This is to certify that the

thesis entitled

The Parallelization of Vectorizable Programs

presented by

Paul Dennis Hovland

has been accepted towards fulfillment
of the requirements for

Master of Science degree in Computer Science

Lionel M. Ni

Major professor

Date January 29, 1993

The Parallelization of Vectorizable Programs

By

Paul Dennis Hovland

A Thesis

Submitted to
Michigan State University
in partial fulfillment of the requirements
for the degree of

Master of Science

Department of Computer Science

December 2, 1992

ABSTRACT

The Parallelization of Vectorizable Programs

By

Paul Dennis Hovland

Many of today's scientific applications are so computationally demanding that they require the use of the most powerful supercomputers available. In the past, this class of applications was solved on vector computers. Today, parallel computers, especially distributed-memory machines, offer better performance. This introduces the question of how to parallelize codes originally targeted for vector architectures. For distributed-memory parallel architectures, the parallelization of programs that vectorize well requires that we find a good way to distribute data among the processors, so that we may exploit global parallelism.

A formal technique utilizing augmented data access descriptors (ADADs) to determine this distribution is presented. This technique differs from previous approaches in that it views the problem of finding a good distribution as an extension of data dependence analysis. The application of this technique to the task of parallelizing

the highly vectorizable programs generated by ADIFOR, an automatic differentiation tool, demonstrates its utility.

ACKNOWLEDGEMENTS

I would like to thank my thesis advisor, Dr. Lionel Ni, and my entire thesis committee for their help and insight as I put this thesis together. Thanks as well to Chris Bischof and Andreas Griewank, for introducing me to the world of automatic differentiation and ADIFOR. Finally, thanks to my friends and family for the love and support they have provided over the years.

TABLE OF CONTENTS

LIST OF FIGURES

CHAPTER 1

Introduction

Since the advent of powerful vector supercomputers such as the Cray and Convex, scientists have expended a great deal of energy creating code that vectorizes well. Furthermore, computer scientists have developed tools that can take ordinary code, perform some analysis, and produce new code that vectorizes well [3]. Such code is characterized by many small loops that traverse an (one or multi-dimensional) array in a single direction.

1.1 Motivation

Today, another type of supercomputer has entered the picture—parallel computers. These machines are characterized by several, and often hundreds of, processors capable of operating more or less autonomously. It is highly desirable that the code that executed well on the vector supercomputers of the past would execute as well on the supercomputers of today with little or no modification. Since each iteration of the small loops mentioned earlier may be executed in parallel (this is, in effect, what

happens in a vector processor), one would expect vector code to parallelize easily, and with good performance. However, this is not always the case. The reason for this is in large part due to the hardware used to execute parallel programs. There is a start-up latency associated with executing several instructions in parallel. In addition, the data to be used for these parallel operations must be distributed to the processor where the computation will take place, the result of which must be stored in its destination.

On a shared memory parallel supercomputer, a variable is retrieved from either the local cache or main memory. In either case, the access time is essentially independent of which processor is performing the computation. However, in a distributed memory parallel supercomputer, each processor has associated with it its own local memory. Accessing variables stored in local memory requires significantly less time than accessing variables located in the memory associated with some other processor, as the data must be communicated between the processors via some sort of network. Because it is more scalable, the predominant paradigm in massively parallel systems is distributed memory. So, if we wish to achieve good performance on a modern supercomputer, we must try to minimize the communication of data from one processor to another. As an example of a program that is well-suited for execution on a vector supercomputer, but exhibits poor performance on a parallel supercomputer, especially one with distributed memory, consider the following code segment:

```
DOALL I=2, 100
   A(I) = B(I) + B(I-1)
ENDDO
```

This loop corresponds to the vector operation:

```
A(2:100) = B(2:100) + B(1:99)
```

and could be executed at very low cost on a vector computer. This program is said to exhibit fine granularity; i.e., the amount of work performed within the parallel loop is very small. This situation presents a problem for all parallel supercomputers, because these machines must resynchronize after each parallel loop. If only a few synchronizations need to be done, this cost is negligible, but if it is done frequently, as in fine-grained programs, the additional cost can be substantial, and performance poor. In addition to synchronization costs, on a distributed memory supercomputer, execution of this code would require a substantial amount of communication. If `A(k)` and `B(k)` are stored on processor $k, k = 1, \cdots, 100$, then in order for us to perform this computation in parallel, we must first communicate the value of `B(k-1)` from processor $k-1$ to processor k. As a consequence of the high costs associated with entering and leaving a parallel loop, in order for code to execute efficiently on a parallel machine, it must have a high ratio of work performed inside parallel loops to number of parallel loops, or coarse granularity. Also, on a distributed memory machine, communication of data from one processor to another must be kept to a minimum.

One approach to attaining the goal of coarse granularity is through the use of "global parallelism." In this paradigm, the entire program executes as parallel tasks on the several processors. If data computed by one processor is required by one or several other processors, it must be communicated to this (these) processor(s). It is extremely important that we minimize the amount of communication that is

performed. This problem reduces to deciding where data should be stored (that is, where it is computed) among the various processors.

We address this problem, proposing the augmented data access descriptor as a means to determine the best distribution for arrays on a distributed memory parallel computer. Li and Chen have proven that the general problem of data distribution on distributed memory machines is NP-complete [8]. However, the method presented provides a good heuristic, in that it is often able to identify a communication free distribution, if one exists. As presented, the algorithm applies to all programs. However, case studies will focus on vectorizable code, in particular, the code generated by ADIFOR, a Fortran source-to-source translator that produces code for derivative computation [2].

1.2 ADIFOR

In addition to existing programs intended for execution on vector computers, some code that is generated automatically may lend itself easily to vectorization, but not necessarily to parallelization. The code generated by ADIFOR, a tool for the automatic differentiation of functions, falls in this category. In order to facilitate our discussion of the parallelization of these programs, a brief explanation of automatic differentiation and ADIFOR is in order.

Traditionally, there have been three approaches to obtaining the derivatives of functions in a computer program. One approach is to manually code the derivative of the function. This task can be exceedingly complicated and tedious, and is prone to

errors, particularly if the function is very complex. Alternatively, one can use finite differences. This approach is the simplest, but it is also the least accurate and can produce large errors if a poor step size is selected. A third option is to use a symbolic manipulator, such as Maple [4]. This is an excellent choice for many functions, but has some drawbacks of it's own. First, most manipulators are unable to handle extremely complicated functions. Second, some redundant computations will be performed if common subexpressions are not eliminated.

An alternative to all of these techniques is automatic differentiation. Automatic differentiation relies upon the fact that every function, no matter how complicated, is simply the composition of a small set of elementary functions, such as multiplication, sine, and square root. The mathematical notion of a composition corresponds directly to a program statement that performs some mathematical operation on the results of previous computations. Thus, the code segment

```
X2 = X*X
F = SIN(X2)
```

corresponds mathematically to the composition of sine and multiplication, i.e. $f = sin(x^2)$. As a consequence of this correlation, we may apply the chain rule

$$\frac{\partial}{\partial t} f(g(t))\Big|_{t=t_0} = \left(\frac{\partial}{\partial s} f(s)\Big|_{s=g(t_0)}\right)\left(\frac{\partial}{\partial t} g(t)\Big|_{t=t_0}\right)$$

repeatedly to the composition of the elementary operations, allowing us to calculate the derivatives of a function computed by an elaborate program accurately and in a completely mechanical fashion. This procedure is referred to as *automatic differenti-*

ation [6].

ADIFOR transforms Fortran 77 programs using this approach. As an example, suppose we have a Fortran subroutine that computes the value of $f = \prod_{i=1}^{5} x(i)$. The code sequence might be written as:

```
subroutine prod5(x,f)
real x(5), f
f = x(1) * x(2) * x(3) * x(4) * x(5)
return
end
```

Then, if we wish to compute the derivatives of f with respect to $x_i, i = 1, \cdots, 5$ and store the results in array g$f(), we might apply our knowledge of calculus and produce:

```
f = x(1) * x(2) * x(3) * x(4) * x(5)
g$f(1) = x(2) * x(3) * x(4) * x(5)
g$f(2) = x(1) * x(3) * x(4) * x(5)
g$f(3) = x(1) * x(2) * x(4) * x(5)
g$f(4) = x(1) * x(2) * x(3) * x(5)
g$f(5) = x(1) * x(2) * x(3) * x(4)
```

where we use g$f(i) to designate $\frac{\partial f}{\partial x_i}$. If instead we apply ADIFOR to this subroutine, ADIFOR produces:

```
r$1 = x(1) * x(2)
r$2 = r$1 * x(3)
r$3 = r$2 * x(4)
r$4 = x(5) * x(4)
r$5 = r$4 * x(3)
r$1bar = r$5 * x(2)
r$2bar = r$5 * x(1)
r$3bar = r$4 * r$1
r$4bar = x(5) * r$2
```

```
do g$i$ = 1, g$p$
  g$f(g$i$) = r$1bar*g$x(g$i$,1) + r$2bar*g$x(g$i$,2)
              + r$3bar*g$x(g$i$,3) + r$4bar*g$x(g$i$,4)
              + r$3*g$x(g$i$, 5)
end do
f = r$3 * x(5)
```

Variables introduced by ADIFOR contain the $ sign, and continuation line characters have been deleted to improve readability. The array variable g$x corresponds mathematically to $\frac{\partial x}{\partial x}$. From calculus, we know that

$$\frac{\partial x_i}{\partial x_j} = \begin{cases} 0 & \text{if } i \neq j \\ 1 & \text{if } i = j \end{cases}$$

Thus, g$x should be initialized to the 5×5 identity matrix. The variable gp corresponds to the number of columns in this array and should be set equal to 5. If this initialization is performed, the code above is equivalent to:

```
r$1 = x(1) * x(2)
r$2 = r$1 * x(3)          = x(1) * x(2) * x(3)
r$3 = r$2 * x(4)          = x(1) * x(2) * x(3) * x(4)
r$4 = x(5) * x(4)
r$5 = r$4 * x(3)          = x(5) * x(4) * x(3)
r$1bar = r$5 * x(2)       = x(2) * x(3) * x(4) * x(5)
r$2bar = r$5 * x(1)       = x(1) * x(3) * x(4) * x(5)
r$3bar = r$4 * r$1        = x(1) * x(2) * x(4) * x(5)
r$4bar = x(5) * r$2       = x(1) * x(2) * x(3) * x(5)

g$f(1) = r$1bar           = x(2) * x(3) * x(4) * x(5)
g$f(2) = r$2bar           = x(1) * x(3) * x(4) * x(5)
g$f(3) = r$3bar           = x(1) * x(2) * x(4) * x(5)
g$f(4) = r$4bar           = x(1) * x(2) * x(3) * x(5)
```

```
g$f(5) = r$3                    = x(1) * x(2) * x(3) * x(4)

f = r$3 * x(5)
```

Comparing this code with the program derived using calculus reveals that the vector g$f does in fact contain $\frac{\partial f(x)}{\partial x}$. In addition, the ADIFOR-generated program computes the product in a binary-tree like fashion, allowing intermediate results to be utilized in the computation of the gradient. As a consequence, no redundant subexpressions are computed, and the ADIFOR-generated code may require fewer floating point operations than the code that we generated by hand.

What is most important about ADIFOR-generated code for our purposes is that it contains many loops of length gp![1] If gp (which, memory permitting, is equal to the number of elements in the independent variables) is large, this code vectorizes very well. In addition, the vectors under consideration are always the same—the first index of the gradient arrays is the only one that varies.

Since arrays are always accessed along the same dimension, it would appear that distributing a different row (or set of rows, if there are more rows than processors) to each processor would minimize the amount of communication necessary. This conclusion is correct, but it does not take into consideration the need to communicate the scalar values computed in between the vector loops. We can eliminate this problem by using scalar promotion so that scalar computations are duplicated on all processors, but this approach increases the total amount of work being performed. Thus, we cannot achieve linear speedup, that is, execution on N processors requires

[1]For a more complete discussion of the code and how it works, see [2].

$\frac{1}{N}$ execution time on a single processor, our ultimate goal. Alternatively, we can try to take advantage of any parallelism inherent in the original code, which ADIFOR is guaranteed to preserve.

1.3 Problem Statement

In Section 5.1, some mathematical analysis using estimates for various properties of ADIFOR-generated code is performed. This analysis provides some very general rules for parallelizing this code. These rules also apply to other vectorizable code. However, the guidelines established are only useful if the programmer is able to find the best distribution for the original program. If the original program is exceedingly complex, this may be a formidable task if the best distribution must be determined by the programmer. Instead, we would like to determine the best distribution of data in an automatic fashion. Toward this aim, a formal technique utilizing augmented data access descriptors to compute a close approximation to this distribution is presented. Empirical results from execution on a Butterfly TC2000 reveal that conclusions reached by the proposed method are reasonable.

1.4 Organization

The organization of this thesis is as follows. Chapter 2 provides a brief explanation of four subjects that are the underpinnings of augmented data access descriptors– data alignment, data dependence analysis, Data Access Descriptors, and the special properties of vectorizable code. The following chapter presents the augmented data

access descriptor, specifying its format, computation, meaning, and application to modular programs. Chapter 4 describes the effects of various loop transformations–promotion, loop unrolling, loop fusion, strip mining, and invariant code movement–on augmented data access descriptors. In addition, an example of how goal-directed loop transformations can be used to eliminate the need for data communication in a program is presented. The final chapter examines the parallelization of ADIFOR-generated code, and provides empirical results to support the conclusions reached. The nature of augmented data access descriptors and how they should be applied to vectorizable code is reviewed, and topics requiring further study are presented.

CHAPTER 2

Foundations

This chapter serves as a brief introduction to data alignment, data dependence analysis, Data Access Descriptors, and the special nature of vectorizable code. This background information is necessary for a complete understanding of augmented data access descriptors, and why they are useful.

2.1 Data Alignment

In order to aid parallelizing compilers in the task of code generation for distributed memory computers (or any computer with non-uniform memory access time), several languages have been developed that enable the programmer to include information concerning the way in which data should be distributed among the various processors [5, 12]. If the proper alignment of data is chosen by the programmer, then communication costs may be minimized. Furthermore, the compiler may exploit multicast communication and other specialized communication techniques to further reduce the cost of communicating information [10]. For example, suppose we wish to multiply

an $N \times N$ matrix A by an N element vector B to produce an N element product vector, C. An ordinary Fortran 77 program to accomplish this task might look like:

```
      DOUBLE PRECISION  A(N,N), B(N), C(N)
      INTEGER OUTER, INNER

      DO OUTER=1, N
        C(OUTER) = 0
        DO INNER=1, N
          C(OUTER) = C(OUTER) + B(INNER)*A(OUTER,INNER)
        ENDDO
      ENDDO
```

The Fortran D version [5] of this program would be something like:

```
      DOUBLE PRECISION  A(N,N), B(N), C(N)
      INTEGER OUTER, INNER

c The next 5 lines are Fortran D specifications regarding
c the alignment of arrays A,B, and C.
      DECOMPOSITION X(N)
      ALIGN A(I,J) with X(I)
      ALIGN B(I) with X(*)
      ALIGN C(I) with X(I)
      DISTRIBUTE X(CYCLIC)

      PARALLEL DO OUTER=1, N
        C(OUTER) = 0
        DO INNER=1, N
          C(OUTER) = C(OUTER) + B(INNER)*A(OUTER,INNER)
        ENDDO
      ENDDO
```

The DECOMPOSITION statement indicates that an N element vector X is to be used as our frame of reference, or *template.* Each element of C aligns exactly with an element of X. In addition, one row of A and the entire vector B is aligned with each

element of X. The DISTRIBUTE statement simply indicates how the data aligned with each element of X should be distributed if there are not enough processors.[1] If we assume that the number of processors is greater than N, then each processor contains a row of A, the vector B, and an element of C. Thus, each processor may compute an element of C without needing to communicate with any of the other processors.

Unfortunately, determining the best distribution for a complicated program can be exceedingly difficult. Also, a small mistake in the alignment statements can be extremely costly. Suppose, for example, that we had produced the Fortran D program:

```
DOUBLE PRECISION  A(N,N), B(N), C(N)
INTEGER OUTER, INNER

DECOMPOSITION X(N)
ALIGN A(I,J) with X(J)
ALIGN B(I) with X(*)
ALIGN C(I) with X(I)
DISTRIBUTE X(CYCLIC)

PARALLEL DO OUTER=1, N
  C(OUTER) = 0
  DO INNER=1, N
    C(OUTER) = C(OUTER) + B(INNER)*A(OUTER,INNER)
  ENDDO
ENDDO
```

The only difference between this program and the previous example is the ALIGN statement for A. However, the change in performance would be dramatic. In order for a given processor to compute its assigned element of C, C_i, it must first get the values of $A_{i,j(i\neq j)}$ from the other processors. This communication could be very costly.

[1] For a more complete explanation of the syntax of this and other Fortran D instructions, see [5].

In order to prevent such mistakes from occurring, and to assist in the automatic generation of data alignment information, several techniques have been developed that determine a good (but not necessarily the best) data alignment in a mechanical fashion. Li and Chen model the data alignment problem as a graph problem, and present a heuristic algorithm for solving the problem [8]. The heuristic approach produces performance results that are comparable to the optimal alignment (found using brute force), and significantly better than the performance of an alignment determined using a greedy algorithm. Gupta and Banerjee use a different approach, attacking the problem from the perspective of the whole program, and solving for an optimal combination of parallelism and communication costs, subject to certain constraints [7]. Both approaches have their merit and may be better suited for many applications than the technique described in Chapter 3. However, the approach we examine has the advantage that it is very simple (and thus suitable for use in a compiler, which must process a program in a reasonable amount of time) and can treat statements, regions of programs, and entire programs in an identical manner. This flexibility is an important feature, because it enables the separate examination of sections of a program. In many cases, one alignment is appropriate for one stage of a computation, while a different alignment is appropriate for a later stage. Using both alignments can result in better performance than just using one alignment or the other for the entire program [11].

2.2 Data Dependence Analysis

Instead of treating automatic data alignment as a completely new problem, we can view data alignment as an extension of data dependence analysis. Data dependence analysis is the investigation of the manner in which the execution of one statement influences the results of another. In order for statements to be executed in parallel, it is necessary that the result of each statement be independent of the results of all other statements. Data dependence may take one of three forms: flow dependence, antidependence, or output dependence. Flow dependence indicates that a variable is defined by statement S1, then used by statement S2. Antidependence indicates that a variable is used by statement S1, then defined by statement S2. Output dependence indicates that a variable is defined by statement S1 and by statement S2. By far the most important of these is flow dependence, because anti- and output dependence may be eliminated through the use of temporary variables.

Dependence analysis is straightforward when statements involve only scalar variables; if a variable appears on the left side of an assignment statement, then a dependence exists with any other statement that references that variable. However, when arrays are referenced using an expression that may vary between iterations of a loop, it is possible for a dependence to exist between separate iterations of the loop. Such a dependence is called a *loop carried dependence.*

There are many ways to test for loop carried dependences. One method is to solve a Diophantine equation describing the array references and check if the solution lies within the bounds of the loop(s). For example, suppose state-

ment S1 defines $A(f_1(i_1, i_2, \ldots, i_{\ell_1}), \ldots, f_m(i_1, i_2, \ldots, i_{\ell_1}))$ and statement S2 uses $A(g_1(j_1, j_2, \ldots, j_{\ell_2}), \ldots, g_m(j_1, j_2, \ldots, j_{\ell_2}))$, where array A has m dimensions, S1 is nested in ℓ_1 loops and S2 is nested in ℓ_2 loops. Then a flow dependence exists if the equation

$$f_k(i_1, i_2, \ldots, i_{\ell_1}) = g_k(j_1, j_2, \ldots, j_{\ell_2}), 1 \leq k \leq m$$

has an integer solution within the loop bounds. Solving the Diophantine equation can be a computationally demanding task, especially when m is large. As a consequence, several conservative tests of data dependence have been developed. Among these are the GCD test, the Banerjee test, and the λ test [9, 13]. These tests are conservative in that they test necessary conditions for a dependence to exist. However, these conditions may not be sufficient. Thus, a test may indicate that a dependence exists when in fact it does not, but it will never indicate that a dependence does not exist when it does.

Traditionally, parallelizing compilers have relied upon data dependence analysis to determine whether particular sections of code or iterations of a loop may be executed in parallel. If a dependence exists, then the loop is not parallelized. However, in the global parallelization scheme, it is assumed that occasionally a variable will be defined by one processor and used by another. What is most important is that the number of times a dependence occurs (thereby creating a need for communication) is minimized. Thus, data alignment may be viewed as an advanced type of data dependence analysis, through which we attempt to minimize some function, which serves as an approximation to communication costs.

2.3 Data Access Descriptors

In developing the Data Access Descriptor, Balasundaram takes a slightly different approach to data dependence analysis [1]. Rather than focusing upon the dependence of one statement on another, Balasundaram examines the manner in which regions of a program influence one another. This approach allows traditional statement-to-statement data dependence analysis to be unified with interprocedural data dependence analysis, which is extremely important in modular programs.

The Data Access Descriptor contains a variety of information regarding the way in which a particular variable is accessed. An extremely important aspect of the Data Access Descriptor is the simple section, a set of hyperplanes that form convex hulls that completely enclose the regions of arrays accessed by a section of a program (which may be a single statement or the entire program). By applying intersection and union operations to these descriptors, it is possible to determine whether a data dependence exists between two regions of a program. For example, suppose we have the following section of code:

```
DO I=1, 5
  DO J=1, 5
    A(J,I) = 1.0
  ENDDO
ENDDO
DO I=1, 3
  DO J=1, 3
    A(J-I+5,I+J-1) = 1.5
  ENDDO
ENDDO
DO I=6, 10
  DO J=1,5
```

```
      A(I,J) = A(I,6-J)
    ENDDO
  ENDDO
```

which we wish to parallelize by executing the third pair of loops concurrently with the execution of the first and second pairs of loops.

Then the portion of A referenced by the first pair of DO loops (a region of the program which we will denote $R1$) can be described by:

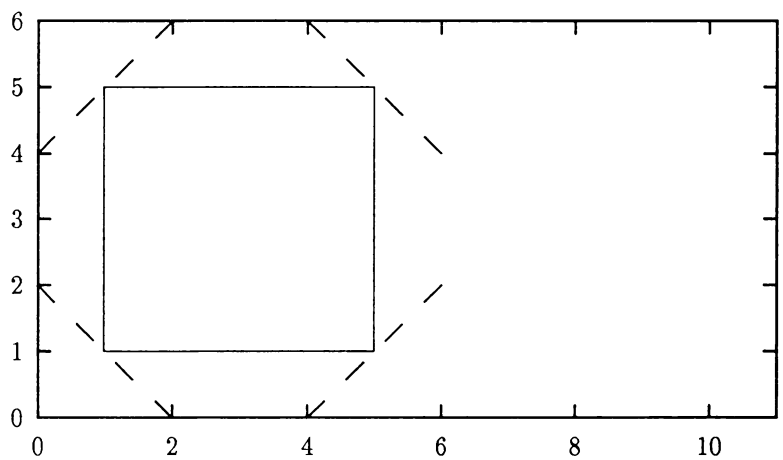
$$1 \leq x \leq 5$$

$$1 \leq y \leq 5$$

$$2 \leq x + y \leq 10$$

$$-4 \leq x - y \leq 4$$

where x and y correspond to the first and second dimensions of A, respectively. This system of equations may be represented graphically as:



where solid lines correspond to equations which actually impose bounds on the array dimensions, and dotted lines correspond to equations providing redundant informa-

tion. The second pair of DO loops, $R2$, access a portion of A described by:

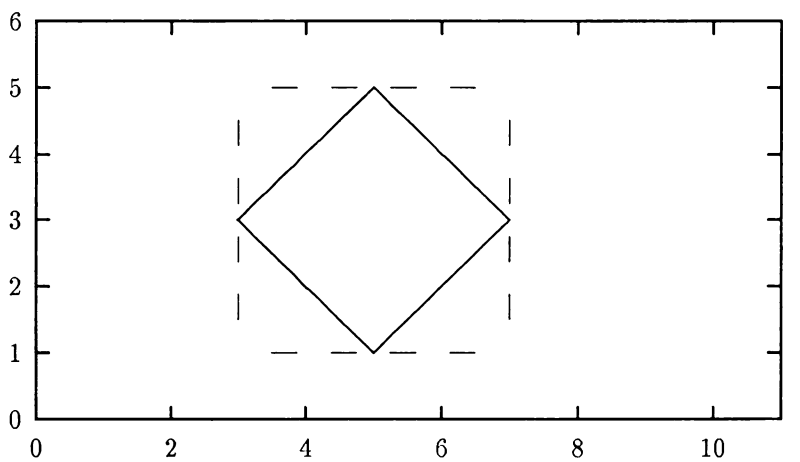
$$3 \leq x \leq 7$$

$$1 \leq y \leq 5$$

$$6 \leq x + y \leq 10$$

$$-4 \leq x - y \leq 0$$

which may be represented graphically as:



The final pair of DO loops, $R3$, reference A subject to the following constraints:

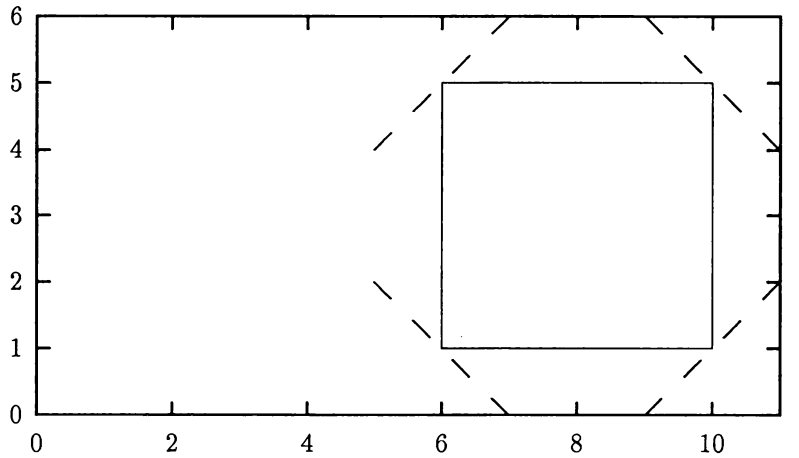
$$6 \leq x \leq 10$$

$$1 \leq y \leq 5$$

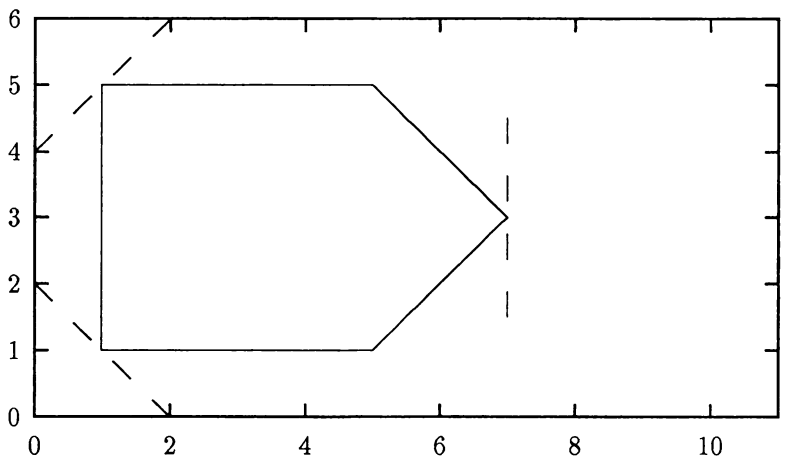
$$7 \leq x + y \leq 15$$

$$1 \leq x - y \leq 9$$

This set of equations has a graphical representation of:



The DADs describing two regions of a program can be combined using union to yield a DAD describing a larger region of the program. So, we can get the simple section describing regions $R1$ and $R2$ by taking the union of the simple sections for $R1$ and $R2$, yielding:



which corresponds to the equations:

$$1 \leq x \leq 7$$

$$1 \leq y \leq 5$$

$$2 \leq x + y \leq 10$$

$$-4 \leq x - y \leq 4$$

If we compute the intersection of this merged simple section with the simple section for $R3$, we discover that it is:

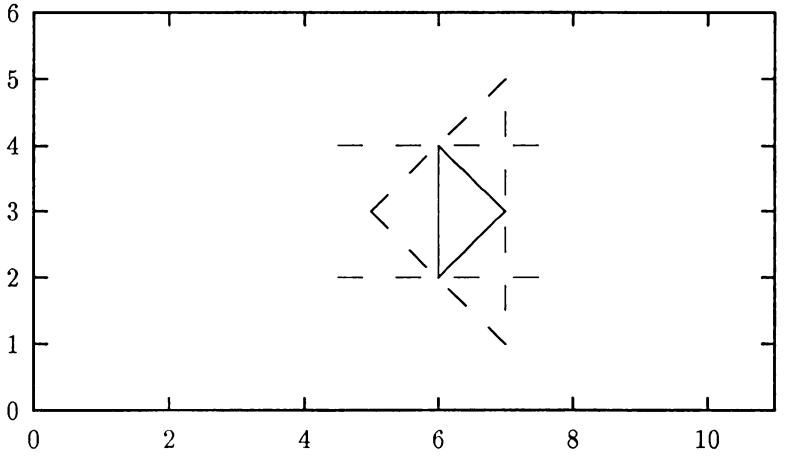
$$6 \leq x \leq 7$$

$$2 \leq y \leq 4$$

$$8 \leq x + y \leq 10$$

$$-4 \leq x - y \leq -2$$

Or, graphically,



The fact that the simple section is not empty indicates that the section of code composed of $R1$ and $R2$ accesses some of the same array elements as $R3$. Therefore, these two sections of the program may not be executed in parallel. It is also easy to determine that region $R1$ could not be executed in parallel with a section composed of $R2$ and $R3$, since $SS(R1) \cap (SS(R2) \cup SS(R3)) \neq \emptyset$.

2.4 Properties of Vectorizable Code

In addition to the standard considerations when performing data alignment analysis, code which vectorizes well has certain special properties which we may wish to keep in mind. Arrays are generally accessed along rows and columns, rather than in irregular patterns. It is these regular accesses that facilitate vectorization. They also facilitate data alignment, in that it is easier to determine how to best distribute data that is always accessed in the same, simple pattern. As was seen in Section 1.2, in the case of ADIFOR-generated code, the accesses are even more regular, as they all occur along the leading dimension of the array. In addition to regular access patterns, vectorizable code is characterized by many small loops. This presents a problem for standard parallelization techniques, because the amount of work performed inside the loops is too small to make the high overhead of executing a loop in parallel tolerable. However, when we exploit global parallelism, this fine granularity may be acceptable. We will keep these special properties in mind as we examine a mechanism for determining the proper data alignment for vectorizable programs.

CHAPTER 3

Augmented Data Access Descriptors

We now explore a mechanism for finding a good data alignment. We define a good alignment as one that leads to a distribution that is free from communication. In cases where we cannot eliminate communication, we attempt to minimize it.

3.1 Augmented Data Access Descriptors

One approach to quantifying the amount of communication overhead associated with various data distributions is to extend the data access descriptor so that it describes both the sections of an array that are accessed by a particular statement and also the manner in which they are accessed. As explained in Section 2.3, the original data access descriptor consists of a set of hyperplanes forming a convex hull within which all array accesses are guaranteed to occur. This information can be very useful, but may be expensive to compute. However, because of the restrictive nature of data

alignment languages like Fortran D and DINO [5, 12], as well as the propensity of vector codes to access arrays along the axes, it is sufficient to restrict our convex hulls to hypercubes.

We are most interested in accesses that proceed along one dimension of the array (as opposed to cases where the subscripts are coupled). Such accesses are easy to identify, because they occur when one of the subscripts of the array is a function of only one of the iteration variables; e.g., `A(3*I+7)=1.0`. These accesses are also important, because they indicate that the array may be distributed across that dimension (and parallelized along that dimension) without any need for communication.

The *augmented data access descriptor* (ADAD) for an array referenced in a particular statement will consist of a $k \times n$ array of tuples, where k is the level of nesting for the statement and n is the number of dimensions in the array, plus an integer element, whose value is set equal to k.[1] For example, in statement S1 in the following code:

```
DO I=1, 100
  DO J=2, 50
     A(I,J,I+J) = B(I,J)                    (S1)
  ENDDO
ENDDO
```

the ADAD describing `A` would contain a 2×3 array of tuples, plus the integer 2, and the ADAD describing `B` would contain a 2×2 array of tuples, plus the value 2. For a $k \times n$ tuple array, each tuple is denoted by $\mathcal{D}(A, S, I, D)$, where A is the array being

[1] Strictly speaking, this integer element is not necessary, as we may deduce k from the number of rows in the ADAD. However, it is included to facilitate the use of a statement's degree of nesting as a tie-breaker in determining alignments (the use of this heuristic for situations in which a communication-free alignment is not possible will be discussed later).

described, S is the statement with which we are concerned, I is an iteration variable, and D is a dimension of array A.

The first two members of each tuple indicate the lower and upper bounds, respectively, on the value subscript D may assume (note that this value is independent of I). The third element gives some indication of whether parallelizing the iteration variable to which that row of the ADAD corresponds will create a need for communication if the array in question is distributed across that dimension. If this element is an asterisk (*) then communication is necessary, and the fourth element in the tuple is the empty set. If the element is a positive integer, then communication is not necessary, if the array is distributed in bands of width equal to this positive integer. In this case, the fourth element of the tuple is a set of offsets indicating which rows (or columns, depending on perspective) are referenced by that statement. This information is important for combining ADADs, a procedure to be described later.

If the third element is zero, no communication is necessary, so long as this is the only reference to the array. The value zero is used as an indicator that the subscript is a nonlinear function of the particular iteration variable. Distribution is complicated, but possible, as discussed in Section 3.2.4. As in the case of an asterisk, the tuple's fourth element is the empty set.

The ADADs for the example above are:

D(A,S1) =

2	dimension 1	dimension 2	dimension 3
I	(1,100,1,{0})	(2,50,*,∅)	(3,150,*,∅)
J	(1,100,*,∅)	(2,50,1,{0})	(3,150,*,∅)

and

D(B,S1) =

2	dimension 1	dimension 2
I	(1,100,1,{0})	(2,50,*,∅)
J	(1,100,*,∅)	(2,50,1,{0})

The details of how these ADADs are computed will be discussed in the next section.

3.2 Finding a Good Alignment Using ADADs

3.2.1 Program Model

In describing ADADs and how they should be used, several assumptions are made about the programs being evaluated. One assumption is that loops have a step size of one. Such a loop is said to be normalized. Since it is possible to normalize any loop [13], this restriction is not significant. We also assume that arrays are primarily indexed by iteration variables. Constants and induction variables are also often used to index arrays. Since constants do not provide a means for distribution and induction variables can always be replaced by expressions involving iteration variables, this

assumption does not impose unnecessary restrictions. The third assumption is that those statements located in the innermost loops are executed most frequently. In general, this is a reasonable assumption, especially if loops involve more than a few iterations. However, if a statement is guarded by a conditional, and the guard is usually false, then it may be executed less frequently than a statement with a smaller degree of nesting. However, since this assumption only governs our decision as to how data should be aligned when a communication-free distribution does not exist, correctness is not affected, and it is hoped that the effect on performance is minimal.

3.2.2 Computing Augmented Data Access Descriptors

The computation of ADADs is very simple. Given a particular reference to a particular array in a particular statement, for each loop within which that statement is nested, we compute a tuple, $\mathcal{D}(A, S, I, D)$, representing the dependences of a particular subscript with respect to the iteration variable corresponding to that loop, as well as the upper and lower bounds of the subscript. The upper and lower bounds, which as mentioned earlier are the first and second elements of a tuple and which we denote $\mathcal{D}(A, S, I, D)[1]$ and $\mathcal{D}(A, S, I, D)[2]$, can often be determined in a straightforward fashion using the upper and lower bounds of the iteration variables. For example, if a subscript is a monotonic function, G, of iteration variables I and J, then the lower bound of the subscript, $\mathcal{D}(A, S, I, D)[1]$ is equal to the lower bound of the function:

$$\mathcal{D}(A, S, I, D)[1] = \min(G(L(I), L(J)), G(L(I), U(J)), G(U(I), L(J)), G(U(I), U(J)))$$

Similarly, the upper bound of the subscript, $\mathcal{D}(A,S,I,D)[2]$, is:

$$\mathcal{D}(A,S,I,D)[2] = \max(G(L(I),L(J)),G(L(I),U(J)),G(U(I),L(J)),G(U(I),U(J)))$$

In cases where G is not monotonic, or where the bounds of the iterations are not known, a conservative assumption regarding these bounds (0 for the lower bound and positive infinity (∞) for the upper bound) may be made.

The computation of the third element of the tuple may be expressed as:

$$\mathcal{D}(A,S,I,D)[3] = \begin{cases} m & \text{if D is a linear function } mI+b \text{ of the iteration variable} \\ 0 & \text{if D is a nonlinear function of the iteration variable, } I \\ & \text{otherwise} \end{cases}$$

Similarly,

$$\mathcal{D}(A,S,I,D)[4] = \begin{cases} \{b\} & \text{if D is a linear function } mI+b \text{ of the iteration variable} \\ \emptyset & \text{otherwise} \end{cases}$$

As an example, the augmented data access descriptor for statement S1 in the following code segment:

```
      DO 100 I=1,100
        DO 110 J=1,100
           A(I,J) = SQRT(I*J)                    (S1)
110     CONTINUE
100   CONTINUE
```

is:

D(A,S1) =

2	dimension 1	dimension 2
I	(1,100,1,{0})	(1,100,*,∅)
J	(1,100,*,∅)	(1,100,1,{0})

This is equivalent to saying: "Statement S1 has a nesting level of 2. With respect to iteration variable I, array A may be distributed across its first dimension, since the first subscript is a linear function $(1I + 0)$ of I, but not across its second dimension. With respect to iteration variable J, array A may be distributed across its second dimension, since the second subscript is a linear function $(1J + 0)$ of J, but not across the first dimension." This is in complete agreement with the actual syntax of S1.

3.2.3 Combining Augmented Data Access Descriptors

Combining two ADADs that describe the same array is also a rather straightforward task, as long as the convex hulls described by the the first two elements of the tuples intersect. If they do not intersect, it may be possible to avoid communication by distributing those two regions of the array in different ways. However, since the ability to do this also implies that the program could be re-written in terms of two different arrays, each of which would have its own augmented data access descriptor, we can ignore this case and assume the hulls intersect.

Definition 3.1 (Merged Augmented Data Access Descriptors)

Merged ADADs consist of an $l \times n$ array of tuples, where l is the number of iteration variables in common to the ADADs being combined, and n is the number of columns

in the original ADADs (also equal to the number of dimensions in the variable being described). As in regular ADADs, each tuple is a 4-tuple. For each iteration variable I and dimension D, we derive a new tuple according to the following procedure.

1. $\mathcal{D}(A, \{S1, S2\}, I, D)[1] = \min(\mathcal{D}(A, S1, I, D)[1], \mathcal{D}(A, S2, I, D)[1]$

2. $\mathcal{D}(A, \{S1, S2\}, I, D)[2] = \max(\mathcal{D}(A, S1, I, D)[2], \mathcal{D}(A, S2, I, D)[2]$

The reason for choosing these elements in this manner is that accesses to the array by these statements must lie within the hypercube whose bounds correspond to the maximum and minimum, respectively, of the upper and lower bounds of the hypercubes containing the accesses of the statements being merged.

3. **If** $\mathcal{D}(A, S1, I, D)[3] \neq \mathcal{D}(A, S2, I, D)[3]$ **or** $\mathcal{D}(A, S1, I, D)[3] = 0$ **or** $\mathcal{D}(A, S2, I, D)[3] = *$ **Then**

   $\mathcal{D}(A, \{S1, S2\}, I, D)[3] = *$; $\mathcal{D}(A, \{S1, S2\}, I, D)[4] = \emptyset$;

   Else

   Let $B = \mathcal{D}(A, S1, I, D)[4] \cup \mathcal{D}(A, S2, I, D)[4]$;

   If $\max(B) - \min(B) < \mathcal{D}(A, S1, I, D)[3]$ **Then**

   $\mathcal{D}(A, \{S1, S2\}, I, D)[3] = \mathcal{D}(A, S1, I, D)[3]$;

   $\mathcal{D}(A, \{S1, S2\}, I, D)[4] = B$;

   Else

   $\mathcal{D}(A, \{S1, S2\}, I, D)[3] = *$; $\mathcal{D}(A, \{S1, S2\}, I, D)[4] = \emptyset$;

   Endif

Endif

As an example, consider the following ADADs:

D(A,S1) =

3	dimension 1	dimension 2	dimension 3
I	(1,49,2,{-1})	(1,100,*,∅)	(1,100,*,∅)
J	(1,49,*,∅)	(1,100,1,{-1})	(1,100,*,∅)
K	(1,49,*,∅)	(1,100,*,∅)	(1,100,2,{0})

D(A,S2) =

3	dimension 1	dimension 2	dimension 3
I	(2,100,2,{0})	(1,200,*,∅)	(1,100,*,∅)
J	(2,100,*,∅)	(1,200,1,{0})	(1,100,*,∅)
K	(2,100,*,∅)	(1,200,*,∅)	(1,100,3,{0})

Then the merged augmented data access descriptor is:

D(A,{S1,S2}) =

3	dimension 1	dimension 2	dimension 3
I	(1,100,2,{-1,0})	(1,200,*,∅)	(1,100,*,∅)
J	(1,100,*,∅)	(1,200,*,∅)	(1,100,*,∅)
K	(1,100,*,∅)	(1,200,*,∅)	(1,100,*,∅)

Definition 3.2 (Global Augmented Data Access Descriptors) *A global augmented data access descriptor is a merged augmented data access descriptor for all*

statements in the section of a program being analyzed.

3.2.4 Using ADADs to Develop Alignment Statements

After all of the ADADs describing a particular array have been combined to form one global ADAD, we can make some decision as to what the best distribution for that array might be.

Lemma 3.1 *If the third element of the tuple corresponding to the iteration variable whose loop is to be parallelized and the dimension of the array across which we wish to distribute is a positive integer, m, then the array should be distributed in bands of width equal to m. In addition, they should be offset by an amount corresponding to the smallest member of the fourth element. If this procedure is followed, then communication will not be necessary. This may be formalized as follows:*

Given:

S – a section of code.

I – an iteration variable in section S.

n – the number of arrays accessed in section S.

$A_k(k = 1..n)$ – the arrays accessed in section S.

D_k – an index corresponding to a dimension of array A_k.

$\mathcal{D}(A, S, I, D)[j]$ – the value of the jth element of the tuple corresponding to iteration variable I and dimension D in the ADAD describing the accesses to array A in section S.

If:

$$(\exists\ I : Loop(I) \subseteq S : (\forall A_k : 1 \leq k \leq n :$$

$$(\exists D_k : 1 \leq D_k \leq Numdims(A_k) : \mathcal{D}(A_k, S, I, D_k)[3] \geq 0)))$$

Then:

The loop may be parallelized in a communication free manner.

Algorithm:

Given any code segment S for which the constraint holds, we can construct a data parallel Fortran D version that is free from communication in the following manner:

1. Change the DO loop associated with I to a parallel loop.

2. For each array A_k choose some dimension D_k satisfying the above constraint, and let $T_k = \mathcal{D}(A_k, S, I, D_k)[3]$. Also, let T_{lcm} equal the least common multiple of the nonzero T_ks.

3. Add a decomposition statement of the form

   DECOMPOSITION X(U)

   to the code, where X is an unused variable name, and

   $U = \max\{\mathcal{D}(A_k, S, I, D_k)[2] \times (T_{lcm}/T_k)\}$.

4. For each array A_k, if $T_k \neq 0$ add an alignment statement of the form

   ALIGN A_k(I,J,...) with X(*stride**M$-$(*stride*$\times$ *offset*))

   to the program, where $stride = T_{lcm}/T_k$, $offset = \max(\mathcal{D}(A_k, S, I, D_k)[4])$, and M corresponds to the placeholding index variable (I, J, etc.) representing dimension D_k. If $T_k = 0$, array A_k is referenced only once and its distribution

should be handled independently and in a manner consistent with that reference. If the reference pattern is complicated enough, this may require specifying the distribution of each row individually.

5. Add a distribution statement, such as

   DISTRIBUTE X(BLOCK_CYCLIC(T_{lcm}))

   to the program.

Example:

As an example of the application of this algorithm, suppose we have the following ADADs.

D(A,*global*) =

	dimension 1	dimension 2
I	(1,100,*,∅)	(2,100,1,{0})

D(B,*global*) =

	dimension 1	dimension 2
I	(2,199,2,{-2,-1})	(1,100,*,∅)

Then, we should create a Fortran D version of the program by adding the following lines to the program.

```
DECOMPOSITION X(200)
ALIGN A(I,J) with X(2*I)
ALIGN B(I,J) with X(I+1)
DISTRIBUTE X(BLOCK_CYCLIC(2))
```

Proof of correctness:

Assume a need for communication exists. Then, there must be a reference to an element of array A_j by processor p, when that element is not stored on processor p. Since we have used the ADADs as a basis for distribution, there is a reference R to A_j such that the value of dimension D_j is not in the range $[p \times \mathcal{D}(A_j, S, I, D_j)[3] + min(\mathcal{D}(A_j, S, I, D_j)[4]), (p+1) \times \mathcal{D}(A_j, S, I, D_j)[3] + min(\mathcal{D}(A_j, S, I, D_j)[4]) - 1]$ (we assume $\mathcal{D}(A_j, S, I, D_j)[3] \neq 0$, for if it did, then there would be only one reference to A_j, and distribution should be performed in a manner consistent with that reference, as discussed in Section 3.2.4). Because of the procedure for merging ADADs, $D_j \notin [p \times \mathcal{D}(A_j, S, I, D_j)[3] + min(\mathcal{D}(A_j, S, I, D_j)[4]), p \times \mathcal{D}(A_j, S, I, D_j)[3] + max(\mathcal{D}(A_j, S, I, D_j)[4])]$, which in turn implies that the individual ADAD for array A_j and reference R, $\mathcal{D}(A, R)$, must have $\mathcal{D}(A_j, R, I, D_j)[3] \neq \mathcal{D}(A_j, S, I, D_j)[3]$, or that the single element of $\mathcal{D}(A_j, R, I, D_j)[4]$ is not an element of $\mathcal{D}(A_j, S, I, D_j)[4]$. But, under the rules for constructing $\mathcal{D}(A_j, S)$ provided in Section 3.2.3, either situation is impossible. Thus, the assumption that communication is necessary is false. QED.

A value of 0 for the third element indicates a special case for which the requisite distribution is difficult–the rows being accessed by successive iterations must be distributed to successive processors. Describing such distributions is possible in languages like Fortran D, but very complicated. Because of the large amount of work involved, as well as the potential for error, one would not typically attempt to add the necessary code by hand. However, since the alignment statements describing this dis-

tribution can be generated automatically, the presence of such references is a strong argument for the use of a tool that can generate alignment information automatically.

If the third element of a tuple is an asterisk, then communication will be necessary if the array is distributed across that dimension and the loop designated by that iteration variable is to be parallelized. In order to determine which distribution will result in the least communication, we re-evaluate the global ADAD, considering only those ADADs whose level of nesting is maximal. This approach is approximate, based on the assumption that the most deeply nested statements will be executed most frequently. If the new global ADAD still has an asterisk for the third element, much communication will be necessary if we insist on using that iteration variable and distributing across that dimension. Any of the distributions favored by an individual ADAD of maximal nesting is acceptable (if no such distribution exists, then any will suffice).

3.3 An Example

As a simple example of the application of augmented data access descriptors, consider the following program. The program does not perform any useful function; it is intended merely as a simple example of various types of array access and the corresponding ADADs.

```
REAL A(100,100),B(100,100),C(200,100),D(200,200,100)
REAL PI
PARAMETER (PI = 3.14159265)
```

```
      INTEGER I,J,K,M,N

      DO 100 I=1,100
      DO 110 J=1,100
        A(I,J) = SQRT(I*J)                                    (S1)
        B(I,J) = (I+J)/2                                      (S2)
110   CONTINUE
      DO 120 K=1,50
        C(2*I-1,2*K-1) = PI * A(I,2*K-1)                      (S3)
        C(2*I,2*K-1) = C(2*I-1,2*K-1) * B(I,2*K-1)            (S4)
        C(2*I-1,2*K) = PI * B(I,2*K)                          (S5)
        C(2*I,2*K) = C(2*I-1,2*K) * A(I,2*K)                  (S6)
120   CONTINUE
      DO 130 K=1,200
        D(I,K,1) = 0.0                                        (S7)
130   CONTINUE
      DO 140 M=2,100
        DO 140 N=M,100
          D(I+N,I,M) = D(I+N,I,M-1) + C(2*I,I+N)              (S8)
140   CONTINUE
100   CONTINUE
```

Then the data access descriptors for A, B, C, and D may be computed for each of S1 through S8. They are:

D(A,S1) =

2	dimension 1	dimension 2
I	(1,100,1,{0})	(1,100,*,∅)
J	(1,100,*,∅)	(1,100,1,{0})

D(B,S2) =

2	dimension 1	dimension 2
I	(1,100,1,{0})	(1,100,*,∅)
J	(1,100,*,∅)	(1,100,1,{0})

D(A,S3) =

2	dimension 1	dimension 2
I	(1,100,1,{0})	(1,99,*,∅)
K_1	(1,100,*,∅)	(1,99,2,{-1})

D(C,S3) =

2	dimension 1	dimension 2
I	(1,199,2,{-1})	(1,99,*,∅)
K_1	(1,199,*,∅)	(1,99,2,{-1})

D(B,S4) =

2	dimension 1	dimension 2
I	(1,100,1,{0})	(1,99,*,∅)
K_1	(1,100,*,∅)	(1,99,2,{-1})

D_1(C,S4) =

2	dimension 1	dimension 2
I	(1,199,2,{-1})	(1,99,*,∅)
K_1	(1,199,*,∅)	(1,99,2,{-1})

D_2(C,S4) =

2	dimension 1	dimension 2
I	(2,200,2,{0})	(1,99,*,∅)
K_1	(2,200,*,∅)	(1,99,2,{-1})

D(B,S5) =

2	dimension 1	dimension 2
I	(1,100,1,{0})	(2,100,*,∅)
K_1	(1,100,*,∅)	(2,100,2,{0})

D(C,S5) =

2	dimension 1	dimension 2
I	(1,199,2,{-1})	(2,100,*,∅)
K_1	(1,199,*,∅)	(2,100,2,{0})

D(A,S6) =

2	dimension 1	dimension 2
I	(1,100,1,{0})	(2,100,*,∅)
K_1	(1,100,*,∅)	(2,100,2,{0})

D_1(C,S6) =

2	dimension 1	dimension 2
I	(1,199,2,{-1})	(2,100,*,∅)
K_1	(1,199,*,∅)	(2,100,2,{0})

D_2(C,S6) =

2	dimension 1	dimension 2
I	(2,200,2,{0})	(2,100,*,∅)
K_1	(2,200,*,∅)	(2,100,2,{0})

D(D,S7) =

2	dimension 1	dimension 2	dimension 3
I	(1,100,1,{0})	(1,200,*,∅)	(1,1,*,∅)
K_2	(1,100,*,∅)	(1,200,1,{0})	(1,1,*,∅)

D(C,S8) =

3	dimension 1	dimension 2
I	(2,200,2,{0})	(3,200,*,∅)
M	(2,200,*,∅)	(3,200,*,∅)
N	(2,200,*,∅)	(3,200,*,∅)

D_1(D,S8) =

3	dimension 1	dimension 2	dimension 3
I	(3,200,*,∅)	(1,100,1,{0})	(1,99,*,∅)
M	(3,200,*,∅)	(1,100,*,∅)	(1,99,1,{-1})
N	(3,200,*,∅)	(1,100,*,∅)	(1,99,*,∅)

D_2(D,S8) =

3	dimension 1	dimension 2	dimension 3
I	(3,200,*,∅)	(1,100,1,{0})	(2,100,*,∅)
M	(3,200,*,∅)	(1,100,*,∅)	(2,100,1,{0})
N	(3,200,*,∅)	(1,100,*,∅)	(2,100,*,∅)

Note that the variable name K is used for two logically different iteration variables. To accommodate this situation, the variable names in the ADADs have been subscripted. From these ADADs we can derive:

D(A,*global*) =

	dimension 1	dimension 2
I	(1,100,1,{0})	(1,100,*,∅)

D(B,*global*) =

	dimension 1	dimension 2
I	(1,100,1,{0})	(1,100,*,∅)

D(C,*global*) =

	dimension 1	dimension 2
I	(1,200,2,{0})	(1,100,*,∅)

D(D,*global*) =

	dimension 1	dimension 2	dimension 3
I	(1,200,*,∅)	(1,200,*,∅)	(1,100,*,∅)

Since I is the only loop common to all statements, it is the only candidate for parallelization. From the global augmented data access descriptors, it may be seen that arrays A and B should be distributed in a row-wise manner (that is, A(1,1..100) and B(1,1..100) on one processor, A(2,1..100) and B(2,1..100) on another, etc.). Matrix C should be distributed in striated row-wise fashion, with striations of width 2. Array D, however, cannot be distributed without a need for communication. Thus, the distribution favored by those statements with the greatest level of nesting is determined. In this case, the statement whose nesting level is maximal is statement S8 (rather than S7, the only other statement to reference D). The merged access descriptor is thus:

D(D,{S8}) =

	dimension 1	dimension 2	dimension 3
I	(3,200,*,∅)	(1,100,1,{0})	(1,100,*,∅)

So, array D should be distributed across its second dimension. It should be noted that all communication could be eliminated if the order of the subscripts in S7 were reversed, which is most likely the programmer's intention. Thus, code that contains errors that do not necessarily lead to incorrect answers (suppose, for example, that the Fortran compiler automatically initializes all arrays to all zeroes) can still lead to inefficient parallel programs.

The Fortran D code corresponding to this alignment is:

```
REAL A(100,100),B(100,100),C(200,100),D(200,200,100)
DECOMPOSITION X(200)
ALIGN A(I,J) with X(2*I)
```

```
      ALIGN B(I,J) with X(2*I)
      ALIGN C(I,J) with X(I)
      ALIGN D(I,J,K) with X(2*J) overflow (WRAP)
      DISTRIBUTE X(BLOCK_CYCLIC(2))

      REAL PI
      PARAMETER (PI = 3.14159265)

      INTEGER I,J,K,M,N

      PARALLEL DO 100 I=1,100
      DO 110 J=1,100
        A(I,J) = SQRT(I*J)
        B(I,J) = (I+J)/2
110   CONTINUE
      DO 120 K=1,50
        C(2*I-1,2*K-1) = PI * A(I,2*K-1)
        C(2*I,2*K-1) = C(2*I-1,2*K-1) * B(I,2*K-1)
        C(2*I-1,2*K) = PI * B(I,2*K)
        C(2*I,2*K) = C(2*I-1,2*K) * A(I,2*K)
120   CONTINUE
      DO 130 K=1,200
        D(I,K,1) = 0.0
130   CONTINUE
      DO 140 M=2,100
        DO 140 N=M,100
          D(I+N,I,M) = D(I+N,I,M-1) + C(2*I,I+N)
140   CONTINUE
100   CONTINUE
```

3.4 Interprocedural Analysis

It is often the case that a section of code that we wish to analyze contains one or more calls to subroutines. Therefore, we must establish a framework for computing ADADs

when such a call occurs. It is a fairly simple task to include procedure calls in the scheme for combining ADADs. First, we compute the ADAD(s) for the subroutine, as usual. Then, the ADAD(s) for the call statement can be computed by replacing the formal parameters in the ADAD(s) computed for the subroutine with the actual parameters of the call. If the arrays undergo reshaping due to the nature of the call, then this reshaping and the indexing used need to be taken into account. Any rows corresponding to iteration variables local to the subroutine can be ignored, since we are restricting ourselves to an analysis at a higher level. If we wish to consider these variables in our alignment analysis, we should determine the best alignment for the section of code before the call, the best alignment for the subroutine, and the best alignment for the section of code after the call. Finally, the ADAD(s) for the call statement can be merged with the ADADs for the rest of the section. This procedure is best illustrated with an example. Suppose we have the following section of code:

```
DOUBLE PRECISION A(100),X(100,100),Y(100)

DO I=1, 100
   A(I) = I                          (S1)
   CALL SUBR(I,A,X(1,I))             (S2)
   Y(I) = 0                          (S3)
   DO J=1, 100
      Y(I) = Y(I) + X(J,I)           (S4)
   ENDDO
ENDDO
```

and that the code for SUBR looks like:

```
SUBROUTINE SUBR(I,X,Y)

DOUBLE PRECISION X(100),Y(100)
```

```
INTEGER I,J

DO J=1, 100
  Y(J) = SQRT(X(I)*J)
ENDDO

END
```

Then, the ADADs for the subroutine are:

D(X,SUBR) =

2	dimension 1
I	(1,∞,1,{0})
J	(1,∞,*,$\emptyset$)

and

D(Y,SUBR) =

2	dimension 1
I	(1,100,*,$\emptyset$)
J	(1,100,1,{0})

Following the rules outlined above, we get the following ADADs for statement S2:

D(A,S2) =

1	dimension 1
I	(1,100,1,{0})

and

$D(\texttt{X,S2}) =$

1	dimension 1	dimension 2
I	(1,100,*,$\emptyset$)	(1,100,1,{0})

These may be combined with the ADADs for the rest of the section:

$D(\texttt{A,S1}) =$

1	dimension 1
I	(1,100,1,{0})

$D(\texttt{Y,S3}) =$

1	dimension 1
I	(1,100,1,{0})

$D(\texttt{X,S4}) =$

2	dimension 1	dimension 2
I	(1,100,*,$\emptyset$)	(1,100,1,{0})
J	(1,100,1,{0})	(1,100,*,$\emptyset$)

and

D(Y,S4) =

2	dimension 1
I	(1,100,1,{0})
J	(1,100,*,∅)

to yield the ADADs for the section:

D(A,*section*) =

1	dimension 1
I	(1,100,1,{0})

D(X,*section*) =

1	dimension 1	dimension 2
I	(1,100,*,∅)	(1,100,1,{0})

and

D(Y,*section*) =

1	dimension 1
I	(1,100,1,{0})

Thus, this computation may be performed in a manner free from communication, if A and Y are distributed across their first dimensions, and X is distributed across its second dimension.

The Fortran D code for this is:

```
DOUBLE PRECISION A(100),X(100,100),Y(100)
```

```
DECOMPOSITION B(100)
ALIGN A(I) with B(I)
ALIGN X(I,J) with B(I)
ALIGN Y(I) with B(I)
DISTRIBUTE B(CYCLIC)

PARALLEL DO I=1, 100
   A(I) = I
   CALL SUBR(I,A,X(1,I))
   Y(I) = 0
   DO J=1, 100
      Y(I) = Y(I) + X(J,I)
   ENDDO
ENDDO
```

CHAPTER 4

The Effects of Loop Transformations on ADADs

Most parallelizing compilers attempt to perform a number of different loop transformations, in the hope of enabling the parallelization of loops that previously could not be parallelized. Loop transformations are also used to increase the granularity of a program. When we use a data parallel programming paradigm, our concerns are different. In particular, our primary goal is to reduce the amount of communication required. Loop transformations may help us to achieve that goal. Therefore, the effects of some of the standard loop transformations on augmented data access descriptors are presented, not because it is easier to modify ADADs than it is to recalculate them, but so that our transformations may be guided by the effect they will have on the ADADs.

4.1 Promotion

Promotion entails the addition of a dimension to a variable, which is indexed by the iteration variable of a loop within which the variable is defined. This technique is particularly appropriate for intermediate variables. For example, consider the following section of code:

```
DO I=1, 100
  DO J=1, 10
     TEMP(J) = COS(A(I,I+J))              (S1)
     B(I,J) = 2.0*TEMP(J)                 (S2)
  ENDDO
ENDDO
```

The variable TEMP is not indexed by I and therefore can not be distributed among the processors using that variable. However, if we perform a promotion, we get the following code:

```
DO I=1, 100
  DO J=1, 10
     TEMP(I,J) = COS(A(I,I+J))            (S1')
     B(I,J) = 2.0*TEMP(I,J)               (S2')
  ENDDO
ENDDO
```

which enables TEMP, A, and B to be distributed among up to 100 different processors.

After a promotion, the ADADs associated with the variable being promoted should have a dimension (column) added. The tuple for the iteration variable with which we are indexing the new dimension will be $(L, U, 1, \{0\})$, where L and U are the lower and upper bounds, respectively, on the iteration variable. The tuple for all other rows is $(L, U, *, \emptyset)$. So, for TEMP in the example above, the original ADADs were:

D(TEMP,S1) =

2	dimension 1
I	(1,10,*,∅)
J	(1,10,1,{0})

and

D(TEMP,S2) =

2	dimension 1
I	(1,10,*,∅)
J	(1,10,1,{0})

After promotion, the ADADs are:

D(TEMP,S1') =

2	dimension 1	dimension 2
I	(1,100,1,{0})	(1,10,*,∅)
J	(1,100,*,∅)	(1,10,1,{0})

and

D(TEMP,S2') =

2	dimension 1	dimension 2
I	(1,100,1,{0})	(1,10,*,∅)
J	(1,100,*,∅)	(1,10,1,{0})

4.2 Loop Unrolling

In loop unrolling, the number of iterations of the loop is reduced by changing the step size, and adding statements to accommodate the values between steps. Since we have restricted actual step sizes to a length of 1, the effective step size may be changed by dividing the loop's upper bound by some factor, which we will refer to as the unrolling factor, and multiplying the iteration variable by that factor. For example,

```
DO I=1, 100
   A(2*I) = B(I+2)          (S1)
ENDDO
```

might become (with an unrolling factor of 4)

```
DO I=1, 25
   A(8*I-6) = B(4*I-1)      (S1a)
   A(8*I-4) = B(4*I)        (S1b)
   A(8*I-2) = B(4*I+1)      (S1c)
   A(8*I) = B(4*I+2)        (S1d)
ENDDO
```

If a loop is unrolled by a factor M, the ADADs for statements within the loop could be expanded to M ADADs, one for each statement introduced by the unrolling. However, it is simpler to modify the original ADAD to one which describes the M derived statements. Only tuples in the row corresponding to the loop being unrolled are affected. The first and second elements in each tuple remain unchanged. If the third element, $\mathcal{D}(A, S, I, D)[3]$, is * or 0, it should remain unchanged. Otherwise, $\mathcal{D}(A, S, I, D)[3]$ should be multiplied by M and the fourth element modified according to the following procedure. For each element E in the original set, $\mathcal{D}(A, S, I, D)[4]$,

introduce M elements to the new set, $\mathcal{D}(A, \{S_a, S_b, ...\}, I, D)[4]$, equal to $E + k \times \mathcal{D}(A, S, I, D)[3], (k = [1 - M, 0])$. The original ADADs for the example above are:

D(A,S1) =

1	dimension 1
I	(2,200,2,{0})

and

D(B,S1) =

1	dimension 1
I	(3,102,1,{2})

The new augmented data access descriptors would be:

D(A,{S1a,S1b,S1c,S1d}) =

1	dimension 1
I	(2,200,8,{-6,-4,-2,0})

and

D(B,{S1a,S1b,S1c,S1d}) =

1	dimension 1
I	(3,102,4,{-1,0,1,2})

So, the Fortran D alignment statements might look like:

```
DECOMPOSITION X(200)
```

```
      ALIGN A(I) with X(I)
C The expression 2*I-4 in the next statement derives from
C the fact that 8/4=2 and B has offset 2, so we need to
C align B(I) with X(2*(I-2)).
      ALIGN B(I) with X(2*I-4) overflow (WRAP)
      DISTRIBUTE X(BLOCK_CYCLE(8))
```

4.3 Loop Fusion

In loop fusion, two adjacent loops are merged to form one loop. The iteration variables of the original loops are replaced by the new iteration variable. For example,

```
DO I=1, 100
   B(I) = 2*I*A(2*I)        (S1)
ENDDO
DO J=1, 100
   C(J) = A(2*J-1) - 1      (S2)
ENDDO
```

could become

```
DO K=1, 100
   B(K) = 2*K*A(2*K)        (S1')
   C(K) = A(2*K-1) - 1      (S2')
ENDDO
```

In order to perform loop fusion, while preserving program correctness, there are certain criteria which must be satisfied. The formal verification that transformations are correctness-preserving is left to books on parallel compiler construction [13]. Instead, we restrict ourselves to examples that are simple enough that correctness is readily apparent.

Modifying the ADADs affected by a loop fusion, once it has been determined that one may occur, is simple. The rows of the ADADs that corresponded to the original iteration variables now correspond to the new iteration variable. Also, the ADADs describing a region of code may change, and should be computed. Thus, for the example above, the original ADADs of

D(A,S1) =

1	dimension 1
I	(2,200,2,{0})

D(B,S1) =

1	dimension 1
I	(1,100,1,{0})

D(A,S2) =

1	dimension 1
J	(1,199,2,{-1})

D(C,S2) =

1	dimension 1
J	(1,100,1,{0})

D(A,{S1,S2}) = NULL

become:

D(A,S1') =

1	dimension 1
K	(2,200,2,{0})

D(B,S1') =

1	dimension 1
K	(1,100,1,{0})

D(A,S2') =

1	dimension 1
K	(1,199,2,{-1})

D(C,S2') =

1	dimension 1
K	(1,100,1,{0})

D(A,{S1',S2'}) =

1	dimension 1
K	(1,200,2,{-1,0})

4.4 Loop Interchange

As the name implies, loop interchange entails the interchange of two loops. The outer loop becomes the inner loop, and the inner loop becomes the outer loop. Thus,

```
DO I=1, 100
  DO J=1, 100
```

```
      A(I,J) = 0.0
    ENDDO
  ENDDO
```

would become

```
DO J=1, 100
  DO I=1, 100
     A(I,J) = 0.0
  ENDDO
ENDDO
```

As with loop fusion, the conditions under which loop interchange may safely be performed depend on sophisticated data dependence analysis [13]. However, assuming loop interchange can be performed, the ADADs associated with individual statements and with regions of the program will remain unchanged.

4.5 Strip Mining

Strip mining is often used in parallelizing compilers to increase the amount of work performed inside of a loop. It is analogous to loop unrolling, except that rather than increasing the number of statements by some factor M, a new loop is introduced to cover the range of iterations between the outer steps. So,

```
DO I=1, 1000
   A(I) = 3.14*(R(I)**2)
ENDDO
```

becomes

```
DO I=1, 10
```

```
    DO J=1, 100
      A(100*(I-1)+J) = 3.14*(R(100*(I-1)+J)**2)
    ENDDO
  ENDDO
```

This is an extremely useful technique when we want to parallelize a particular loop for execution on a shared memory computer with a relatively small number of processors. However, the process of strip mining results in all tuples associated with the original iteration variable, as well as those associated with the new iteration variable, having a third element of *. Thus, when we are attempting to exploit global parallelism on a large distributed memory computer, strip mining can have an adverse effect. Instead of increasing the work load per processor through strip mining, we should utilize the distribution facilities of the data parallel language being used.

4.6 Invariant Code Movement

If a statement within a loop uses variables that do not change within the loop, defines a variable that is not defined elsewhere in the loop nor used earlier in the loop, and the statement does not occur within a conditional, then that statement may be moved to a position immediately before the loop. This is useful for eliminating extra calculations. For example,

```
DO J=1, 5
   V(J) = 0.0                        (S1)
   DO I=1, 100
      A(J) = 3.14*R(J)*R(J)          (S2)
      V(J) = V(J) + A(J)*F(I)        (S3)
   ENDDO
ENDDO
```

can be changed to

```
DO J=1, 5
   V(J) = 0.0                          (S1)
   A(J) = 3.14*R(J)*R(J)               (S2')
   DO I=1, 100
      V(J) = V(J) + A(J)*F(I)          (S3)
   ENDDO
ENDDO
```

When a statement like this is moved outside of a loop, the row in the ADAD for the statement corresponding to that loop's iteration variable should be eliminated, and the integer representing the nesting level of the statement should be decremented. So, in the previous example,

D(A,S2) =

2	dimension 1
I	(1,5,*,∅)
J	(1,5,1,{0})

becomes

D(A,S2') =

1	dimension 1
J	(1,5,1,{0})

Although this technique does not assist in the parallelization of the program, it can result in significant performance improvements, by eliminating unnecessary work.

4.7 An Example

To understand the usefulness of directed loop transformations, consider the following section of a program:

```
DO I=1, 100
  DO J=1, 100
     A(J,I) = A(J,I) + SQRT(J*I)     (S1)
  ENDDO
  DO K=1, 50
     A(2*K,I) = A(2*K-1,I)           (S2)
  ENDDO
  DO L=1, 100
    A(L,I+1) = A(L,I)                (S3)
  ENDDO
ENDDO
```

If we attempt to determine the best alignment for this code, we get the following ADADs:

D_1(A,S1) =

2	dimension 1	dimension 2
I	(1,100,*,∅)	(1,100,1,{0})
J	(1,100,1,{0})	(1,100,*,∅)

D_2(A,S1) =

2	dimension 1	dimension 2
I	(1,100,*,∅)	(1,100,1,{0})
J	(1,100,1,{0})	(1,100,*,∅)

D_1(A,S2) =

2	dimension 1	dimension 2
I	(2,100,*,∅)	(1,100,1,{0})
K	(2,100,2,{0})	(1,100,*,∅)

D_2(A,S2) =

2	dimension 1	dimension 2
I	(1,99,*,∅)	(1,100,1,{0})
K	(1,99,2,{-1})	(1,100,*,∅)

D_1(A,S3) =

2	dimension 1	dimension 2
I	(1,100,*,∅)	(2,101,{1})
L	(1,100,1,{0})	(2,101,*,∅)

D_2(A,S3) =

2	dimension 1	dimension 2
I	(1,100,*,∅)	(1,100,{0})
L	(1,100,1,{0})	(1,100,*,∅)

and

D(A,*global*) =

1	dimension 1	dimension 2
I	(1,100,*,∅)	(1,101,*,∅)

So, there is no way in which to distribute this code without the need for communication being introduced. However, if we unroll the first and third loops by a factor of two, then fuse the three loops, we get:

```
DO I=1, 100
  DO J=1, 50
     A(2*J-1,I) = A(2*J-1,I) + SQRT((2*J-1)*I)     (S1a')
     A(2*J,I) = A(2*J,I) + SQRT(2*J*I)             (S1b')
     A(2*J,I) = A(2*J-1,I)                         (S2')
     A(2*J-1,I+1) = A(2*J-1,I)                     (S3a')
     A(2*J,I+1) = A(2*J,I)                         (S3b')
  ENDDO
ENDDO
```

which has the following ADADs:

D_1(A,S1a') =

2	dimension 1	dimension 2
I	(1,99,*,∅)	(1,100,1,{0})
J	(1,99,2,{-1})	(1,100,*,∅)

D_2(A,S1a') =

2	dimension 1	dimension 2
I	(1,99,*,∅)	(1,100,1,{0})
J	(1,99,2,{-1})	(1,100,*,∅)

D_1(A,S1b') =

2	dimension 1	dimension 2
I	(2,100,*,∅)	(1,100,1,{0})
J	(2,100,2,{0})	(1,100,*,∅)

D_2(A,S1b') =

2	dimension 1	dimension 2
I	(2,100,*,∅)	(1,100,1,{0})
J	(2,100,2,{0})	(1,100,*,∅)

D_1(A,S2') =

2	dimension 1	dimension 2
I	(2,100,*,∅)	(1,100,1,{0})
J	(2,100,2,{0})	(1,100,*,∅)

D_2(A,S2) =

2	dimension 1	dimension 2
I	(1,99,*,∅)	(1,100,1,{0})
J	(1,99,2,{-1})	(1,100,*,∅)

D_1(A,S3a') =

2	dimension 1	dimension 2
I	(1,99,*,∅)	(2,101,{1})
J	(1,99,2,{-1})	(2,101,*,∅)

D_2(A,S3a') =

2	dimension 1	dimension 2
I	(1,99,*,∅)	(1,100,{0})
J	(1,99,2,{-1})	(1,100,*,∅)

D_1(A,S3b') =

2	dimension 1	dimension 2
I	(2,100,*,∅)	(2,101,{1})
J	(2,100,2,{0})	(2,101,*,∅)

D_2(A,S3b') =

2	dimension 1	dimension 2
I	(2,100,*,∅)	(1,100,{0})
J	(2,100,2,{0})	(1,100,*,∅)

and

D(A,*global*) =

2	dimension 1	dimension 2
I	(1,100,*,∅)	(1,101,*,∅)
J	(1,100,2,{-1,0})	(1,101,*,∅)

So, after the transformations, we may distribute A along the first dimension, using a stride size of 2, and we will have no need for communication. This could be achieved using:

```
DECOMPOSITION X(100)
ALIGN A(I) WITH X(I)
DISTRIBUTE X(BLOCK_CYCLE(2))
```

This represents a significant improvement over the original version, which would have had to forgo all parallelism, or suffer the added cost of communicating information between processors.

CHAPTER 5

Performance Analysis

Having described ADADs, the manner in which they are combined, how they should be interpreted, and the effect upon them of various loop transformations, we are now prepared to examine the manner in which they may be applied to the parallelization of vectorizable programs. We begin by examining some of the properties of ADIFOR-generated code and examine how an understanding of these properties assists in the task of parallelization. We then propose a method using ADADs to decide how a vectorizable program should be parallelized and apply this method to two simple programs. Finally, we confirm the conclusions made about these programs through performance measurements on a parallel computer.

5.1 Parallelizing ADIFOR-generated code

The parallelization of code generated by ADIFOR is a complicated problem, because nothing is known about the algorithms to which ADIFOR will be applied. In cases where the original algorithm is not parallelizable, it is highly desirable that the

parallelism inherent to ADIFOR-generated code be exploited to the greatest extent possible. However, it is not clear that this is the case when the original algorithm has a large amount of parallelism. In such cases, it may be best to ignore the additional parallelism introduced by ADIFOR.

Because ADIFOR uses primarily the forward mode[1] of automatic differentiation [2], a large number of vector operations are embedded in the newly generated code. For example, given the code section:

```
      IC=ICOMP(J,I)
      SIGT(J,I)=SIGMAT(IC)
      C(J,I)=CC(IC)
      QEXT(J,I)=Q(IC)
```

ADIFOR will produce:

```
            ic = icomp(j, i)
C           sigt(j, i) = sigmat(ic)
            do 99798 g$i$ = 1, g$p$
              g$sigt(g$i$, j, i) = g$sigmat(g$i$, ic)
99798       continue
            sigt(j, i) = sigmat(ic)
C           c(j, i) = cc(ic)
            do 99797 g$i$ = 1, g$p$
              g$c(g$i$, j, i) = g$cc(g$i$, ic)
99797       continue
            c(j, i) = cc(ic)
C           qext(j, i) = q(ic)
            do 99796 g$i$ = 1, g$p$
              g$qext(g$i$, j, i) = g$q(g$i$, ic)
99796       continue
            qext(j, i) = q(ic)
```

[1] for a discussion of the differences between the forward and reverse modes of automatic differentiation, please see [6].

Two options for global parallelism present themselves. If the original algorithm possessed parallelism in an outer loop, the ADIFOR-generated code will too, and we can use this for parallelizing the code. However, if this parallelism does not exist, we can exploit the parallelism inherent in the vector loops. In order to achieve this, we replicate scalar operations across all processors, then perform all vector operations in parallel. This is roughly equivalent to promoting all scalars and fusing the vector loops. We shall refer to this technique as the *inside-out method.* For example, the code fragment:

```
A=PI*(R**2)
W[1..30]=A*X[1..30]
```

might become:

PROCESSOR 1	PROCESSOR 2	PROCESSOR 3
`A=PI*(R**2)`	`A=PI*(R**2)`	`A=PI*(R**2)`
`W[1..10]=A*X[1..10]`	`W[11..20]=A*X[11..20]`	`W[21..30]=A*X[21..30]`

In order to get an upper bound on the speedup achievable with this method, assume that the time to branch and join is negligible compared to the total time for a computation. Also, let:

N = number of processors to be used
C = number of columns of the jacobian to be evaluated
$E(0)$ = time to compute original function
$E(1)$ = time to compute function plus one column of Jacobian
$E(2)$ = time to compute function plus two columns of Jacobian

Then, the time to compute one element of each of the gradient vectors (one column of the Jacobian), E_V, is:

$$E_V = E(2) - E(1)$$

Furthermore, the time spent doing scalar operations (computing the value of the function, computing reverse mode objects, and loop overhead), E_S, is:

$$E_S = E(1) - E_V = 2 * E(1) - E(2)$$

The run time of the sequential program should therefore be:

$$E_S + CE_V$$

The time required using the inside-out method and a slice size of 1 is:

$$\left\lceil \frac{C}{N} \right\rceil (E_S + E_V) = \left\lceil \frac{C}{N} \right\rceil E(1)$$

Accordingly, speedup is:

$$\frac{E_S + CE_V}{\left\lceil \frac{C}{N} \right\rceil (E_S + E_V)}$$

Using a more appropriate slice size ($\lceil C/N \rceil$), we achieve a runtime of:

$$E_S + \left\lceil \frac{C}{N} \right\rceil E_V$$

Since E_S is guaranteed to be at least $E(0)$, the running time of the original algorithm, this represents a significant improvement in cases where C is substantially larger than

N. The speedup is:

$$\frac{E_S + CE_V}{E_S + \left\lceil \frac{C}{N} \right\rceil E_V}$$

5.2 Applying ADADs to Vectorizable Code

Based on the results of the previous section, we propose a method for determining the best manner in which to parallelize a vectorizable program. First, the original program should be simplified by replacing all vector operations by scalar operations. Next, ADADs may be used to determine an alignment for the resultant code. If this alignment if free from, or nearly free from, communication, then this alignment may be employed in parallelizing the original program. Otherwise, we may utilize the inside-out method, promoting the scalar operations and fusing the vector loops to yield a program with a high degree of global parallelism, at the cost of duplicate computations.

As an application of this technique, consider the programs in Appendix A. Program 1 computes $f_i = sin(x_i)$ through a series of trigonometry identities. The reason for doing this is to make the solution easy to check. Program 2 computes $f = \sum_{i=1}^{100} sin(x_i)$ through the same identities. Note that in the case of ADIFOR-generated code, replacing the vector operations with scalar operations yields a program roughly equivalent to the original program. Thus, it is sufficient to examine the original programs, and apply our conclusions to the ADIFOR-generated code, which

can be found in Appendix B. It is obvious from inspection that the first program possesses a high degree of parallelism while the second is naturally sequentially. However, the same conclusions can be reached methodically through the use of ADADs.

First, consider program 1. The augmented data access descriptors are:

D(temp1,S1) =

1	dimension 1
I	(1,100,1,{0})

D_1(x,S1) =

1	dimension 1
I	(1,100,1,{0})

D_2(x,S1) =

1	dimension 1
I	(1,100,1,{0})

D(temp2,S2) =

1	dimension 1
I	(1,100,1,{0})

D_1(x,S2) =

1	dimension 1
I	(1,100,1,{0})

D_2(x,S2) =

1	dimension 1
I	(1,100,1,{0})

D(temp3,S3) =

1	dimension 1
I	(1,100,1,{0})

D_1(x,S3) =

1	dimension 1
I	(1,100,1,{0})

D_2(x,S3) =

1	dimension 1
I	(1,100,1,{0})

D_3(x,S3) =

1	dimension 1
I	(1,100,1,{0})

D_4(x,S3) =

1	dimension 1
I	(1,100,1,{0})

D(f,S4) =

1	dimension 1
I	(1,100,1,{0})

D_1(temp1,S4) =

1	dimension 1
I	(1,100,1,{0})

D_2(temp1,S4) =

1	dimension 1
I	(1,100,1,{0})

D(temp2,S4) =

1	dimension 1
I	(1,100,1,{0})

D(temp3,S4) =

1	dimension 1
I	(1,100,1,{0})

D(x,S4) =

1	dimension 1
I	(1,100,1,{0})

These may be combined to yield the global ADADs:

D(x,*global*) =

	dimension 1
I	(1,100,1,{0})

D(temp1,*global*) =

	dimension 1
I	(1,100,1,{0})

D(temp2,*global*) =

	dimension 1
I	(1,100,1,{0})

D(temp3,*global*) =

	dimension 1
I	(1,100,1,{0})

D(f,*global*) =

	dimension 1
I	(1,100,1,{0})

Thus, this program may be parallelized by parallelizing the I loop and distributing x,temp1,temp2,temp3, and f across their first (and only) dimension. The Fortran D code for doing the alignment is:

```
DECOMPOSITION A(100)
ALIGN F(I) with A(I)
ALIGN X(I) with A(I)
ALIGN TEMP1(I) with A(I)
ALIGN TEMP2(I) with A(I)
ALIGN TEMP3(I) with A(I)
DISTRIBUTE A(CYCLIC)
```

Next, we verify that program 2 can not be parallelized. Since it is impossible to

distribute a variable if it is only a scalar, we promote these variables, yielding program 3. However, even this program is inherently sequential. The augmented data access descriptors for `farray` are:

D_1(`farray`,S4) =

1	dimension 1
I	(1,100,1,{0})

and

D_2(`farray`,S4) =

1	dimension 1
I	(2,101,1,{1})

When combined, these yield the global ADAD:

D(`farray`,*global*) =

	dimension 1
I	(1,101,*,∅)

Thus, we are unable to distribute `farray` in any way that prevents communication. This limitation is what makes the inside-out method so attractive when we attempt to parallelize the ADIFOR-generated code.

5.3 Empirical Results

In order to confirm the conclusions reached in the previous section, both programs were implemented on a BBN Butterfly TC2000. Although the TC2000 is a shared

memory machine, it is similar to distributed memory machines in that it exhibits non-uniform memory access times, dependent upon which processor attempts to reference a particular storage location.

The programs were processed by ADIFOR and the resultant program parallelized. As can be observed from the data in Figures 1 and 2, the performance of actual implementations was in keeping with the projections made using our simplistic model in Section 5.1. In particular, note that the speedup of the inside-out method is bounded by $N/2$.

The degradation in efficiency as N increases can probably be attributed to the overhead associated with forks and joins, and would probably be less significant in larger problems. Taking this into account, we can conclude that for computationally intensive programs, the speedup of a program that requires no communication will be nearly linear, while only a speedup of $N/2$ can be achieved for programs that must be parallelized using the inside-out technique to eliminate the need for communication.

Thus, by using ADADs to determine a good alignment for the original code and by applying the conclusions of section 5.1 regarding the speedup available using the inside-out method for a particular ratio of vector to scalar computations, we can make accurate predictions concerning the best way to parallelize a program. In the case of ADIFOR-generated code, if the efficiency of a program containing alignment information is better than 50%, it is probably not worthwhile to attempt to use the inside-out technique on the derivative program.

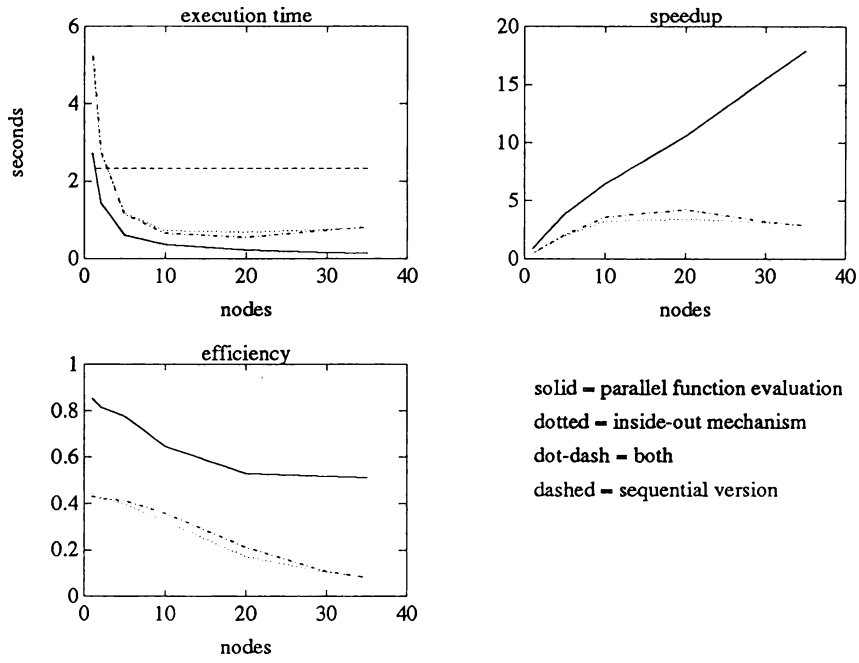


Figure 5.1. Performance of various parallelization methods for program 1

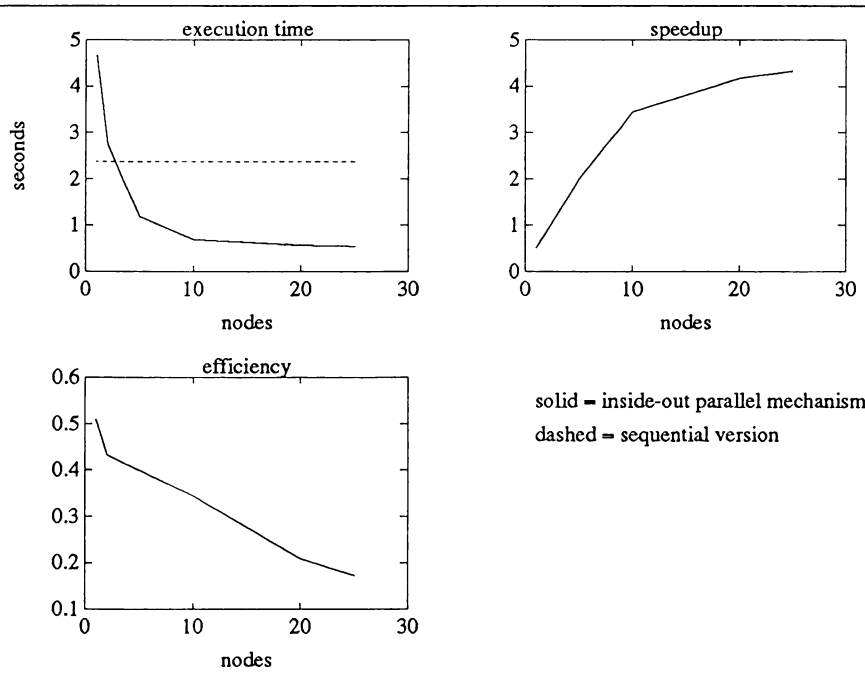


Figure 5.2. Performance of inside-out method for program 2

CHAPTER 6

Conclusions

6.1 Summary

Many of today's scientific applications, such as climate modeling and superconductivity studies, are so complex that they require the use of the most powerful supercomputers available. Today, this class is represented by parallel computers. A great deal of effort has gone into developing programs that vectorize well. In addition, there exists an abundance of naturally vectorizable code, such as that generated by ADIFOR, an automatic differentiation tool. Because there exists such a large class of computationally difficult problems that vectorize well, we would like to be able to parallelize vectorizable programs.

However, the parallelization of programs that perform well on vector supercomputers is not a trivial task, because vector supercomputers and parallel supercomputers exploit different types of parallelism. Vector computers utilize fine-grained parallelism, while parallel computers, especially distributed memory supercomputers,

achieve better performance using coarse-grained parallelism. This results in part from the memory hierarchies of the machines in question. So, the problem becomes one of determining the best distribution of data among the various processors.

We presented a formal technique for determining a good distribution of data using the augmented data access descriptor (ADAD). This technique differs from previous approaches in that it views the problem of data alignment as an extension of data dependence analysis, rather than a completely new problem. Thus, the Data Access Descriptor [1], used for data dependence analysis, may be augmented to facilitate data alignment analysis, yielding the augmented data access descriptor. This approach is simple, efficient, and quite accurate.

Ways to handle interprocedural analysis and the effects of loop transformations were presented. The former was discussed so that programs with good modularity are not penalized by incomplete analysis. The effects of loop transformations were examined so that they might guide decisions as to which transformations should be performed. An example showing how this knowledge could be utilized was provided.

The question of how to apply ADADs to the parallelization of vectorizable code was also addressed. ADADs can be used to discover a good data alignment for any program. However, in the case of vectorizable programs, a second option presents itself. If an analysis of the program with all vector operations replaced by a scalar operation reveals that an alignment free from, or nearly free from, communication exists, then this alignment may be used to parallelize the program. Otherwise, the scalar operations of the program may be promoted and the vector loops fused, enabling a high degree of global parallelism, at the cost of duplicate computations.

We term this technique the inside-out method. The viability of this procedure was confirmed through performance analysis on a BBN Butterfly TC2000.

6.2 Future Studies

Possible subjects for future research include the development of a tool to automatically compute ADADs, plus the development of a tool that can use ADADs and the original source code to automatically generate source code in some language with data alignment constructs, such as Fortran D or DINO [5, 12]. Construction of these tools, especially the former, would help to resolve several unknown aspects of ADAD analysis. In particular, it is not clear what the storage requirements for the data structures used to describe an actual application program might be. Also, analysis of real programs would facilitate an assessment of the viability of ADADs in the automatic data partitioning of vectorizable programs. Should ADADs prove inadequate, we may employ a more sophisticated analysis (at the cost of performance) which preserves the notion of data alignment as data dependence analysis and recognizes the importance of interprocedural analysis and directed loop transformations.

An important feature of ADAD analysis is that it can be done incrementally. Analysis is performed on small regions of a program, and the ADADs are then merged so that they describe larger regions of the program. This feature makes ADAD analysis particularly well suited for a programming environment, because small changes to the program do not mandate another complete analysis of the program, but instead incur a small additional cost. In addition, incremental merging of ADADs may allow

a programmer to detect what statements are blocking a communication free distribution. This knowledge may enable the programmer to rewrite the program in a manner better suited for global parallelization. Thus, future studies should examine what hierarchy of ADAD analysis best facilitates these two goals.

Another issue yet to be addressed is whether it is better to apply the method of ADAD analysis to the vectorizable code itself, or a scalar version of that code (in the case of ADIFOR, the scalar version is simply the original program). In the section 5.2, we observed that one can make good predictions as to the best approach using information about the ratio of vector statements to scalar statements, and the application of ADADs to the scalar version of the code. However, we could have come to the same conclusion by applying ADADs to the ADIFOR-generated code, after several loop transformations. The former approach offers simplicity and speed, while the latter approach could potentially offer greater accuracy. It remains to some future analysis to determine which approach is best.

BIBLIOGRAPHY

BIBLIOGRAPHY

[1] V. Balasundaram. Interactive parallelization of numerical scientific programs. Technical Report TR89-95, Dept of Computer Science, Rice University, 1989.

[2] Christian Bischof, Alan Carle, George Corliss, Andreas Griewank, and Paul Hovland. Adifor: Generating derivative codes from Fortran programs. *Scientific Programming*, 1(1), 1992.

[3] David Callahan, Jack Dongarra, and David Levine. Vectorizing compilers: A test suite and results. In *Proceedings of Supercomputing '88*, pages 98–105. IEEE Computer Science Press, 1988. also Argonne Technical Report ANL/MCS-TM-109.

[4] Bruce W. Char, Keith O. Geddes, Gaston H. Gonnet, Benton L. Leong, Michael B. Monagan, and Stephen M. Watt. *Maple V Language Reference Manual.* Springer Verlag, New York, 1991.

[5] G. Fox, S. Hiranandani, K. Kennedy, C. Koelbel, U. Kremer, C.-W. Tseng, and M.-Y. Wu. Fortran d language specification. Technical Report CRPC-TR90079, Center for Research in Parallel Computation, Rice University, December 1990.

[6] Andreas Griewank. On automatic differentiation. In *Mathematical Programming: Recent Developments and Applications*, pages 83–108, Amsterdam, 1989. Kluwer Academic Publishers.

[7] M. Gupta and P. Banerjee. Demonstration of automatic data partitioning techniques for parallelizing compilers on multicomputers. *IEEE Transactions on Parallel and Distributed Systems*, 3(2):179–193, 1992.

[8] J. Li and M. Chen. The data alignment phase in compiling programs for distributed-memory machines. *Journal of Parallel and Distributed Computing*, 13:213–221, 1991.

[9] Z. Li, P.-C. Yew, and C.-Q. Zhu. An efficient data dependence analysis for parallelizing compilers. *IEEE Transactions on Parallel and Distributed Systems*, 1(1):26–34, 1990.

[10] P. K. McKinley, H. Xu, and L. M. Ni. Efficient communication services for scalable architectures. Technical report, Dept. of Computer Science, Michigan State University, 1992.

[11] Lionel Ni. Private communication.

[12] M. Rosing, R. Schnabel, and R. Weaver. The DINO parallel programming language. *Journal of Parallel and Distributed Computing*, 13:30–42, 1991.

[13] M. Wolfe. *Optimizing Supercompilers for Supercomputers.* MIT Press, Cambridge, Mass., 1989.

APPENDICES

APPENDIX A

Sample Programs

```
      PROGRAM example1
C  This program contains a great deal of parallelism
      double precision f(100),x(100)
      do 100 i = 1 , 100
        x(i) = sqrt(dble(i)) * sin(dble(i))
 100  continue
      call func(f,x)
      do 110 i = 1, 10
        write(*,*) x(i), f(i), sin(x(i))
 110  continue
      end

      subroutine func(f,x)
```

```
      double precision f(100),x(100)
      double precision temp1(100), temp2(100), temp3(100)

      do 200 i = 1, 100
        temp1(i) = (sin(x(i)))**2 + (cos(x(i)))**2
        temp2(i) = (1/(sin(x(i))*sin(x(i))))
        temp3(i) = (cos(x(i))*cos(x(i)))/(sin(x(i))*sin(x(i)))
        f(i) = sin(x(i)) * (temp1(i)*temp2(i) - temp1(i)*temp3(i))
200   continue

      end
```

```
      PROGRAM example2

C   This program is naturally sequential

      double precision f,x(100),answer,deriv

      do 100 i = 1 , 100
        x(i) = sqrt(dble(i)) * sin(dble(i))
 100  continue

      call func1(100,f,x)

      answer = 0

      do 110 i = 1, 100
        answer = answer + sin(x(i))
        deriv = deriv + cos(x(i))
 110  continue

      write(*,*) f,answer

      end


      subroutine func1(n,f,x)

      integer n

      double precision f,x(n), temp1, temp2, temp3

      f = 0.0d0

      do 200 i = 1, n
        temp1 = (sin(x(i)))**2 + (cos(x(i)))**2
        temp2 = (1/(sin(x(i))*sin(x(i))))
        temp3 = (cos(x(i))*cos(x(i)))/(sin(x(i))*sin(x(i)))
        f = f + sin(x(i)) * (temp1*temp2 - temp1*temp3)
 200  continue

      end
```

```
      PROGRAM example3

C   This is a new version of program 2, with scalar variables
C   promoted to vectors.

      double precision f,x(100),answer,deriv

      do 100 i = 1 , 100

        x(i) = sqrt(dble(i)) * sin(dble(i))

 100  continue

      call func1(100,f,x)

      answer = 0

      do 110 i = 1, 100
        answer = answer + sin(x(i))
        deriv = deriv + cos(x(i))
 110  continue

      write(*,*) f,answer

      end



      subroutine func1(n,f,x)

      integer n

      double precision f,farray(101),x(n)
      double precision temp1(100), temp2(100), temp3(100)

      farray(1) = 0.0d0

      do 200 i = 1, n
        temp1(i) = (sin(x(i)))**2 + (cos(x(i)))**2
        temp2(i) = (1/(sin(x(i))*sin(x(i))))
```

```
      temp3(i) = (cos(x(i))*cos(x(i)))/(sin(x(i))*sin(x(i)))
      farray(i+1) = farray(i) + sin(x(i)) * (temp1(i)*temp2(i)
     +               - temp1(i)*temp3(i))
200   continue

      f = farray(101)

      end
```

APPENDIX B

ADIFOR-generated Code

```
      subroutine g$func$6(g$p$, n, f, g$f, ldg$f, x, g$x, ldg$x)

C       The ADIFOR-generated subroutine for program 1

C
C       Formal x is active.
C       Formal f is active.
C
        integer g$p$
        integer g$pmax$
        parameter (g$pmax$ = 100)
        integer g$i$
        double precision temp1bar
        double precision d$9bar
        double precision d$10
        double precision d$9
        double precision d$8
        double precision d$7
        double precision d$4bar
        double precision d$6
        double precision d$5
        double precision d$4
```

```
      double precision d$3
      double precision d$2
      double precision d$1
      double precision d$0
C
      integer i
      real sin
      real cos
      integer n
      double precision f(n), x(n), temp1, temp2, temp3
      double precision g$f(ldg$f, n), g$x(ldg$x, n), g$temp1(g$pmax$),
     * g$temp2(g$pmax$), g$temp3(g$pmax$)
      shared x,g$f,g$x
      integer ldg$f
      integer ldg$x
      if (g$p$ .gt. g$pmax$) then
        print *, 'Parameter g$p is greater than g$pmax.'
        stop
      endif
      do 99999, i = 1, n
C       temp1 = sin(x(i)) ** 2 + cos(x(i)) ** 2
        d$0 = x(i)
        d$1 = sin(d$0)
        d$3 = x(i)
        d$4 = cos(d$3)
        do 99994 g$i$ = 1, g$p$
          g$temp1(g$i$) = cos(d$0) * (2 * d$1) * g$x(g$i$, i) + ((-sin
     *(d$3) * (2 * d$4)) * g$x(g$i$, i))
99994     continue
        temp1 = d$1 ** (2) + (d$4 ** (2))
C       temp2 = (1 / (sin(x(i)) * sin(x(i))))
        d$0 = x(i)
        d$1 = sin(d$0)
        d$2 = x(i)
        d$3 = sin(d$2)
        d$4 = d$1 * d$3
        d$5 = 1 / d$4
        d$4bar = (-d$5 / (d$4))
        do 99993 g$i$ = 1, g$p$
          g$temp2(g$i$) = cos(d$0) * (d$4bar * d$3) * g$x(g$i$, i) + c
```

```
     *os(d$2) * (d$4bar * d$1) * g$x(g$i$, i)
99993     continue
          temp2 = d$5
C         temp3 = (cos(x(i)) * cos(x(i))) / (sin(x(i)) * sin(x(i)))
          d$0 = x(i)
          d$1 = cos(d$0)
          d$2 = x(i)
          d$3 = cos(d$2)
          d$5 = x(i)
          d$6 = sin(d$5)
          d$7 = x(i)
          d$8 = sin(d$7)
          d$9 = d$6 * d$8
          d$10 = d$1 * d$3 / d$9
          d$4bar = (1.0d0 / d$9)
          d$9bar = (-d$10 / (d$9))
          do 99992 g$i$ = 1, g$p$
            g$temp3(g$i$) = (-sin(d$0) * (d$4bar * d$3)) * g$x(g$i$, i)
     *+ ((-sin(d$2) * (d$4bar * d$1)) * g$x(g$i$, i)) + cos(d$5) * (d$9b
     *ar * d$8) * g$x(g$i$, i) + cos(d$7) * (d$9bar * d$6) * g$x(g$i$, i
     *)
99992     continue
          temp3 = d$10
C         f(i) = sin(x(i)) * (temp1 * temp2 - temp1 * temp3)
          d$0 = x(i)
          d$1 = sin(d$0)
          d$4 = temp1 * temp2 - temp1 * temp3
          temp1bar = -d$1 * (temp3)
          temp1bar = temp1bar + d$1 * temp2
          do 99991 g$i$ = 1, g$p$
            g$f(g$i$, i) = temp1bar * g$temp1(g$i$) + d$1 * temp1 * g$te
     *mp2(g$i$) + (-d$1 * (temp1) * g$temp3(g$i$)) + cos(d$0) * d$4 * g$
     *x(g$i$, i)
99991     continue
          f(i) = d$1 * d$4
200       continue
99999   continue
      end
```

```
      subroutine g$func1$6(g$p$, n, f, g$f, ldg$f, x, g$x, ldg$x)

C       The ADIFOR-generated subroutine for program 2

C
C       Formal x is active.
C       Formal f is active.
C
        integer g$p$
        integer g$pmax$
        parameter (g$pmax$ = 100)
        integer g$i$
        double precision temp1bar
        double precision d$9bar
        double precision d$10
        double precision d$9
        double precision d$8
        double precision d$7
        double precision d$4bar
        double precision d$6
        double precision d$5
        double precision d$4
        double precision d$3
        double precision d$2
        double precision d$1
        double precision d$0
        integer ldg$f
C
        integer i
        real sin
        real cos
        integer n
        double precision f, x(n), temp1, temp2, temp3
        double precision g$f(ldg$f), g$x(ldg$x, n), g$temp1(g$pmax$), g$
     *temp2(g$pmax$), g$temp3(g$pmax$)
        integer ldg$x
        if (g$p$ .gt. g$pmax$) then
          print *, 'Parameter g$p is greater than g$pmax.'
          stop
        endif
```

```
        f = 0.0d0
        do 99993 g$i$ = 1, g$p$
          g$f(g$i$) = 0.0d0
99993   continue
        do 99999, i = 1, n
C         temp1 = sin(x(i)) ** 2 + cos(x(i)) ** 2
          d$0 = x(i)
          d$1 = sin(d$0)
          d$3 = x(i)
          d$4 = cos(d$3)
          do 99992 g$i$ = 1, g$p$
            g$temp1(g$i$) = cos(d$0) * (2 * d$1) * g$x(g$i$, i) + ((-sin
     *(d$3) * (2 * d$4)) * g$x(g$i$, i))
99992     continue
          temp1 = d$1 ** (2) + (d$4 ** (2))
C         temp2 = (1 / (sin(x(i)) * sin(x(i))))
          d$0 = x(i)
          d$1 = sin(d$0)
          d$2 = x(i)
          d$3 = sin(d$2)
          d$4 = d$1 * d$3
          d$5 = 1 / d$4
          d$4bar = (-d$5 / (d$4))
          do 99991 g$i$ = 1, g$p$
            g$temp2(g$i$) = cos(d$0) * (d$4bar * d$3) * g$x(g$i$, i) + c
     *os(d$2) * (d$4bar * d$1) * g$x(g$i$, i)
99991     continue
          temp2 = d$5
C         temp3 = (cos(x(i)) * cos(x(i))) / (sin(x(i)) * sin(x(i)))
          d$0 = x(i)
          d$1 = cos(d$0)
          d$2 = x(i)
          d$3 = cos(d$2)
          d$5 = x(i)
          d$6 = sin(d$5)
          d$7 = x(i)
          d$8 = sin(d$7)
          d$9 = d$6 * d$8
          d$10 = d$1 * d$3 / d$9
          d$4bar = (1.0d0 / d$9)
```

```
          d$9bar = (-d$10 / (d$9))
          do 99990 g$i$ = 1, g$p$
            g$temp3(g$i$) = (-sin(d$0) * (d$4bar * d$3)) * g$x(g$i$, i)
     *+ ((-sin(d$2) * (d$4bar * d$1)) * g$x(g$i$, i)) + cos(d$5) * (d$9b
     *ar * d$8) * g$x(g$i$, i) + cos(d$7) * (d$9bar * d$6) * g$x(g$i$, i
     *)
99990     continue
          temp3 = d$10
C         f = f + sin(x(i)) * (temp1 * temp2 - temp1 * temp3)
          d$0 = x(i)
          d$1 = sin(d$0)
          d$4 = temp1 * temp2 - temp1 * temp3
          temp1bar = -d$1 * (temp3)
          temp1bar = temp1bar + d$1 * temp2
          do 99989 g$i$ = 1, g$p$
            g$f(g$i$) = g$f(g$i$) + temp1bar * g$temp1(g$i$) + d$1 * tem
     *p1 * g$temp2(g$i$) + (-d$1 * (temp1) * g$temp3(g$i$)) + cos(d$0) *
     * d$4 * g$x(g$i$, i)
99989     continue
          f = f + d$1 * d$4
200       continue
99999   continue
      end
```