text{index}_1 > \text{index}_2) \]
\[ \quad \quad k = 1 \]
\[ \quad \text{else if } (\text{index}_1 == \text{index}_2) \]
\[ \quad \quad k = 2 \]
\[ \quad \text{else} \]
\[ \quad \quad k = 3 \]
switch(k) {
    case 1:
        if (flag) {
            printdoall (A, index1 - 1)
            oldindex = index1
            A = index1
        }
        flag = 1
        lwb ++
        if (a > b) {
            index1 = x0 + b/g * lwb
            index2 = y0 + a/g * lwb
        } else {
            index1 = y0 + a/g * lwb
            index2 = x0 + b/g * lwb
        }
        if (index2 ≥ oldindex)
            continue
        else
            flag = 0
            break
    case 2:
        flag = 1
        lwb ++
        if (a > b) {
            index1 = x0 + b/g * lwb
            index2 = y0 + a/g * lwb
else {
    index₁ = y₀ + a/g * lwb
    index₂ = x₀ + b/g * lwb
}
break

// case 3:
if (flag) {
    printdoall (A, index₂ - 1)
    oldindex = index₂
    A = index₂
}
flag = 1
lwb++
if (a > b) {
    index₁ = x₀ + b/g * lwb
    index₂ = y₀ + a/g * lwb
}
else {
    index₁ = y₀ + a/g * lwb
    index₂ = x₀ + b/g * lwb
}
if (index₁ ≥ oldindex)
    continue
else
    flag = 0
break
}
4.9.4 Example

Consider the following loop:

\begin{verbatim}
DO I = 1, 20
    S : A(6*I - 5) = \ldots
    ...
S' : \ldots = A(5*I + 2)
END DO
\end{verbatim}

Our diophantine equation is $6x - 5 = 5y + 2$ with $a = 6$, $b = 5$, and $c = 7$. Let $g = \gcd(a, b) = 1$ and since $g = au - bv$, $u = 1$ and $v = 1$. A solution set to the diophantine equation is $x_0 = u*c/g = 7$ and $y_0 = v*c/g = 7$.

Now we let $y_1 = (T - y_0)*g/a = -1$. Since $y_1 \in \mathbb{Z}$, $lwb = -1$. Our first collision is at $x_{index} = x_0 + b/g*lwb = 2$ and $y_{index} = y_0 + a/g*lwb = 1$.

Since $(x_{index} = 2) > (y_{index} = 1)$ we have a dependence direction of type $>$ and split the loop between $x_{index} - 1 = 1$ and $x_{index} = 2$. Now we let $oldx_{index} = x_{index} = 2$, increment $lwb$ so that $lwb = 0$ and recalculate $x_{index} = x_0 + b/g*lwb = 7$ and $y_{index} = y_0 + a/g*lwb = 7$. Since $x_{index} = y_{index}$ we need not split the loop as the dependence direction is now of type $=$ and the dependence falls within the loop iteration and not across it. Next we increment $lwb$ such that $lwb = 1$ and let $x_{index} = x_0 + b/g*lwb = 12$ and
\( yuindex = y_0 + a/g * lwb = 13 \). Now we check for our dependence direction type and since \((xindex = 12) < (yindex = 13)\) we have type \(<\). Therefore, we split the loop between \( yindex - 1 = 12 \) and \( yindex = 13 \). We continue in this manner until our \( yindex > U \). Our algorithm for this example prints the following set of sequential parallel blocks:

\[
\text{DOALL } I = 1 \\
S : A(6 * I - 5) = \cdots \\
\vdots \\
S' : \cdots = A(5 * I + 2) \\
\text{END DOALL}
\]

\[
\text{DOALL } I = 2, 12 \\
S : A(6 * I - 5) = \cdots \\
\vdots \\
S' : \cdots = A(5 * I + 2) \\
\text{END DOALL}
\]

\[
\text{DOALL } I = 13, 18 \\
S : A(6 * I - 5) = \cdots \\
\vdots \\
S' : \cdots = A(5 * I + 2) \\
\text{END DOALL}
\]

\[
\text{DOALL } I = 19, 20 \\
S : A(6 * I - 5) = \cdots \\
\vdots \\
S' : \cdots = A(5 * I + 2) \\
\text{END DOALL}
\]
4.9.5 Time Factor

Given these parameters, section 4.9.2 explains that the left side of the diophantine equation is increasing more rapidly than the right side, but we do not know which side of the diophantine equation is actually greater at the beginning of the loop. Depending on the value of $T$, we are subject to have a dependence direction of type $<$, followed by a dependence direction of type $=$, and then a dependence direction of type $<$. If we have a dependence direction of type $=$, we need not split the loop as the collision is within a single loop iteration and does not cross over to another iteration. If a dependence direction of type $=$ does not exist, depending on the value of $T$, we may still have a cod point. To determine the number of sequential DoAll blocks, we must first find the number of collisions that occur between a given $T$ and $U$. We use the same process as explained in section 4.5.5. Once we have determined our $n$, we must next determine $k$, the number of times we need not split the loop because the members of our collision already reside in unique parallel blocks. Since all three types of dependence direction may exist given these parameters, we must check all three as our algorithm does and keep a running counter. If $xindex > yindex$ we have a dependence direction of type $>$ and increment our counter $k$ whenever $yindex < oldxindex$. If $xindex = yindex$ we have a dependence direction of type $=$ and we increment $k$ since the collision is within the loop iteration and not across iterations and consequently the loop does not have to be split. If
We have a dependence direction of type $<$ and increment $k$ whenever $x_{\text{index}} < y_{\text{index}}$. Our $k$ will now represent the number of times we do not need to split the loop since the members of each collision are already in a unique parallel block. Now that we know $n$ and $k$, we can determine $l$ and $t$ as before. In this example, our diophantine equations is $6x - 5 = 5y + 2$. Our $g = \gcd(a, b) = 1$. We rewrite the diophantine equation such that $6x - 5y = 7$. Since $g = au - bv$, our $u = 1$ and $v = 1$. Then $x_t = 7 - 5t$ and $y_t = 7 - 6t$. Now our inequality is $1 \leq 7 - 5t < 7 - 6t \leq 20$ such that $-\frac{13}{6} \leq t \leq \frac{6}{5}$. Since we are only interested in integer solutions we rewrite the inequality as $-2 \leq t \leq 1$. Then $n = U' - T' + 1 = 4$. We find that $k = 1$, so that $l = n - k = 3$ and $t = 4$. Then our ratio is 4:20 so that our parallel algorithm for this example executes the loop 5 times faster than the sequential execution.

4.10 Case 10

4.10.1 Parameters

Let $b > a$ and $c < 0$.

4.10.2 Logic

This case is similar to section 4.9 except here the right side of the diophantine equation is increasing more rapidly than the left side. But again we do not know which side of the equation is actually greater at the beginning of the
loop since $c < 0$. Therefore, for each collision we must check for the type of dependence direction that dominates. Let $g = \text{gcd}(a, b)$. A solution set to the diophantine equation is $x_0 = u \cdot c/g$ and $y_0 = v \cdot c/g$ where $g = au - bv$.

In this case, dependence direction of type $>$ will dominate the loop so we let $x_1 = (T - x_0) \cdot g/b$ and if $x_1 \in \mathbb{Z}$, the lower bound $lw = x_1$ or if $x_1 \notin \mathbb{R}$, $lw = x_1 + 1$. To find the first collision we set $x_{\text{index}} = x_0 + b/g \cdot lw$ and $y_{\text{index}} = y_0 + a/g \cdot lw$. The remaining logic for this case follows precisely from section 4.9. We check for the type of dependence direction involved and split the loop where required.

### 4.10.3 Algorithm

See section 4.9.3.

### 4.10.4 Example

Consider the following loop:

```plaintext
DO I = 1, 20
   S : A(4*I + 2) = ... 
   ...
   S' : ..., = A(5*I - 4)
END DO
```

Our diophantine equation is $4x + 2 = 5y - 4$ with $a = 4$, $b = 5$, and $c = -6$. Let $g = \text{gcd}(a, b) = 1$ and $u = -1$, $v = -1$. A solution set to the diophantine equation is $x_0 = u \cdot c/g = 6$ and $y_0 = v \cdot c/g = 6$. Now
we let $x_1 = (T - x_0) * g/b = -1$ and $lw b = -1$. Our first collision is at $x_{index} = x_0 + b/g * lw b = 1$ and $y_{index} = y_0 + a/g * lw b = 2$. Since $(y_{index} = 2) > (x_{index} = 1)$ we have a dependence direction of type < and our first loop split is between $y_{index} - 1 = 1$ and $y_{index} = 2$. Our algorithm for this example prints the following set of sequential parallel blocks:

DOALL $I = 1$

$S : A(4 * I + 2) = \cdots$

$\vdots$

$S' : \cdots = A(5 * I - 4)$

END DOALL

DOALL $I = 2, 10$

$S : A(4 * I + 2) = \cdots$

$\vdots$

$S' : \cdots = A(5 * I - 4)$

END DOALL

DOALL $I = 11, 15$

$S : A(4 * I + 2) = \cdots$

$\vdots$

$S' : \cdots = A(5 * I - 4)$

END DOALL

DOALL $I = 16, 20$

$S : A(4 * I + 2) = \cdots$

$\vdots$

$S' : \cdots = A(5 * I - 4)$

END DOALL
4.10.5 Time Factor

These parameters result in a situation similar to section 4.9.5 except here the right side of the diophantine equation increases more rapidly than the left. But again since there is a possibility of a cod point we must check all three dependence direction possibilities. We follow the same process as in section 4.9.5. In this example, our diophantine equation is $4x + 2 = 5y - 4$.

Our $g = gcd(a, b) = 1$. We rewrite the diophantine equation such that $4x - 5y = -6$. Since $g = au - bv$, our $u = -1$ and $v = -1$. Then $x_t = 6 - 5t$ and $y_t = 6 - 4t$. Our inequality is $1 \leq 6 - 4t < 6 - 5t \leq 20$ such that $-\frac{14}{5} \leq t \leq \frac{5}{4}$. Again we are only interested in integer solutions, so we rewrite the inequality as $-2 \leq t \leq 1$. Then $n = U' - T' + 1 = 4$. We find that $k = 1$, so that $l = n - k = 3$ and thus $t = 4$. Then our ratio is 4:20 so that our parallel algorithm for this example executes the loop 5 times faster than the sequential execution.
Chapter 5

Conclusion

The partialout solution for determining the reaching definitions information is a compromise between the iterative solution and interval analysis. It uses the same data-flow equations as the iterative method but eliminates the multiple passes. From interval analysis, it borrows the concept of locating the innermost region within a program, calculating the partialout for this region and then expanding out to larger and larger regions until the partialout of the outermost loop itself is found.

The algorithm is most efficient in large programs that have several nested loops involved. For example, a program with five nested loops will require a total of seven complete passes using the iterative approach, whereas the partialout approach would require somewhere in the area of less than two passes. Highest efficiency of the partialout algorithm is demonstrated in large programs with several nested loops, where the loop structures themselves are only a small part of the large program. The iterative solution forces several
passes recalculating the IN and OUT for all the blocks, whereas the partialout approach concentrates only on the loops or problem areas.

The algorithm is least efficient in programs that contain no loops or programs with a single loop, where the loop itself contains almost all the blocks of the program. Theoretically this seems correct, since the partialout solution must calculate a partialin and a partialout for each block in the loop. If this involves doing these calculations for most of the blocks in the program, then the efficiency will decrease.

Programs with no loops or programs with a single loop that involves most of the program still benefit from the partialout solution. This is because the iterative solution must make an extra pass to cross check for any changes in the calculations of the final out. The partialout solution avoids this extra pass thus eliminating needless recalculations.

Further study in this area might include a similar algorithm for the definitions of available expressions. Available expressions information is a valuable tool used by most optimizing compilers to determine global common subexpression elimination. Most compilers use an iterative solution to this problem similar to the iterative solution for reaching definitions. We believe an algorithm could be designed for the available expressions that would eliminate the multiple passes by calculating a partialout for each loop structure.

As more and more massively parallel machines appear on the market, the need for compilers that can parallelize sequential code will greatly be
enhanced. Since the greatest gains during execution time comes from the ability to parallelize statements within loop structures, this is where our research focused. Specifically, we explored statements within loop structures that access array elements. Presently, if a dependence exists between two statements which use array-indexing in a loop, our current compilers do not parallelize the loop but instead execute the code in a purely sequential manner. When executing code sequentially, all but one processor remain idle. Execution time can be decreased if it is possible to parallelize even small areas of a loop. We can investigate where a dependence will cause the statements involved to access the same memory location (what we termed collisions) and then insure that these collisions are executed in different time frames or parallel blocks. Our method was to rewrite the sequential code into a series of sequential parallel or DoAll blocks. Each DoAll block represents a group of loop iterations that may be executed in parallel. Each iteration of the DoAll block is assigned to a different processor, each processor executing its iteration concurrently. There is a sequential order to the set of DoAll or parallel blocks where the first block must finish execution before the next DoAll block begins. A sequential ordering of parallel blocks will permit us to insure that no two members of a collision reside in the same parallel or DoAll block.

Current available data dependence tests can define the type of dependence between two statements of a loop and the dependence direction and distance
of this dependence. If the dependence direction is of type $=$, we know that the dependence lies within a single loop iteration. A dependence of this type can be automatically parallelized within a single DoAll block which embraces the entire loop structure. Each iteration of the loop is assigned to a different processor and these processors execute concurrently. Tiny, the restructuring tool we used in this research, is capable of determining this kind of dependence and restructures the loop into a single DoAll block. If the dependence direction is of type $>$ or of type $<$, we have what is called a loop-carried dependence. These two types of dependence direction causes a dependence from one loop iteration back to a previous loop iteration or a dependence from one iteration to a future loop iteration respectively. With these types of dependence direction, we cannot parallelize the entire loop as we must insure that the memory accesses are executed in the proper order.

For example, consider the following simple loop structure:

```plaintext
DO I = 1, 2
    S: A(I) = ... 
    :
    S': ... = A(2*I)
END DO
```

If we were to parallelize this loop with a single DoAll block, each iteration of the loop would be executed by a different processor concurrently. But statement $S'$ of loop iteration $I = 1$ accesses the same memory location $A(2)$ as statement $S$ of iteration $I = 2$. To maintain the integrity of the
original program, loop iteration \( I = 1 \) must be executed before loop iteration \( I = 2 \). If these two loop iterations are not executed sequentially, there is a possibility that statement \( S \) of loop iteration \( I = 1 \) will assign an incorrect value of \( A(2) \).

Consequently, loops which have statements with these types of dependence direction must be further investigated in our attempt to parallelize them. We make use of the diophantine equation \( ax + a_0 = by + b_0 \) resulting from two statements \( S \) and \( S' \), both of which use array-indexing. For example, given statement \( S : A(3*I+4) = \cdots \) and statement \( S' : \cdots = A(4*I-2) \), the diophantine equation would be \( 3x + 4 = 4y - 1 \) with \( a = 3, b = 4, \) and \( c = b_0 - a_0 = -6 \). By investigating the coefficients \( a \) and \( b \) and the constant \( c \), we are able to parallelize these loops using the optimal number of processors given the restraints of the dependencies. Our parallel algorithms split the original loop into a series of smaller loops each of which has no dependencies. The iterations within each of these smaller loops are executed in parallel whereas the smaller loops themselves are executed sequentially.

For the trivial case, where \( a = b \) and \( c = 0 \), we have a loop-independent dependence or a dependence within each loop iteration but not across loop iterations. Therefore, loops which carry these parameters may be restructured into a single \textbf{DoAll} block and each iteration of the loop is executed by a separate processor in parallel. If \( a = b \) and \( c > 0 \) or \( c < 0 \), then the value of \( c/g \) where \( g = \text{gcd}(a, b) \) determines where the loop must be split to
insure no two members of a collision reside in the same loop or DoAll block. If $a > 0$, $b < 0$ or $a < 0$, $b > 0$ then we need to split the loop only once at the change of direction point. Since one side of the diophantine equation is increasing while the other side is decreasing, the change of direction point signals when the side increasing is now larger than the side that is decreasing. By splitting the loop at the change of direction point, we are assured that no two members of a collision reside in the same DoAll or parallel block. If $a > b$ and $c < 0$, then we always have a dependence direction of type $>$ with the left side of the diophantine equation increasing more rapidly than the right. With the use of the parametric equations $x_t$ and $y_t$ as stated in section 3.1, we can determine where the collisions occur and consequently split the loop such that no two members of the collision reside in the same DoAll block. For parameters $b > a$ and $c > 0$, the reasoning is the same as above, except now the left side of the diophantine equation is increasing more rapidly than the right side. For the parameters $a > b$ and $c > 0$ or $b > a$ and $c < 0$, any one of the three types of dependence direction is a possibility. Consequently, we must first check the type of dependence direction that holds for each collision and then split the loop accordingly. The algorithms presented in Chapter 4 guarantee that no two members of a collision reside in the same DoAll or parallel block. The algorithms also use the optimal number of processors possible. If the algorithm needs more processors than are available on a given machine, then we can use a virtual memory mode
where each available processor can execute multiple iterations of the loop until all iterations of the respective DoAll or parallel block are complete.

This research concentrated on single-dimension arrays and a data dependence between only two statements $S$ and $S'$ of a loop. Further research in this area could investigate multi-dimension arrays and data dependence between more than two statements. With multi-dimension arrays, each dimension could be investigated for collision areas, and then an intersection of these collision areas would designate the actual collisions of the arrays. The DoAll or parallel blocks could be formulated for these collisions insuring no two members of a collision reside in the same parallel block. If any one dimension showed no data dependence between statements, then there could be no collisions and the loop could be restructured into a single DoAll or parallel block. If a data dependence exists between several statements of a loop, parallelization of the loop would become a more complex procedure. Each data dependence would have to be investigated separately for parallelism, then the parallel or DoAll blocks of each dependence would have to be matched for cross collisions and the loop splitting executed accordingly.
Bibliography


