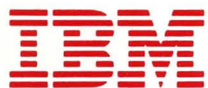


# Vectorization and Vector Migration Techniques

Technical Bulletin

# Vectorization and Vector Migration Techniques

Author: **David B. Soll**
Applications Analysis and
Implementation Techniques
Data Systems Division
Kingston, New York

Editor: **Ralph Stephens**
IBM Technical Education Center
Chicago, Illinois

**Technical Bulletin**

## Acknowledgement

For their assistance and support of this technical bulletin, the author would like to express his appreciation to the following:

- Paul Dorn, Washington Systems Center, North Central Mktg. Division
- John R. Ehrman, Santa Teresa Laboratory, General Products Division

The information contained in this document has not been submitted to any FORMAL IBM TEST AND IS DISTRIBUTED ON AN "AS IS" basis without any warranty either expressed or implied. The use of this information or the implementation of any of these techniques is a customer responsibility and depends on the customer's ability to evaluate and integrate them into the customer's operational environment. While each item may have been reviewed by IBM for accuracy in a specific situation, there is no guarantee that the same or similar results will be obtained elsewhere.

**First Edition, June 1986**

All rights reserved.

In this document, any references made to an IBM licensed program are not intended to state or imply that only IBM's licensed program may be used; any functionally equivalent program may be used instead.

It is possible that this material may contain reference to, or information about IBM products (machines and programs), programming or services that are not announced in your country. Such references or information must not be construed to mean that IBM intends to announce such IBM products, programming, or services in your country.

Publications are not stocked at the address given below; requests for IBM publications should be made to your IBM representative or the the IBM branch office serving your locality.

A form for reader's comments is provided at the back of this publication. If the form has been removed, comments may be addressed to: IBM Corporation, Chicago Technical Education Center, One IBM Plaza, Chicago, Illinois 60611.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation whatever. You may, of course, continue to use the information you supply.

© Copyright International Business Machines Corporation 1986

# Preface

Techniques for migrating Scientific & Engineering Fortran applications to the System/370 Model 3090 Vector Facility are presented. The development of a vector migration strategy to exploit the System/370 Model 3090 Vector Facility is described. Local vectorization techniques and program-wide modifications which enhance the vector content of applications are discussed. The use of the VS Fortran Version 2 Compiler and Library, The Engineering and Scientific Subroutine Library and related products to exploit the System/370 Model 3090 Vector Facility are also introduced.

This document was authored by David B. Soll of the Application Analysis Group, a part of the Scientific and Engineering Products organization of the IBM Data Systems Division. In order to determine the best ways of using the System/370 Model 3090 Vector Facility for engineering and scientific work, this group has analyzed numerous applications. This analysis is the basis for this technical bulletin.

The migration of applications using the Vector Facility, the kinds of problems it is designed to solve, and the Fortran language support provided to the user have also been studied.

The objectives in this technical bulletin are to:

- develop and describe techniques for effective use of the IBM 3090 Vector Facility;
- share application migration experiences with users;
- provide feedback into the product development cycle for IBM processors.

This technical bulletin discusses this information from the user's point of view, and introduces application techniques that can help exploit the Vector Facility. Also included are:

- illustrations of programming considerations that can hinder fully exploiting the Vector Facility,
- a discussion of how to avoid some of the obstacles to using the Vector Facility effectively,

- suggestions on ways to reduce the effort in performing the migration process by using existing packages,
- recommendations on the kinds of tools to use.

# Contents

**1.0 Introduction** . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **1**

**2.0 Application Migration** . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **3**
2.1 What is the Goal of Application Migration? . . . . . . . . . . . . . . . . . . 3
2.2 Vector Migration: Application Selection . . . . . . . . . . . . . . . . . . . . . 5
2.2.1 Selection Criteria . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 5
2.2.2 Virtual Storage Compatibility . . . . . . . . . . . . . . . . . . . . . . . . . . 5
2.2.3 Algorithm Analysis . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 6
2.2.4 Engineering and Scientific Subroutine Library . . . . . . . . . . . . . 7
2.3 Application Migration: Strategy . . . . . . . . . . . . . . . . . . . . . . . . . . . 7
2.4 Vector Migration Methodology . . . . . . . . . . . . . . . . . . . . . . . . . . . . 8
2.5 Application Migration: Initial Steps . . . . . . . . . . . . . . . . . . . . . . . . 8
2.5.1 Language Conversion . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 9
2.6 Analyzing Inhibitors to Migration . . . . . . . . . . . . . . . . . . . . . . . . . 9
2.7 Characterizing the Application . . . . . . . . . . . . . . . . . . . . . . . . . . . . 9
2.7.1 Where Vector Content May Be Found . . . . . . . . . . . . . . . . . . . 10
2.7.2 How Vector Content May Be Expressed . . . . . . . . . . . . . . . . . 11
2.7.3 Style . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 11
2.7.4 An Example . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 12
2.8 Measuring the Application . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 14
2.9 Vector Compilation . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 15
2.10 Scope of Application Modification . . . . . . . . . . . . . . . . . . . . . . . . 16
2.11 Level of Effort . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 16

**3.0 Overview of Vectorization Concepts** . . . . . . . . . . . . . . . . . . . . **19**
3.1 The Basic Unit of Vectorization: The DO Loop . . . . . . . . . . . . . . . 19
3.2 The Basic Action of Vectorization: Loop Sectioning . . . . . . . . . . . . 19
3.3 Loop Selection . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 20
3.3.1 Vectorizing Outer DO Loops . . . . . . . . . . . . . . . . . . . . . . . . . . 21
3.4 Data Independence . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 23
3.5 Recurrences . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 24
3.6 Indirect Addressing . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 26
3.7 The Stride of a Vector . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 27
3.8 Sources of Numerically Different Results . . . . . . . . . . . . . . . . . . . 29
3.8.1 Vectorization of Reduction Operations . . . . . . . . . . . . . . . . . . 29
3.8.2 Vectorization of Library Intrinsic Functions . . . . . . . . . . . . . . 31
3.8.3 Fortran-66 and Fortran-77 Execution of DO Loops . . . . . . . . . 31

**4.0 Local Vectorization Considerations** . . . . . . . . . . . . . . . . . . . . . **33**
4.1 Not All DO Loops Are Appropriate for Vectorization . . . . . . . . . . 33
4.2 Not All DO Loops Are Well Suited for Vectorization . . . . . . . . . . . 33
4.3 Not All Loops Are DO Loops . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 34
4.4 Some Loops Should Be Written as DO Loops . . . . . . . . . . . . . . . . . 34

4.5 Some DO Loops Iterate Too Few Times .......... 35
4.6 Some Loop Iteration Counts Are Unknown .......... 36
4.7 Some DO Loops Cannot Be Vectorized .......... 36
4.8 Summary .......... 37

**5.0 Local Vectorization Techniques .......... 39**
5.1 Stride .......... 40
5.1.1 Stride and Recurrences .......... 41
5.1.2 Stride Minimization .......... 42
5.2 Data Organization .......... 43
5.2.1 Reorganizing Data to Improve Stride .......... 43
5.2.2 Using EQUIVALENCE to Improve Vector Length .......... 44
5.2.3 Data Restructuring .......... 45
5.3 Temporary Variables .......... 46
5.3.1 Scalar Temporaries .......... 46
5.3.2 Scalar Array Element References .......... 47
5.4 Subscript Considerations .......... 48
5.4.1 Subscripts With Constant Increments .......... 48
5.4.2 Computed Auxiliary Subscripting Variables .......... 48
5.4.3 Linearized Multi-Dimensional Subscripts .......... 49
5.4.4 Auxiliary Subscripts with Unknown Increment .......... 50
5.5 Recurrence, Part 2 .......... 50
5.5.1 Hiding Recurrences .......... 51
5.6 Unrolling Loops .......... 52
5.7 Loop Segmentation .......... 53
5.8 Statement Reordering .......... 54
5.9 Loop Distribution .......... 55
5.10 Indirect Addressing .......... 56
5.11 Conditional Operations .......... 56
5.11.1 Conditional Operations and IF Conversion .......... 57
5.11.2 Writing Conditional Code .......... 58
5.11.3 Improving Conditional Code .......... 59
5.12 Data Dependent Loops .......... 61
5.13 Loops Containing External References .......... 63
5.14 Loops Containing Input/Output Statements .......... 64
5.15 Restating an Algorithm .......... 64
5.16 Vector Optimizations .......... 66
5.16.1 Vector Sub-Sections .......... 67
5.16.2 Indirect Addressing .......... 68
5.16.3 Improving Vector Density .......... 68
5.17 Local Vectorization Techniques: Summary .......... 70

**6.0 Global Migration Considerations .......... 71**
6.1 Global Restructuring .......... 72
6.2 Incorporating Loops Across Modules .......... 72
6.3 Changing the Solution Method .......... 74

**7.0 Summary .......... 75**

**Appendix A. Glossary of Terms and Concepts .......... 77**

**Appendix B. References .......... 81**

# Figures


1. Performance Improvement vs. Vectorization . . . . . . . . . . . . . . . . . . 4
2. Typical Coding of Matrix Multiplication . . . . . . . . . . . . . . . . . . . . 6
3. Revised, Efficient Form of Matrix Multiplication . . . . . . . . . . . . . . 7
4. Solution Paths for General Problems . . . . . . . . . . . . . . . . . . . . . . 12
5. Application Organization: Sequential . . . . . . . . . . . . . . . . . . . . . 13
6. Application Organization: Vector . . . . . . . . . . . . . . . . . . . . . . . . 14
7. Performance Gain Over Time, and Level of Effort Required . . . . . . 17
8. Basic Unit of Vectorization: the DO Loop . . . . . . . . . . . . . . . . . . . 19
9. A Vectorizable Loop Before Sectioning . . . . . . . . . . . . . . . . . . . . 20
10. A Vectorizable Loop After Sectioning . . . . . . . . . . . . . . . . . . . . . 20
11. Loop Selection: Original Code . . . . . . . . . . . . . . . . . . . . . . . . . . 21
12. Loop Selection: Equivalent Code . . . . . . . . . . . . . . . . . . . . . . . . 21
13. Loop Selection: Vectorized Code . . . . . . . . . . . . . . . . . . . . . . . . 22
14. Loop Selection: Vectorizing an Outer Loop . . . . . . . . . . . . . . . . . 22
15. Schematic Form of Vectorized Outer Loop . . . . . . . . . . . . . . . . . . 22
16. Simple Example of a Data Independent Loop . . . . . . . . . . . . . . . . 23
17. Execution Order of Data Independent Loop . . . . . . . . . . . . . . . . . 23
18. Simple Example of a Data Dependent Loop . . . . . . . . . . . . . . . . . 24
19. Execution Order of Data Dependent Loop . . . . . . . . . . . . . . . . . . 24
20. Loops Demonstrating Recurrences . . . . . . . . . . . . . . . . . . . . . . . 24
21. Loop Demonstrating a Implicit Recurrence . . . . . . . . . . . . . . . . . . 25
22. A Loop With No Recurrence . . . . . . . . . . . . . . . . . . . . . . . . . . . 25
23. Loop With No Recurrence . . . . . . . . . . . . . . . . . . . . . . . . . . . . 25
24. Execution Order of Loop with No Recurrence . . . . . . . . . . . . . . . 26
25. Similar Loop, Now Containing a Recurrence . . . . . . . . . . . . . . . . 26
26. Example of Indirect Addressing . . . . . . . . . . . . . . . . . . . . . . . . . 26
27. Vectorizable Indirect-Addressing Loops . . . . . . . . . . . . . . . . . . . . 27
28. Non-Vectorizable Indirect-Addressing Loop . . . . . . . . . . . . . . . . . 27
29. A DO Loop With Stride-1 Memory References . . . . . . . . . . . . . . . 27
30. DO Loops With Stride 1 and Stride 50 . . . . . . . . . . . . . . . . . . . . 28
31. Identical DO Loops With Different Strides and Counts . . . . . . . . . 28
32. Summing Elements of a Vector . . . . . . . . . . . . . . . . . . . . . . . . . 29
33. Dot Product of Vectors A and B . . . . . . . . . . . . . . . . . . . . . . . . . 30
34. Partial Summation in Reduction Operations . . . . . . . . . . . . . . . . . 30
35. A DO Loop With Unknown Control Parameters . . . . . . . . . . . . . . . 31
36. A DO Loop Inappropriate for Vectorization . . . . . . . . . . . . . . . . . . 33
37. A DO Loop Not Well Suited for Vectorization . . . . . . . . . . . . . . . . 34
38. A Hand-Coded Loop . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 34
39. A Hand-Coded Backward Loop . . . . . . . . . . . . . . . . . . . . . . . . . . 35
40. A Standard Backward DO Loop . . . . . . . . . . . . . . . . . . . . . . . . . 35
41. A DO Loop With Small Iteration Count . . . . . . . . . . . . . . . . . . . . 35
42. A DO Loop With Unknown Iteration Count . . . . . . . . . . . . . . . . . . 36
43. DO Loop With REAL Index . . . . . . . . . . . . . . . . . . . . . . . . . . . . 36

44. DO Loop With Index Converted to INTEGER .................. 37
45. Loops With Different Strides .................................. 41
46. Loop With Dependence in One Dimension ..................... 41
47. Loop With Many Long-Stride References ...................... 42
48. Copying Data to Minimize Long-Stride References ............ 42
49. Inappropriate Data Organization for Vectorization ........... 43
50. Data Organization More Appropriate for Vectorization ......... 44
51. Loops With 3-Dimension Arrays ................................ 44
52. Loops With 2-Dimension EQUIVALENCEd Arrays ............. 45
53. Poor Loop and Data Structure .................................. 45
54. Improved Loop and Data Structure ............................. 46
55. Scalar Temporary Becomes Vector Temporary ................ 46
56. Scalar Array Element as a Temporary ......................... 47
57. Auxiliary Array Subscript With Constant Increment ........... 48
58. Computed Array Subscript ...................................... 49
59. Eliminating An Auxiliary Subscripting Variable .............. 49
60. Auxiliary Subscripting Variable With Unknown Increment ...... 50
61. Same Loop with Constant Auxiliary Subscript ................ 50
62. Loops Not Containing a Recurrence ........................... 51
63. Subscript Relationships in Previous Example ................ 51
64. Loop With Unrolling and Without .............................. 52
65. Loop Unrolled Along Non-Vector Dimension ................... 52
66. Nested Loops Available for Segmentation ..................... 53
67. Loops After Segmentation ...................................... 53
68. Loops Before and After Segmentation ......................... 54
69. Removing an Order Dependence ................................ 54
70. Loop Suitable for Distribution ................................ 55
71. Original Loop Is Split Into Two Loops ........................ 55
72. Indirect Addressing With a Temporary Vector ................ 56
73. Example of a Conditional Operation ........................... 57
74. Loops With Conditional Operations ........................... 57
75. Loop With Conditional Control ................................. 57
76. Loop With Data Dependence .................................... 58
77. Loop With Conditional Control ................................. 58
78. Loop Containing a Condition .................................. 58
79. Control Dependence Changed to Data Dependence ............. 59
80. Data Dependence With Different Conditions ................... 59
81. Loops With Control Dependence ............................... 60
82. Computation on 3-Dimensional Grid ........................... 60
83. Loops With Modified Control Dependence ..................... 61
84. Data Dependent Loop With Branch Out ......................... 61
85. Vectorizable Version of Data Dependent Loop ................ 62
86. Loop Containing a CALL Statement ........................... 63
87. Loop Containing WRITE Statement ............................ 64
88. Loop With WRITE Statement Moved ............................ 64
89. All Six Ways to Multiply Two Matrices ........................ 65
90. Visualizing Matrix Multiplication .............................. 66
91. Subsets of Vectors ............................................. 67
92. Eliminating Subsets of Vectors ................................ 67
93. Counting Conditional Selections ............................... 69
94. Compressing Vector A Into Vector X .......................... 69
95. Expanding Vector X Into Vector A, Zero Filler ............... 69
96. Expanding Vector X Into Vector A, With Replacement ......... 70

97. Loops Distributed Across Modules ........................ 73
98. Loops Incorporated Into a Single Module .................... 73

# 1.0 Introduction

This technical bulletin presents one view of the process of migrating Fortran application programs to exploit the System/370 Model 3090 Vector Facility.

The discussion of the application migration process is divided into the following parts:

1. The first part involves establishing a migration methodology. This begins with determining the objectives of the migration effort, and setting realistic goals for the resulting (anticipated) performance gains. Next, a migration strategy is developed from a three stage process. This process consists of

   a. characterizing the application program,

   b. recompiling using the VS Fortran Version 2 vectorizing compiler, and

   c. analyzing the results of these two activities to formulate the migration strategy.

   This strategy may consist of a combination of local and global program modifications which enhance the "vector content" of the application.

2. The second part reviews key vectorization concepts, and how they apply to the System/370 Model 3090 Vector Facility and the VS Fortran Version 2 Vectorizing Compiler.

3. The third part includes descriptions of a number of local vectorizations which may be required as part of the migration strategy. A series of examples is used to illustrate programming practices which make an application's possibilities for vector execution — its "vector content" — as visible as possible to the vectorizing compiler.

4. The fourth part introduces some of the program-wide ("global") aspects of a migration effort. The scope of this discussion will be limited to those aspects of overall program modification concerned with module or logic organization, and those aspects of matching data structures which promote vectorization. No attempt is made to analyze the vectorization aspects of the many alternative methods of solution or different numerical techniques available.

A brief summary then presents an overview of the migration process.

In the rest of this technical bulletin, we will assume a working familiarity with the Fortran language, particularly as implemented by the VS Fortran Compiler and Library. No previous experience with vector computation is assumed.

In the following discussion, we will be using terminology that may not be familiar. Some key terms are defined in Appendix A, "Glossary of Terms and Concepts" on page 77. An overview of the characteristics of the System/370 Model 3090 Vector Facility and the VS Fortran Version 2 Vectorizing Compiler are presented in 3.0, "Overview of Vectorization Concepts" on page 19.

# 2.0 Application Migration

The application migration methodology is a process of discovery by which the "vector content" of an application may be found and be made visible to the compiler, and used to improve the performance of the application. An important aspect of this process is the determination of a strategy by which this performance will be improved, and the time to execute the application correctly will be reduced. Part of the formulation of this strategy is to realistically assess the scope of the migration activity, the potential performance improvement, and the level of effort required.

Our objectives, therefore, are to investigate

- techniques for exploiting the vector facility,
- methods of improving vector execution performance,
- where to look for vectorizable code, and
- some approaches to using the Vector Facility effectively.

Another objective is to describe some of the Application Analysis group's experiences in migrating different types of programs to the vector hardware and making them run efficiently. This process is known as "enabling vector usage", or sometimes simply as "enabling".

## 2.1 What is the Goal of Application Migration?

A simple theoretical model (known as "Amdahl's Law") gives an approximate indication of the maximum possible performance improvement (P) when a program having a given vectorization ratio (V) is executed on hardware having a specified ratio of vector to scalar computation speeds (a): $P = 1/(1\text{-}V+V/a)$. The family of curves in Figure 1 on page 4 gives an indication of the maximum level of performance improvement (P) we might expect, given various scalar-to-vector performance ratios (a), for each degree of vectorization (V). The degree of vectorization is the percentage of the application's scalar CPU time which may be migrated to execute on the vector hardware.

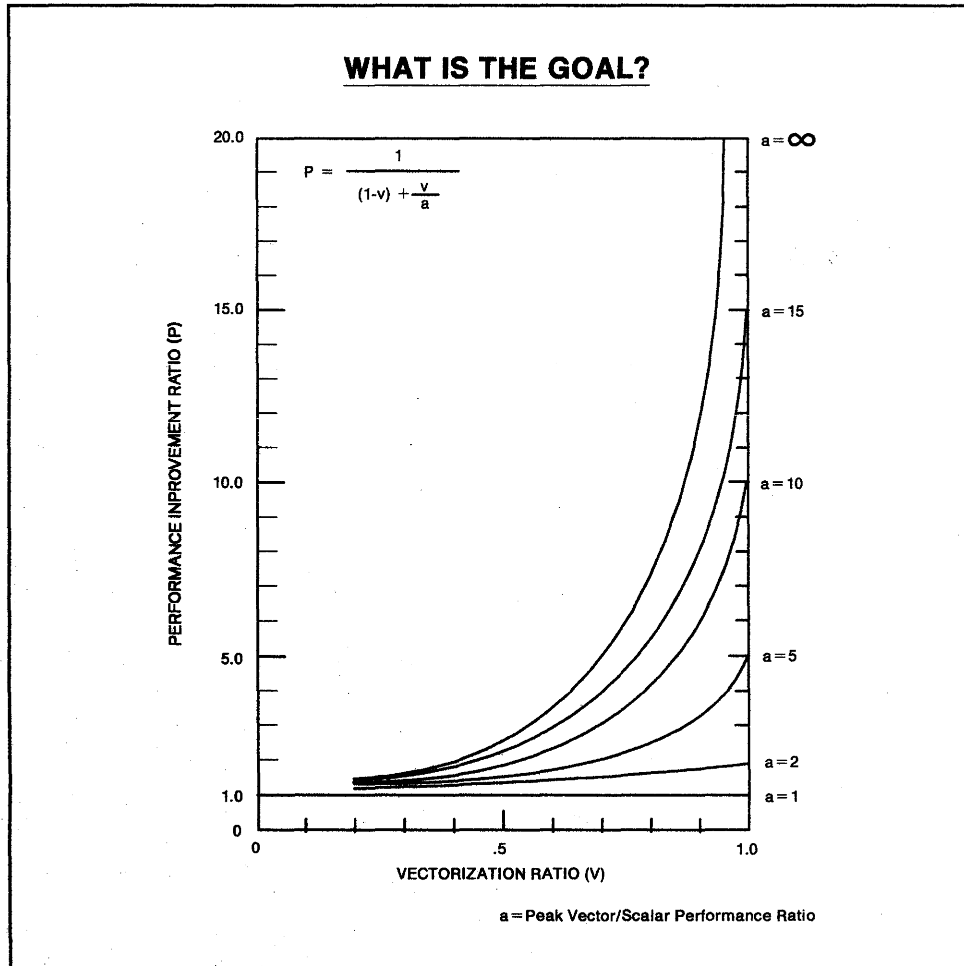



**Figure 1. Performance Improvement vs. Vectorization**

Several observations should be made about these curves.

1. It can be seen that even if an application can obtain an 80% vectorization ratio (V), and the speed of vector execution is infinitely fast ("a" is infinite), then the application can only realize a factor of 5 performance improvement! This is because 20% of the application's original (scalar) CPU time is still spent executing in scalar mode on the scalar hardware. Thus, you should not anticipate substantial performance improvements unless your application is (a) highly vectorizable, and (b) the vector facility on which it executes is capable of high vector-to-scalar speed ratios.

2. It is very difficult to characterize accurately the ratio "a" of vector to scalar execution speeds. It is sometimes tempting to use known quantities like hardware cycle times, but these numbers often have little to do with the actual performance of real applications on a vector facility.

3. The vectorization ratio (V) must be understood with some care. As we will see in the following discussions, there are situations where the

choice of scalar or vector execution must be made judiciously; it is *not* sufficient simply to push as much of an application's instruction stream as possible onto the Vector Facility in order to get the "performance improvement" implied by the curves in Figure 1 on page 4.

The goal, therefore, is to

- focus on those parts of the applications that take the most CPU time,
- convert them to enable execution in vector mode (while recognizing that some portion of the code will remain scalar), and
- keep realistic expectations for performance improvements.

## 2.2 Vector Migration: Application Selection

The vector migration effort can be an ongoing activity, or can be limited to performing a restricted set of tasks in a reasonable time frame to obtain a cost effective result for the effort invested. It is clear, then, that we must be aware of the tradeoffs involved in choosing which application programs are to be migrated to a form which better exploits vector computation. We must ask, "How do we select an application program for vectorization?"

### 2.2.1 Selection Criteria

The first criterion for selection might be to ask "is it a long running, frequently used program?" It is probably not cost effective to spend a lot of effort on a short running program which is used infrequently.

The next criterion involves "CPU intensity". In general, the greatest performance improvement will be realized from an application which has a high CPU utilization rate. However, this should not exclude an application with significant I/O content, since the reorganization performed to improve vectorization often has the added benefit of reducing I/O activity. We must keep in mind that only the computational content of the code will directly benefit from vectorization.

While it is frequently assumed that high CPU utilization implies high floating point content, it should be noted that the System/370 Model 3090 Vector Facility can operate on integer and logical vectors as well as floating point vectors. However, the speed improvement for programs containing a high proportion of integer and logical operations is sometimes not as great as for floating point operations.

### 2.2.2 Virtual Storage Compatibility

"Virtual storage compatibility" for vector applications involves essentially the same considerations encountered when virtual storage is used for scalar applications: data references should be *localized* as much as possible. For array references, this generally means that the data should be used in "storage order" as much as possible.

Since one major benefit of the Vector Facility is that a single vector instruction performs an operation on a vector of multiple data elements, any needed memory accesses are most efficient when the vector of data elements is stored in contiguous memory locations.

As noted above, most of the steps taken to exploit the Vector Facility involve good programming practices rather than specific techniques required for exploiting specific vector hardware. When the user has finished the process of vectorization, he generally has higher quality scalar code, as well as having used "good" vector-enabling techniques.

### 2.2.3 Algorithm Analysis

There is considerable knowledge and experience in user installations to suggest that certain algorithms are more vectorizable than others, and are more appropriate for vector execution. Although it is possible to replace certain algorithms, this replacement must be approached with some care to ensure that this is the correct action. Sometimes, the replacement algorithm is just a restatement of the original algorithm, applied to the data in a different order. And, sometimes the replacement algorithm, although it is more vectorizable, unfortunately requires more iterations to achieve the same numerical result.

The following simple example illustrates how a restatement of the original algorithm can help. Multiplication of two matrices A and B to give a product matrix C is typically written in the form shown in Figure 2.

```
      DO 1 I = 1, M
        DO 1 J = 1, P
          C(I,J) = 0.0
          DO 1 K = 1, N
            C(I,J) = C(I,J)+A(I,K)*B(K,J)
   1  CONTINUE
```

**Figure 2. Typical Coding of Matrix Multiplication**

In fact, this is only one of *six* possible ways to permute the sequence of DO statements in writing this simple computation. (Many other sophisticated variations are possible, which we will ignore for now.) As discussed in 5.15, "Restating an Algorithm" on page 64, there is a way to revise the nest of loops in Figure 2 which yields better performance. This is shown in Figure 3 on page 7.

```
      DO 2 J = 1, P
        DO 1 I = 1, M
          C(I,J) = 0.0
    1   CONTINUE
        DO 2 K = 1, N
          DO 2 I = 1, M
            C(I,J) = C(I,J)+A(I,K)*B(K,J)
    2 CONTINUE
```

**Figure 3. Revised, Efficient Form of Matrix Multiplication**

It is beyond the scope of the discussions in this technical bulletin to include considerations relating to modifying the mathematics or solution techniques applied to the many types of engineering and scientific problems which may benefit from vectorization. The application developer probably has had considerable experience with the type of algorithms he is working with, since he had to make the original choices matching the solution technique or algorithm to the problem. Therefore we will note that a great deal of published material[1] is available which contains many references to this kind of information, and restrict our discussions to those cases in which the solution technique is retained, but perhaps the algorithm is modified to use the data in a different order.

### 2.2.4 Engineering and Scientific Subroutine Library

As we discuss the migration process, we will continue to look for opportunities to replace an existing (scalar) algorithm with one of the highly tuned vectorized versions available in the Engineering and Scientific Subroutine Library (ESSL). For[2] example, a Fast Fourier Transform (FFT) written in Fortran might be replaced by calls to the high-performance FFT routines in ESSL, rather than spending the effort that would be needed to analyze, rewrite, and tune the scalar Fortran version.

## 2.3 Application Migration: Strategy

The combination of all of these various approaches, which includes both local vectorization and some more global considerations, as well as looking for opportunities to make use of the ESSL vector subroutines, leads to the general question: "How can we form a strategy for migrating an application program to exploit the vector facility?".

To form this strategy, it is helpful to ask these questions:

- Is the code manageable?

---

[1] See *REFERENCES*, Appendix B.

[2] See the *IBM Engineering and Scientific Subroutine Library* Guide and Reference (Form Number GC23-0182).

- Can the user accomplish the migration task in a reasonable time?
- How much effort has to be put into the migration task?
- Does the user understand the code?
- Does the user have to make a decision about changing the algorithm or not?
- Can the user work within a few modules, or does he have to restructure the program?
- Are pre-packaged vectorized functions available for the user's purposes?
- Will localized modifications to the program be sufficient?

These are some of the considerations the user could deal with even before submitting his program to the compiler. Thus, the application migration process requires the user to understand his application. He needs the answers to these questions to properly interpret the results of the compiler vectorization, and as an aid to the formulation of a vector migration strategy.

## 2.4 Vector Migration Methodology

The migration process does not have to be established for each individual application. Although the process is application dependent, many applications will have some characteristics in common. Similarities are found within an industry or class of problem, and within specific solution techniques that occur over and over again. These similarities may be exploited by applying the experience of other migration efforts to establish general guidelines for the migration of a given application.

The scope of the migration methodology has been limited in this technical bulletin to the migration of existing applications by finding the code and data organization which best expresses the vector content of the problem without replacing the solution method or technique.

The task for the user then, is to determine, for the type of solution technique involved, what type of organization is most efficient for vectorization, and what kind of algorithm is being used. Some algorithms imply a certain amount of data independence, not just from the algorithm itself, but also from the underlying mathematics and/or the physical problem being solved.

## 2.5 Application Migration: Initial Steps

The simplest and easiest approach is for the user to simply apply the Fortran vectorizing compiler to the application, and let it vectorize "everything". If the application code and data are appropriately organized, the desired performance improvements may become available with no further effort being needed.

Of course, this type of migration does not require much explanation. In practice, however, it is most usually found that the vectorization process is

application dependent, particularly when the application was not originally designed for vector execution. This technical bulletin describes some of the more challenging aspects of migrating scalar Fortran application code to vectorizable code, where a simple recompilation with VS Fortran Version 2 is not sufficient.

### 2.5.1 Language Conversion

A convenient first step in the migration process is to move all of the application source code to the standard Fortran-77 language base represented by VS Fortran Version 2, LANGLVL(77). Although this conversion is not required (VS Fortran Version 2 will accept and vectorize Fortran-66, VS Fortran LANGLVL(66) constructs), it is recommended that the code be converted to the standard language level before making modifications for vectorization.

To assist in this effort, a Language Conversion Program (LCP) is available[3].

## 2.6 Analyzing Inhibitors to Migration

The process of migration now proceeds through several stages. Beginning with the Fortran-77 base, we will (1) characterize the application in several ways to be described shortly, (2) apply the VS Fortran Version 2 Vectorizing Compiler to the source code, and (3) analyze the results. We must then decide whether the simple vectorization results are acceptable as is, or whether there are inhibitors to further vectorization which must be overcome.

If these inhibitors are well localized (e.g., restricted to a single routine), then local recoding may be all that is required. If, however, it can be determined that the basic inhibitor to vectorization is the logical organization of the program's modules, or even the organization of its data, then we may begin to formulate the migration strategy previously mentioned. We will now discuss each of these steps to illustrate the activities involved.

## 2.7 Characterizing the Application

Characterizing the application involves discovering a number of things the user may already know about the application, such as

- What solution technique is used, and will it vectorize well?
- What kind of vector inhibitors exist in the program?
- Are the inhibitors related to

---

[3] See the *IBM Fortran Language Conversion Program General Information* manual (form number GC23-0154) for additional information on the LCP product.

- the syntactic style (the way in which the loops are expressed),
- the complexity of the indexes within those loops, or
- is the inhibitor the overall organization of the code?

(When these things are not already known, some of the techniques to be described will help in discovering them.)

Information about the solution technique provides insight into the type of modification which might be performed, without actually changing the technique itself. Other observations regarding the style and structure of the code help to set expectations for the migration effort, decide on the potential for using ESSL routines, and establish the potential for reordering the algorithm to improve its vector content.

For example, the original programmer may have masked the program's inherent data independence[4] by over-modularizing the program. That is, the program may have been organized in such a way that computations which might be performed on vectors of operands are spread over many subroutines, each of which operates on a single scalar datum at a time. When the program is executing in "sequential" (scalar) mode this may not have made any difference. For vector exploitation, however, it is recommended that the user review how both the program's logic and the data were organized, and how the program's data addressing patterns were associated with the program's instructions.

### 2.7.1 Where Vector Content May Be Found

When the original problem was analyzed, using a specific computational style may not have been particularly important. It is observed that most scientific and engineering applications have some vector content. In order to appreciate the degree to which a specific application can utilize the Vector Facility, the user has to know something about the original problem, how the problem has been modeled, and how that model has been expressed.

Most of today's complex problems cannot be solved exactly, but must be represented approximately. The resulting approximate mathematical solution is then represented by an approximate numerical solution. This process of refinement yields a tractable computational solution to the original physical problem which, it is hoped, closely resembles reality. At each stage of this refinement, the trend is towards more simplifications which will permit valid solutions to be found. These simplifications, in turn, involve such practices as uncoupling of physical effects or mathematical equations, independent treatment of processes (quasi-static, quasi-stationary) and the like. Such uncoupling or separation of effects provides the data independence which permits vectorization, in addition to the natural independence of effects present in the original problem.

In order to maximize the vectorization potential of an application program, then, it is our task to avoid hiding the inherent data independence present

---

4 See 3.4, "Data Independence" on page 23.

during the translation from the model of the problem to the mathematical description, to the numerical approximation, and finally to the expression of that numerical solution in Fortran.

### 2.7.2 How Vector Content May Be Expressed

The migration strategy can be partially formed by assessing the potential for vectorization present in each of several classes of solution methods. Figure 4 on page 12 illustrates the solution process for a general problem, showing three possible paths which result in an application program.

Along the left-hand path, a class of explicit or direct solution techniques is indicated. As a group, they have a certain amount of similarity in terms of the way of expressing the algorithms, methods of handling data and techniques of writing code. The data independence required for vectorization is usually expressed in the physical space of the problem, such as in the relationship between parameters evaluated on a physical grid. There is a degree of freedom in selecting an array (grid) dimension for vectorization presented by this class of solutions which may not be present in other classes.

The right-hand path depicts the class of implicit solutions, including large systems of equations. Along this path, the data independence is expressed in the relationships between the rows and columns of the matrices being manipulated, according to the rules of the matrix or linear algebra technique being used. The middle path indicates a mixed situation.

Thus the visibility of the inherent vector content of the problem must be maintained at each of the several stages of analysis, from the original statement of the problem to the final expression in FORTRAN.

At the bottom of the figure is a box labeled "machine tuning", which will be discussed last in this "top-down" process.

### 2.7.3 Style

For scalar machines, it usually did not matter how the program's *code* was organized as the user migrated his application from running in fixed, real memory to running in virtual storage. However, he did have to consider both the way the *data* was organized and the way it was referenced. This resulted in adopting a style of programming where the user organized and addressed his data to account for the program's behavior when the data was in virtual storage.

Style also counts in writing good vector code. However, it may be a more important factor on vector machines than on scalar machines, because "style" includes not only the micro-scale — the individual program loops — but the macro-scale — the way in which the program is designed. Whether the user is designing a new application or migrating an existing one, he should keep in mind the idea of maximizing the use of storage-order addressing. This should also help to organize the program so that it vectorizes well.

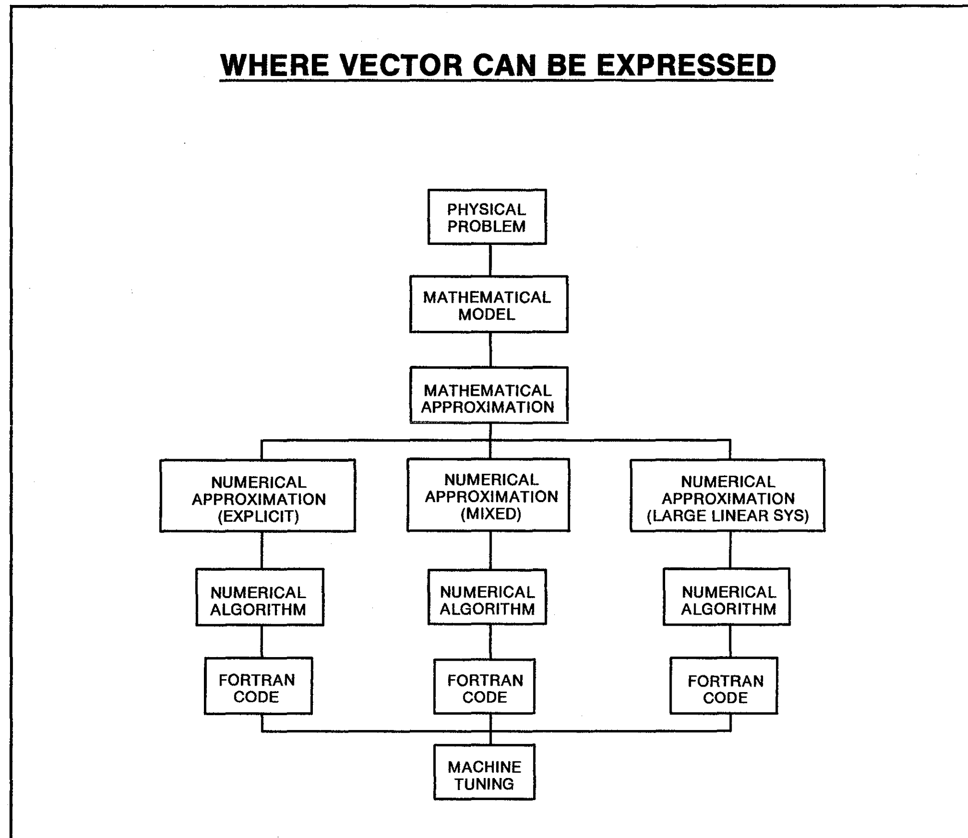



**Figure 4. Solution Paths for General Problems**

### 2.7.4 An Example

Suppose we have an application which performs a simulation, in the order of what happens at a single point (in a 3-dimensional space, perhaps) with many functions, many physical phenomenon, and many actions.

The program may originally have been designed for a sequential machine, and the user may have modularized this program so that each routine computed the results of one of such actions at a point; when all actions at a single point were completed, the process was repeated in a cyclic manner. This situation is illustrated by the simplified flow diagram in Figure 5 on page 13.

What happens to one point may be happening to many, or indeed to all the points, and this is one clue telling us where we may look to uncover the required data independence. That is, each routine or function may be doing the same thing to each point, but because the program was structured to perform the computations on a "point-by-point" basis, the compiler will not be able to detect the fact that the same operations are being performed on many points.

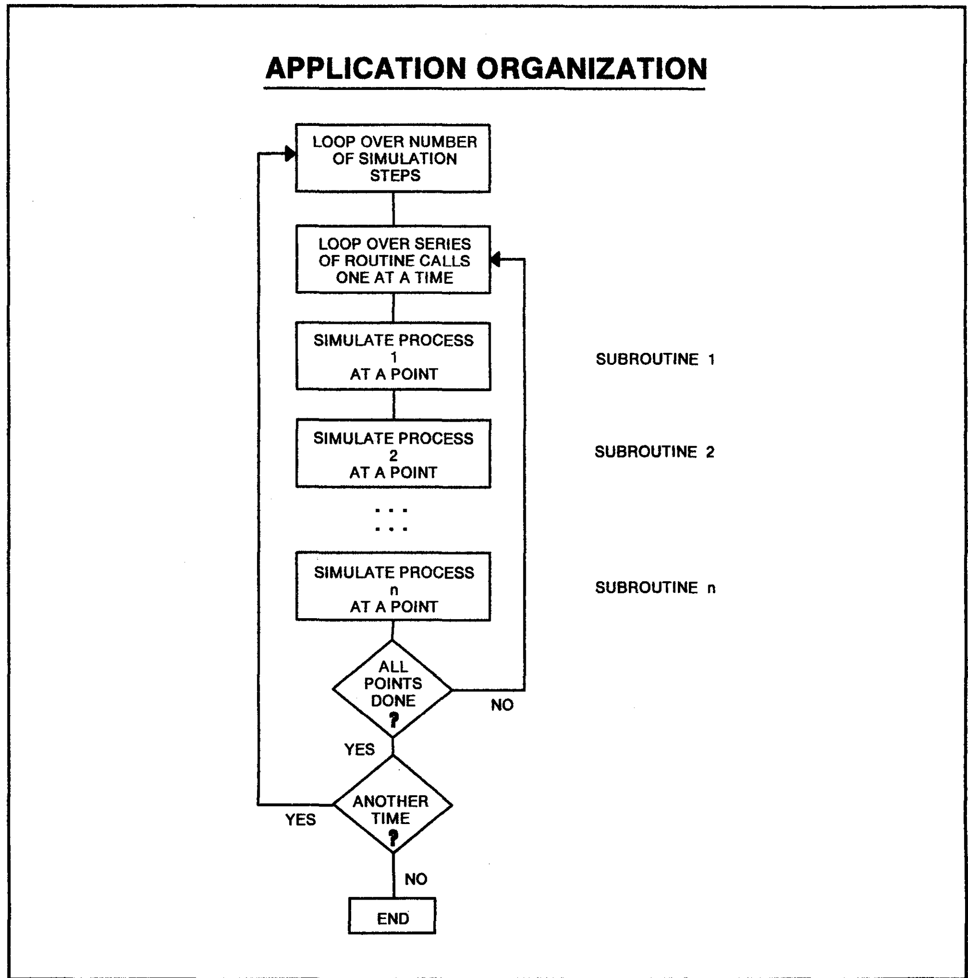



**Figure 5. Application Organization: Sequential**

However, by viewing this problem in a different way, we realize that a reorganization is possible: the same solution technique is applied to the data points, only in a different order. This means that the user may apply an algorithm in stages by applying the first part of an algorithm to all points to which it relates. The second part of the algorithm is applied to its set of points, and so on. This situation is illustrated in Figure 6 on page 14.

The data structure for this design might be much larger than for the original design, since many variables which were (undimensioned) scalars could now become arrays of points, or vectors. (The Dynamic COMMON feature of VS Fortran[5] permits full use of large virtual storage, such as the 2 gigabytes of virtual addressing available under MVS/XA, and makes it easy to manage this "scalar expansion".)

5 See the *VS Fortran Version 2 Programming Guide* (form number SC26-4118).

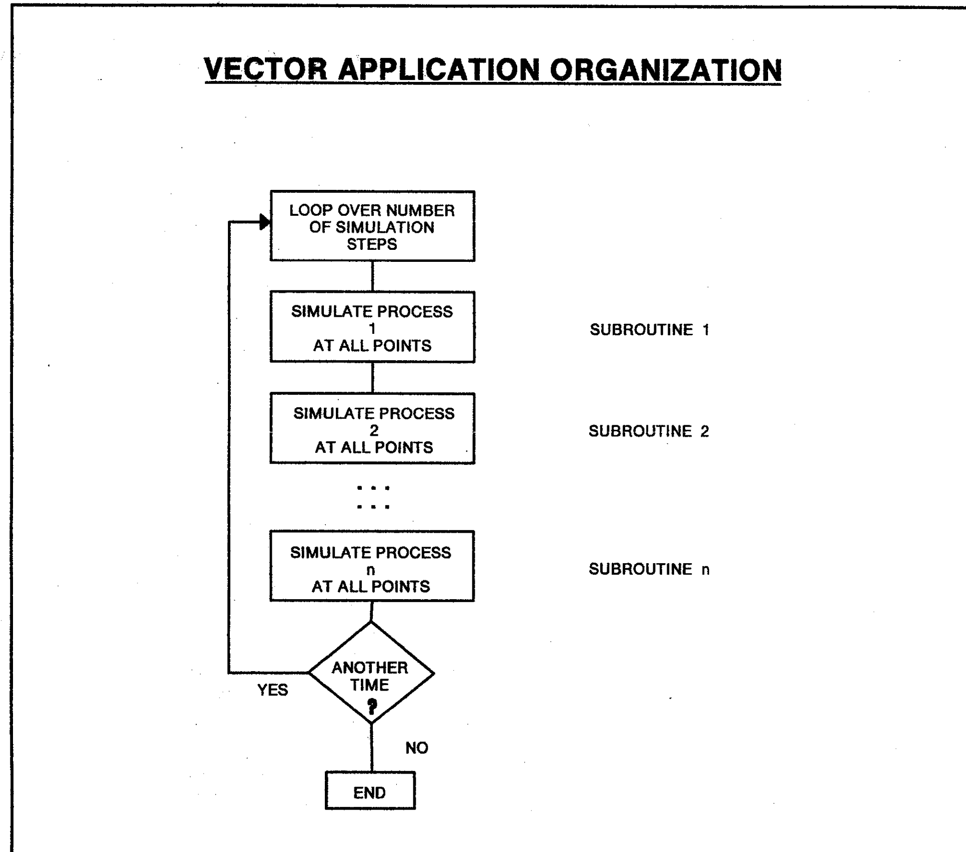



**Figure 6. Application Organization: Vector**

# 2.8 Measuring the Application

Adapting a program to enable vectorization is one form of program optimization. Like other program optimization activities, it is always useful to make measurements to determine where the program's CPU time is being spent. The distribution of CPU time can provide valuable information regarding how and where to focus one's effort in modifying the program.

One method of determining the CPU time characteristics of a Fortran application is through the use of an execution analyzer such as the VS Fortran Execution Analyzer[6], or any similar diagnostic tool, to detect "hot-spots" — segments of code where large fractions of the application's CPU time are spent. MVS provides STIMER/TTIMER macros, while VS Fortran Version 2 Interactive Debug[7] provides a TIME function. In addition, most large installations have some type of timer facility.

---

6 VS Fortran Execution Analyzer, Program Number 5798-DXJ.

7 See the *VS Fortran Version 2 Interactive Debug* Guide and Reference (Form Number SC26-4223).

Regardless of the method used to capture the CPU utilization, what is necessary is that the user understand where the time is being spent, not only among the routines, but within the most CPU intensive routines as well.

The interpretation of the distribution of CPU time may depend on the design of the application. The distribution of CPU time, along with an understanding of the logic on the program, can provide a valuable aid in the formulation of a migration strategy. The appearance of a "hot-spot" in the CPU time distribution may be interpreted differently if it occurs in a simulation organized in a "point-by-point" manner, than if it occurs in a code organized to handle many points per function.

These time-distribution observations should give the user direction as to where he should focus his effort. If the CPU time distribution is uneven, and is concentrated in one part of a program, the task may be as simple as analyzing one single loop; if it does not already vectorize, it may be either a replaceable function, or a modifiable or replaceable loop.

The user may thus have narrowed the scope of his efforts from many routines to one or a few loops, and presumably has reduced the amount of work to be done.

## 2.9 Vector Compilation

Assuming that the user has achieved an understanding of the organization and behavior of his application, the next thing to do is to submit the program to the VS Fortran Version 2 Vectorizing Compiler. If the resulting performance improvement is satisfactory, or if no further effort can be invested in program modification, the migration activity is complete.

In many situations, however, migrating an existing application program for vector execution may require more than simple vectorization (reliance on the vectorization capability of the compiler); it may also require more than just the analysis of inner DO loops. Rather, it may require an understanding of both the static structure and the dynamic behavior of the program.

The information gained through the characterization and measurement of the application now provides a basis for the interpretation of the results of vector compilation. That is, which DO loops are vectorized by the compiler? Are all of the important (CPU-intensive) loops vectorized? Do the loops which vectorize represent the real vector content of the application? Although the most CPU-intensive loops vectorize, they may not represent the greatest vector potential.

Since all vector operations do not perform at the same rate, the simple fact of vectorization may not be sufficient to achieve maximum performance improvement. Considerations such as vector length and density, stride and other addressing patterns will affect vector performance. Beginning with section 3, we will discuss some of the specific techniques which can be used to improve vectorization and vector execution performance. Before

introducing these techniques, the migration strategy should be completed by determining the scope of the migration, that is, the amount of modification and the level of effort required to accomplish that modification.

## 2.10 Scope of Application Modification

There are two basic types of program modification. In one case, the user alters the *form* of his program. The vector content may already be present, but may be expressed in a form that is awkward, overly complex, or ambiguous.

In the other case, the user alters the *method of solution* or the content of what the program is doing. This kind of modification may not always be desirable, but nevertheless, the user should be aware that sometimes it is necessary to consider modifying the solution technique in cases where the potential benefit may be sufficiently great.

With the accumulated information about the characteristics and organization of the application, and its CPU time behavior, the user is now in a position to make an important decision. Is simple vectorization enough? If the performance improvement resulting from application of the VS Fortran Version 2 Vectorizing Compiler is not maximizing the vector potential of the application then what migration steps should be taken?

## 2.11 Level of Effort

The process of defining a strategy for migration should not only identify the inhibitors to vectorization, and provide a plan for their removal, but must take into account the level of effort involved as well.

The curves in Figure 7 on page 17 are presented to illustrate the qualitative characteristics of the relationship between effort and benefit for the vector migration activity. The three "steps" in the left-hand curve indicate stages of incremental improvement in performance. The right hand curve illustrates level of effort. This is intended to convey the idea that each incremental level improvement in performance requires an increased level of effort. The performance gain eventually reaches a plateau, where all available vector content has been realized.

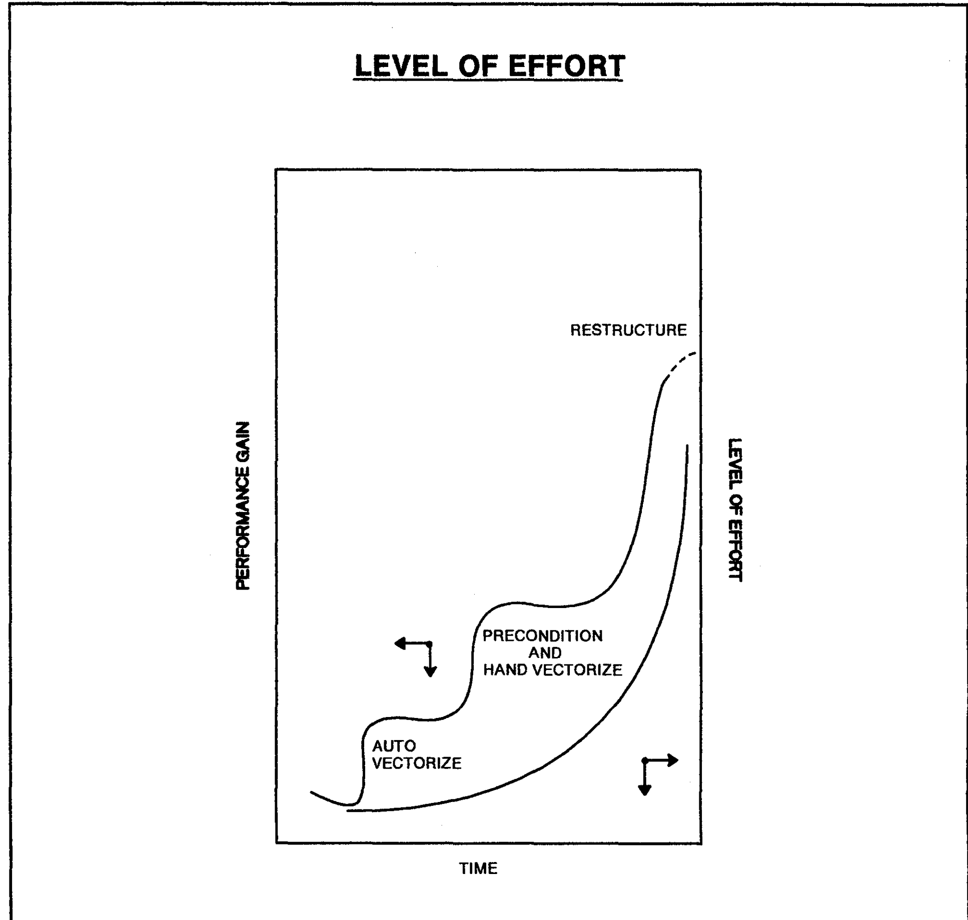



**Figure 7. Performance Gain Over Time, and Level of Effort Required**

# 3.0 Overview of Vectorization Concepts

The System/370 Model 3090 Vector Facility and the VS Fortran Version 2 Vectorizing Compiler introduce new techniques for engineering and scientific computation. We will review some of the relevant concepts here, before discussing program modifications that can help exploit the capabilities of the Vector Facility.

## 3.1 The Basic Unit of Vectorization: The DO Loop

The basic unit of vectorization is defined as the "DO" loop. Figure 8 illustrates a typical DO loop.

```
      DO 99 I = 1, N
        A(I) = A(I) + ...
   99 CONTINUE
```

**Figure 8. Basic Unit of Vectorization: the DO Loop**

There are other ways to code loops in Fortran, but only loops expressed as DO loops will be considered for vectorization.

## 3.2 The Basic Action of Vectorization: Loop Sectioning

The vector registers in the System/370 Model 3090 Vector Facility hold a predetermined number of data elements, which will only rarely be identical to the number of elements in a computer vector. Therefore, it is almost always necessary to split the computer vector into segments called *sections*. Each section may contain at most the number of elements a vector register can hold; this number is called the *section size*, and is denoted by "Z". Z is usually a power of 2, and is 128 for the System/370 Model 3090 Vector Facility.

Vectorizing a DO loop produces instructions that operate on *groups* of data elements.

```
        DO   10   J = 1, N
   10     A(J) = B(J)
```

**Figure 9. A Vectorizable Loop Before Sectioning**

Thus, the loop in Figure 9 is converted by the compiler into one loop (the original loop, now over "groups" of elements) which contains a second (conceptual) "loop over the elements in the group". This second (conceptual) loop represents the actions of the vector instructions themselves.

```
      DO   10   J = 1, N, Z     <-- Note Increment Z

        DO   xx   jv = J, J + MIN(N-J,Z-1), 1
   xx     A(jv) = B(jv)

   10   CONTINUE
```

**Figure 10. A Vectorizable Loop After Sectioning**

Note the following differences between the original loop in Figure 9 and the vectorized loop in Figure 10:

- The innermost loop ("DO xx") is executed in the vector hardware, in "groups" (sections) of "Z" elements at a time.
- The outer loop increment is "Z" instead of 1, so the vector instructions in the loop are executed approximately N/Z times, rather than the N times required for the equivalent scalar loop.
- The "remnant" left over (when N is not evenly divisible by "Z") is also executed in the vector hardware.

## 3.3 Loop Selection

Loop selection is a fundamental vectorization capability of the VS Fortran Version 2 Vectorizing Compiler. Unlike many other compilers, the VS Fortran Version 2 Vectorizing Compiler analyzes the innermost *eight* loops in a nest of DO loops, and selects the single loop whose vectorization will lead to the fastest execution of the entire nest. As with many other vectorization actions, there are ways to write the statements in the nest to increase the compiler's chances of exploiting possible vectorization opportunities.

```
      DO 15 I = 1, N
        DO 15 J = 1, M
          X(I,J) = AA(I,J) + BB(I,J)
15        Y(I,J) = AA(I,J) * BB(I,J)
```

**Figure 11. Loop Selection: Original Code**

Sometimes, based on the economic analysis the compiler performs to provide the most efficient loop execution, a loop structure like that shown in Figure 11 will be vectorized either by selecting the "J" (inner) loop for vectorization, or by selecting the "I" (outer) loop. That is, based on the available information regarding the loop limits, dimensions, stride, cost of instruction issue, and so forth, the compiler might vectorize the outer, "I" loop.

In addition to considering both DO loops as vectorization candidates, the compiler will also evaluate the possibility that the nest will execute fastest if *neither* loop is vectorized. It is quite possible that scalar execution might be faster than vector execution; if this is the case, the compiler will generate scalar code for the nest.

## 3.3.1 Vectorizing Outer DO Loops

In order to select the outer loop in Figure 11 for vectorization, the compiler must determine that the nest of loops would give the same result as if it were written as shown in Figure 12, with the two DO statements interchanged.

```
      DO 15 J = 1, M
        DO 15 I = 1, N
          X(I,J) = AA(I,J) + BB(I,J)
15        Y(I,J) = AA(I,J) * BB(I,J)
```

**Figure 12. Loop Selection: Equivalent Code**

The compiler determines that it is indeed safe to "interchange" the order of these two loops, since all computations within the loops are independent of the order in which they are computed.

Note carefully, however, that even though it is *safe* to "interchange" the order of the loops, they always remain in their *original* order! When the outer loop in Figure 11 is selected for vectorization, the compiler will generate instructions which section the loop on the leftmost ("I") subscript, as illustrated in Figure 13 on page 22:

```
      DO 15 I = 1, N, Z
        DO 15 J = 1, M
 ┌──────────────────────────────────────────────┐
          DO xx iv = I, I+MIN(N-Z,Z-1), 1
            X(iv,J) = AA(iv,J) + BB(iv,J)
 xx         Y(iv,J) = AA(iv,J) * BB(iv,J)
 └──────────────────────────────────────────────┘
 15   CONTINUE
```

**Figure 13. Loop Selection: Vectorized Code**

It can be seen in this example that the actual order of statement execution is not the same as for the "interchanged" loops in Figure 12 on page 21. Thus, it is useful to remember that "interchange" testing is just one stage in the compiler's assessment of the vectorization opportunities in a nest of DO loops.

Another example may help to clarify this process. When we speak of "vectorizing the outer loop", we actually mean that *all* eligible statements within that loop, including nested inner loops, will be vectorized on the index of that loop.

```
      DO 97 J = 1, 700
        H(J) = A(J) * B(J)
        DO 98 I = 1, 700
 98       C(J,I) = C(J,I) * (D(J,I) + H(J))
 97   CONTINUE
```

**Figure 14. Loop Selection: Vectorizing an Outer Loop**

In Figure 14, there are computational statements in both loops. When the outer loop with index J is vectorized, the statement in the inner loop is vectorized on J also. This result is indicated schematically in Figure 15, where the actions of the vector instructions are represented by the "loops" in the boxes.

```
      DO 97 J = 1, 700, Z
 ┌──────────────────────────────────────────────┐
        DO xx jv = J, J+MIN(700-Z,Z-1)
 xx       H(jv) = A(jv) * B(jv)
 └──────────────────────────────────────────────┘
        DO 98 I = 1, 700
 ┌──────────────────────────────────────────────┐
          DO ww jv = J, J+MIN(700-Z,Z-1)
 ww         C(jv,I) = C(jv,I) * (D(jv,I) + H(jv))
 └──────────────────────────────────────────────┘
 98     CONTINUE
 97   CONTINUE
```

**Figure 15. Schematic Form of Vectorized Outer Loop**

The processing in this nest first calculates a "J-section" of values of the array H, and then calculates 700 J-sections of the array C by stepping through all the values of the index I. Then, the next section (in J) of values of the array H is calculated, followed by the second set of 700 J-sections of the array C; and so on.

The important points to remember about vectorizing nests are:

- all statements inside the DO loop chosen for vectorization are considered for vector execution, not just those statements immediately nested in that DO loop;
- vectorizing an outer loop may actually cause more of the nest's computational work to be done in vector mode than if an inner loop were vectorized.

## 3.4 Data Independence

A key factor in enabling the vectorization of an application is the "data independence" of the vectors of data to be operated on by the Vector Facility's instructions. In general, data independence means that every operand in a vector is operated on independently of every other operand within that vector.

To illustrate a simple example of data independence, consider the DO loop in Figure 16:

```
      DO  99  J = 1, 20
 99     A(J) = A(J) + B(J)
```

**Figure 16. Simple Example of a Data Independent Loop**

The execution order of this loop is shown in Figure 17.

```
        A(1)  =  A(1)  +  B(1)
        A(2)  =  A(2)  +  B(2)
             - - -
        A(20) =  A(20) +  B(20)
```

**Figure 17. Execution Order of Data Independent Loop**

A value computed in each iteration of the loop in Figure 17 is not used in other iterations. The computation of the elements of A(I) in this loop are therefore independent of each other for all values of I. This independence of data values from one DO loop iteration to another is a key factor in allowing execution on the System/370 Model 3090 Vector Facility, and this DO loop can be vectorized.

By way of contrast, consider the DO loop in Figure 18 on page 24.

```
      DO   99   J = 1, 20
99      A(J+1) = A(J) + B(J)
```

**Figure 18. Simple Example of a Data Dependent Loop**

The execution order of this loop is shown in Figure 19.

```
        A(2)  =  A(1)  +  B(1)
        A(3)  =  A(2)  +  B(2)
            - - -
        A(21) =  A(20) +  B(20)
```

**Figure 19. Execution Order of Data Dependent Loop**

The value computed in all but the first iteration of the loop in Figure 19 is dependent on the value computed in the *previous* iteration. Therefore, this DO loop does not satisfy the requirement for data independence, and the loop cannot be vectorized.

In 3.6, "Indirect Addressing" on page 26, we will see another common programming practice that may not satisfy the data independence requirement.

Most of the localized program changes we will consider in this technical bulletin are intended to make any existing data independence as "visible" as possible to the VS Fortran Version 2 Vectorizing Compiler. At the same time, we may also need to be prepared to perform whatever other data or module reorganizations necessary to permit us to express this data independence clearly, or possibly even to eliminate certain dependences.

## 3.5 Recurrences

Recurrences occur so commonly in Fortran applications that it is worth examining their special properties with regard to vectorization. A recurrence carries a dependence between the elements of a vector (usually in the form of a linear relationship among the subscripts) which prevents its being used in a vector operation. The two loops in Figure 20 illustrate operations containing recurrences.

```
      DO 21 I = 2, 100
21      Q(I) = Q(I-1) + A(I)

      DO 22 I = 1, 99
22      Q(I+1) = Q(I) + A(I)
```

**Figure 20. Loops Demonstrating Recurrences**

In both cases, every computed element (except the first) of the array "Q" depends on the just-computed value of the preceding element; therefore the elements are not independent. This has the effect of inhibiting vectorization of the computations of the elements of "Q".

The compiler will not vectorize operations containing recurrences. As noted in the discussion of loop splitting in 5.9, "Loop Distribution" on page 55, the compiler will try to split a loop which contains recurrences in order to permit the vectorization of the other statements. In fact, even if the recurrence is implicit as the result of EQUIVALENCE statements, as seen in Figure 21, the compiler will assume that a dependence exists, and will not vectorize the statement.

```
      EQUIVALENCE (R(1), Q(1))
      - - - -
      DO 23 I = 1, N
   23   R(I+1) = Q(I) + A(I)
```

**Figure 21. Loop Demonstrating a Implicit Recurrence**

In this example, "R" and "Q" refer to the same storage. If "R" is replaced by "Q" within the loop, we find the same recurrence relationship as given by the preceding example in Figure 20 on page 24.

The subscript relationships illustrated in Figure 20 on page 24 need not always imply a recurrence. In Figure 22, the DO loop increment has been changed from 1 to 2, and the recurrence vanishes!

```
      DO 21 I = 2, 100, 2
   21   Q(I) = Q(I-1) + A(I)
```

**Figure 22. A Loop With No Recurrence**

This DO loop may now be vectorized, because all values computed in the loop are independent of one another.

The dependence of the computed value of one element of a vector on others does not necessarily mean the computation cannot be vectorized. Suppose the loop in Figure 18 on page 24 had been written in the slightly modified form shown in Figure 23.

```
      DO 99 I = 1, 20
   99   A(J) = A(J+1) + B(J)
```

**Figure 23. Loop With No Recurrence**

The execution order of this loop is shown in Figure 24 on page 26.

```
A(1)  =  A(2)  +  B(1)
A(2)  =  A(3)  +  B(2)
 - - -
A(20) =  A(21) +  B(20)
```

**Figure 24. Execution Order of Loop with No Recurrence**

The value computed in each iteration of the loop is *independent* of all values computed in *previous* iterations of the loop. Therefore, this loop can be vectorized.

However, if the DO loop index runs from 20 to 1 in steps of -1, a recurrence does exist, and the loop cannot be vectorized. This is illustrated in Figure 25.

```
      DO 99 I = 20, 1, -1
99      A(J) = A(J+1) + B(J)
```

**Figure 25. Similar Loop, Now Containing a Recurrence**

Even though this DO loop contains the same statement as in Figure 23 on page 25, the change in the direction of loop traversal causes a recurrence.

## 3.6 Indirect Addressing

Indirect addressing is concerned with addressing an array by using subscripts which are themselves subscripted. Thus, it involves a separate array of subscript values in addition to the array whose elements are directly involved in the operation(s) to be performed. This is illustrated in Figure 26.

```
      DO 26 J = 1, N
26      A(INDX(J)) = 0.0
```

**Figure 26. Example of Indirect Addressing**

This array of subscript values need not be in any specific order, and may participate in vector operations. As long as the array which is indirectly addressed does not appear on both sides of the equal sign in a Fortran statement, the statement may be eligible for vectorization. This is equivalent to saying that a statement which uses an indirectly addressed variable may be vectorized if there are either only loads of the variable, or only stores of the variable, but not if there are both.

The examples in Figure 27 on page 27 both illustrate this requirement.

```
      DO 27 J = 1, N
 27     R(J) = A(INDX(J)) + C(J) * .5

      DO 28 J = 1, N
 28     A(INDX(J)) = R(J) + C(J) * .5
```

**Figure 27. Vectorizable Indirect-Addressing Loops**

Each of the DO loops is vectorizable, since it involves only fetches (loads) or only stores of the indirectly addressed array "A". Note that the values of the elements of the INDX array may be arbitrary, so long as they do not violate the declared dimension bounds of the array A.

The last example, in Figure 28, shows a loop which will not be vectorized, since the compiler has no way to determine whether any value in the list vector "INDX" is repeated.

```
      DO 29 I = 1, N
 29     A(INDX(I)) = A(INDX(I)) + B(I)
```

**Figure 28. Non-Vectorizable Indirect-Addressing Loop**

If two elements of the INDX array have the same value, say INDX(i1) and INDX(i2), then A(INDX(i2)) would not be independent of A(INDX(i1)), since they are the same element of "A". Thus, the requirement of data independence is not satisfied, and this loop cannot be vectorized. Since the array of subscript values may be dynamically generated, uniqueness of values in general cannot be guaranteed.

## 3.7 The Stride of a Vector

The value of localized memory references was mentioned earlier, in 2.2.2, "Virtual Storage Compatibility" on page 5. The concept of "stride" is helpful in understanding how data can be organized and referenced in order to improve localization.

As defined on Appendix A, "Glossary of Terms and Concepts" on page 77, the stride of a vector is the addressing increment between successive elements divided by the element length. In Fortran terms, "stride" has a much simpler characterization.

```
      DIMENSION A(1000), B(1000)
      - - - -
      DO 44 J = 1, 1000
        A(J) = A(J) + B(J)
   44 CONTINUE
```

**Figure 29. A DO Loop With Stride-1 Memory References**

In Figure 29 on page 27, the elements of the arrays “A” and “B” are being referenced in order of *adjacent* elements; thus, their addressing increment is the same as the length of each element. We call this a “stride-1” reference pattern, or sometimes simply “stride-1” for short.

In Figure 30, there are two loops referring to the array “C”.

```
*    STRIDE 1                          *    STRIDE 50
      DIMENSION C(50,300)                    DIMENSION C(50,300)
      - - - -                                - - - -
      DO 45 J = 1, 50                        DO 46 K = 1, 300
        C(J,2) = 0.0                           C(17,K) = 0.0
   44 CONTINUE                            46 CONTINUE
```

**Figure 30. DO Loops With Stride 1 and Stride 50**

The loop on the left is varying the leftmost subscript of the array C, and because Fortran stores arrays in “column-major” order in which the leftmost subscript varies most rapidly, the memory references will be “stride-1”. However, the loop on the right varies the rightmost subscript of the array “C”; thus the memory references will be to elements of “C” at a stride of 50.

The stride of memory references is only one of the factors controlling the compiler's choice of a loop to vectorize. For example, in Figure 31, the compiler might choose to vectorize either the inner or the outer loop, depending on the array sizes.

```
      REAL A(20,20),B(20,20)               REAL A(80,80),B(80,80)
      - - - -                              - - - -
      DO 1 K = 1, 20                       DO 1 K = 1, 80
        DO 1 J = 1, 20                       DO 1 J = 1, 80
          A(J,K)=B(J,K)*A(J,K)                 A(J,K)=B(J,K)*A(J,K)
    1 CONTINUE                           1 CONTINUE
```

**Figure 31. Identical DO Loops With Different Strides and Counts**

The nest of DO loops on the left might be vectorized on the outer loop, because the overhead of sectioning the inner loop 20 times can be avoided, and the stride of 20 is reasonably small. The nest on the right might be vectorized on the inner loop to minimize stride costs, but at the expense of having to do the vector loop initiation 80 more times.

Experience has shown that when all other factors can be kept unchanged, the best vector performance is usually obtained for small strides. Many of the examples that follow will illustrate techniques for reducing the stride of array references in loops suitable for vectorization.

# 3.8 Sources of Numerically Different Results

When a program is being converted from scalar to vectorized form, it is usual practice to compare the numerical results from the two versions. Almost all the vector instructions generated by the VS Fortran Version 2 Vectorizing Compiler produce results that are identical to the results produced by the equivalent scalar instructions.

However, there are two different situations where the results produced from vectorized programs may be different from the results produced when those programs are executed in scalar mode. These situations involve

- reduction operations, and
- intrinsic function references.

We will discuss each of these in turn, and explain how to prevent vectorizations which could lead to the resulting numerical differences, by using the NOREDUCTION and NOINTRINSIC sub-options of the VECTOR compiler option. The user is cautioned to consider these differences during the numerical validation of vectorized code.

Furthermore, the difference in interpretation of the DO statement between the Fortran-66 and Fortran-77 standards is another possible source of numerically different results.

## 3.8.1 Vectorization of Reduction Operations

Reduction operations involve accumulating the sum of the elements of a vector into a scalar. Special hardware instructions can be generated by the VS Fortran Version 2 Vectorizing Compiler to perform these operations. Two typical reductions are summing the elements of a vector, and calculating the inner ("dot") product of two vectors.

The sum of the elements of a vector is illustrated in Figure 32. The VS Fortran Version 2 Vectorizing Compiler will recognize and automatically vectorize such a sum reduction operation.

```
      S = 0.0
      DO 99 I = 1, N
99      S = S + A(I)
```

**Figure 32. Summing Elements of a Vector**

Similarly, the "dot product" or "inner product", illustrated in Figure 33 on page 30 is another example of code for which the vector result may not be "bit-by-bit" identical to the scalar computation due to the use of the vector accumulate instructions.

```
      D = 0.0
      DO 99 I = 1, N
99      D = D + A(I) * B(I)
```

**Figure 33. Dot Product of Vectors A and B**

The VS Fortran Version 2 Vectorizing Compiler will automatically recognize and vectorize such loops.

These examples illustrate the few instances where the user does not achieve "bit for bit" numerical equivalence of results between the scalar and vector implementations of the same operation. This is because the vector summation is performed using vector "accumulate" instructions which do not necessarily perform the addition of elements in the same order as the original scalar code.

The accumulate operation is accomplished by forming *partial sums* (for the System/370 Model 3090 Vector Facility, the number of partial sums is 4), and then by summing the partial sums.

For example, the elements A(1), A(2), . . . A(N) in Figure 32 on page 29 would be added first as four partial sums, as shown in Figure 34.

```
Partial sum 1 = A(1) + A(5) + A(9)  + ...
Partial sum 2 = A(2) + A(6) + A(10) + ...
Partial sum 3 = A(3) + A(7) + A(11) + ...
Partial sum 4 = A(4) + A(8) + A(12) + ...
```

**Figure 34. Partial Summation in Reduction Operations**

Finally, the four partial sums are added in sequence to form the final result. Because the order of summation is different from a normal DO loop's scalar summation, cancellation and truncation of intermediate results may occur in different places. As a result, some numerical differences may occur when reduction operations are vectorized.

Another possible source of result differences is that sums of short precision (REAL*4) operands are accumulated in long precision (REAL*8), thus reducing truncation errors.

If some step in the migration process requires that reduction operations give the same results as scalar code, the user may specify the NOREDUCTION sub-option (abbreviated NORED) of the VS Fortran Version 2 Vectorizing Compiler's VECTOR option. This will turn off the compiler's attempted recognition and vectorization of reduction operations.

### 3.8.2 Vectorization of Library Intrinsic Functions

Numerical differences may also be encountered when comparing the results obtained using the VS Fortran Version 2 intrinsic function library with results using previous Fortran libraries (G, G1, H, H-Extended, and VS Fortran Version 1). This is due to the major improvements in precision introduced in the VS Fortran Version 2 Library. Many of the algorithms used in the VS Fortran Version 2 Library have been upgraded to provide greatly increased accuracy over the entire numerical range of the various library functions. In addition, the results from the vector and scalar versions of the VS Fortran Version 2 Library routines provide bit-identical results for all arguments.

To help with verifying results during application migration, the NOINTRINSIC sub-option (abbreviated NOINT) of the VECTOR option tells the compiler that the user wishes the compiler *not* to invoke the vector versions of the Fortran intrinsic functions. Since the compiler normally selects the vector versions of the Fortran intrinsic functions automatically during the vectorization process, the use of this sub-option will force the compiler to use the scalar versions for all intrinsics.

Having specified the NOINTRINSIC sub-option at compile time, the user may provide a link-time library containing the *old* library routines. Thus, the numerical results from the intrinsic functions will be identical to the "old" values, while the rest of the program may be vectorized. Once vectorization results are satisfactory, the INTRINSIC sub-option (which is the default) is specified, and the new library routines will be used automatically.

The user may now validate the vectorized code assured of bit-for-bit equivalence between the scalar and vector versions of the intrinsics when using VS Fortran Version 2. Numerical differences in the results can then be attributed to the increased accuracy of the new routines (assuming that possible differences due to reduction operations have already been taken into account.)

### 3.8.3 Fortran-66 and Fortran-77 Execution of DO Loops

One other factor to consider in preparing programs for vectorization is the different treatment of DO loops between the Fortran-66 and Fortran-77 standards. In Fortran-66, DO loops are always traversed at least once. Fortran-77 requires that the "trip count" of the loop be checked before beginning the execution of the loop, and if the count is not greater than zero, the loop must not be executed.

```
      DO 99 I = M, N
        A(I) = A(I) + ...
   99 CONTINUE
```

**Figure 35. A DO Loop With Unknown Control Parameters**

If the loop limit N is smaller than the initial value M in the DO loop illustrated in Figure 35, then the loop will not be executed at all when the compiler parameter LANGLVL(77) is specified. If an application program depends on at least one execution of the DO loop, different numerical results might result.

Even though the VS Fortran Version 2 Vectorizing Compiler will vectorize programs written at both the Fortran-66 and Fortran-77 language levels, the user is cautioned to check for the possibility that his program depends on one of the two standard interpretations of the DO statement.

# 4.0 Local Vectorization Considerations

Vectorization is centered on DO loops, and the statements they contain. In this section we will examine some of the factors that determine whether a given DO loop should or should not be vectorized.

## 4.1 Not All DO Loops Are Appropriate for Vectorization

```
      DO 99 I = 1, N
        J = 2 * MOD(I,M) / K
        A(J) = A(J) + ...
   99 CONTINUE
```

**Figure 36. A DO Loop Inappropriate for Vectorization**

However, not all "DO" loops are appropriate for vectorization. Recall that vectorization requires data independence among the elements of a vector.

The computation of the subscript "J" in Figure 36, on which A depends, does not necessarily have a unique set of values. Some elements of A could depend on each other as a result. Both the MOD function and the division of an integer by an integer provide a set of values, some of which may be repeated.

In this example, that means that for several values of "I", there might be identical values of "J". Thus an element of A may be recomputed several times, which means that this loop cannot be expressed as a vector operation. Although it appears to be a simple loop, it is not vectorizable because the computations of the elements of "A" are not independent of each other.

## 4.2 Not All DO Loops Are Well Suited for Vectorization

```
      COMMON /COM/ INC, ...
      - - - -
      DO 99 I = 1, N, M
        A(I+INC) = A(I) + ...
   99 CONTINUE
```

**Figure 37. A DO Loop Not Well Suited for Vectorization**

Some DO loops may not be suitable for vectorization. The DO loop in Figure 37 ranges from 1 to N, with an increment of M. The computation of one index for A has an increment INC whose value (for purposes of this example) we assume is not known within the scope of this routine. By itself, this would not automatically prevent vectorization.

In this case, however, there is no way to guarantee that the value of this increment does not cause an overlap of subscript values with the index of the "DO" loop. This means that values of A may not be independent of each other under all conditions. It may be that they actually *are* independent, but there is no way for the compiler to examine them and determine that this loop may be safely vectorized.

## 4.3 Not All Loops Are DO Loops

Another FORTRAN loop which will not vectorize is the "IF-GOTO" loop, as shown in Figure 38.

```
      I = 0
   11 I = I + 1
        - - - -
        A(I) = ...
      IF (I .LT. N) GO TO 11
```

**Figure 38. A Hand-Coded Loop**

The "IF-GOTO" loop of course provides much the same function as a DO loop in scalar mode. The compiler, however, does not recognize an "IF-GOTO" loop as a "DO" loop, and will not vectorize it.

## 4.4 Some Loops Should Be Written as DO Loops

In Fortran-66, a DO loop index was expected to increase. In cases where it was necessary for the loop index to decrease, either an auxiliary subscript variable had to be created (see 5.4.2, "Computed Auxiliary Subscripting Variables" on page 48 for further discussion), or a hand-written "IF-GOTO" loop was written, as shown in Figure 39 on page 35.

```
      I = N
   11 CONTINUE
        - - - -
        A(I) = ...
        I = I - 1
        IF (I .GT. 0) GO TO 11
```

**Figure 39. A Hand-Coded Backward Loop**

The Fortran-77 standard permits a negative increment for DO loops, so this "IF-GOTO" loop can now be written in the simpler form shown in Figure 40.

```
      DO 11 I = N, 1, -1
        - - - -
        A(I) = ...
   11 CONTINUE
```

**Figure 40. A Standard Backward DO Loop**

The VS Fortran Version 2 Vectorizing Compiler can recognize this loop as a "DO" loop, and can now consider it for vectorization.

# 4.5 Some DO Loops Iterate Too Few Times

Although the "DO" loop in Figure 41 is properly posed and unambiguous, the VS Fortran Version 2 Vectorizing Compiler will determine that this loop is executed too few times to benefit from vector execution.

```
      DO 99 I = 1, 2
        A(I) = A(I) + ...
   99 CONTINUE
```

**Figure 41. A DO Loop With Small Iteration Count**

The compiler takes account of the fact that all vector instructions have some amount of overhead (CPU time) associated with the initiation and termination of the instruction. As a result, depending on which of the vector instructions is being used, there is a minimum vector length (or loop iteration count) below which it is computationally more efficient to perform the loop's operations in scalar mode.

On the average, this number is approximately 12. A loop with a length of less than 12 should probably not be vectorized. (In fact, for the example in Figure 41, a loop with a length of 2 should probably have been written explicitly, rather than as a "DO" loop, just as a matter of good programming practice.)

## 4.6 Some Loop Iteration Counts Are Unknown

If the value of the loop iteration count "N" is determined by some other computation or is an unknown parameter of the problem, as illustrated in Figure 42, then there is no *a priori* way for the compiler to determine the benefits of vectorization.

```
      DO 99 I = 1, N
        A(I) = A(I) + ...
 99   CONTINUE
```

**Figure 42. A DO Loop With Unknown Iteration Count**

Assuming that no vectorization inhibitors are present, and in the absence of other information, the compiler will estimate a "reasonable" iteration count, and such a loop would probably be vectorized. This will not automatically lead to improved performance, however; if the actual value of N is small, scalar execution could be more efficient.

## 4.7 Some DO Loops Cannot Be Vectorized

The Fortran-77 standard permits the index and control parameters of a DO loop to have integer or real values. While almost all programs use integer variables and values for the control parameters, a program might use real values, as illustrated in Figure 43.

```
      REAL X
      - - - -
      DO 29 X = 0.46, 8.95, 0.01
 29     SUM = SUM + EXP(-B * ATAN(X))
```

**Figure 43. DO Loop With REAL Index**

Such a loop will not be vectorized by the VS Fortran Version 2 Vectorizing Compiler. The equivalent form shown in Figure 44 on page 37 will be vectorized.

```
      REAL X
      - - - -
      DO 29 J = 46, 895, 1
        X = FLOAT(J) / 100.0
 29     SUM = SUM + EXP(-B * ATAN(X))
```

**Figure 44. DO Loop With Index Converted to INTEGER**

## 4.8 Summary

Now that the background for the vector migration process has been established, we proceed to discuss some of the details of local vectorization. We will discover, however, that no amount of local modification for vector migration will provide significant improvement if the basic computation ordering, module structure, or data structure is inconsistent with the vectorization process. Some of these more global considerations are presented in 6.0, "Global Migration Considerations" on page 71.

The next section, 5.0, "Local Vectorization Techniques" on page 39, discusses some of the methods which apply to local modifications for vector migration.

# 5.0 Local Vectorization Techniques

The following section discusses the vectorization of loops and loop structures and provides numerous examples. These examples are not intended to state rules or a list of "do's and don'ts". Instead, the examples attempt to identify items that prevent or inhibit vectorization, and techniques that help to enhance vectorization. As long as the user keeps data independence in mind, he is free to write code in the way he is accustomed to. No particular style or particular technique is emphasized.

In many applications, the quickest and easiest path to improved performance on the System/370 Model 3090 Vector Facility is to replace a Fortran-written algorithm with calls to one or more of the routines in the Engineering and Scientific Subroutine Library. These routines have been highly optimized to exploit the characteristics of the Vector Facility, and they can provide substantial performance gains in applications that use them heavily.

However, not all applications spend almost all of their time executing just a few algorithms. In these cases, whether a user develops a new application or migrates an old one, style still counts. Style encompasses both the manner in which the algorithms are written or expressed and the way in which the overall program and data are organized.

Besides the Engineering and Scientific Subroutine Library, the VS Fortran Version 2 Vectorizing Compiler is the major tool available to achieve improved vector performance. This compiler has additions to the features of VS Fortran Version 1, and a new option which controls the vectorization of Fortran source code. Although the compiler uses state-of-the-art techniques for vectorization, certain programming styles and practices in current use may result in the vector content of the code being obscured or hidden from the compiler. Also, as discussed earlier, the program's organization may have spread the vector content across many subroutines, where the compiler (which can process only one subroutine at a time) cannot "see" it.

Therefore, the user sometimes has to intervene. When intervention is required, we have observed that a number of types of helpful modifications occur again and again. The following sections discuss some of the more common practices which have been successfully used to improve vectorization.

The examples which follow have been taken out of context. They are loops whose structures have been simplified to illustrate a point. The examples used are skeletons, and are intended to provide clues to improving the vectorization process, even when embedded in complicated code structures.

These examples are not intended as a list of right vs. wrong programming techniques. Rather, they are intended to suggest potential improvements to vector programming as more general practices which can be applied to realistic and probably more complex application programs. Some of the examples, in fact, use simple loops which are vectorizable in their current form. The discussion and evaluation of these examples is worthwhile, however, since it serves to indicate some of the limits on the vectorization process. In these examples, the ellipses (written as " . . . " or as "- - - -") are used to indicate other work to be performed within the loop(s) to remind us that we are only considering one type of Fortran construct at a time, in what is potentially a very complex structure otherwise.

Some of these expressions may prevent the code from vectorizing if they appear in a larger loop. The degree of vectorization may vary, depending on the user's choice of the "VECTOR" compiler option, LEVEL(1) or LEVEL(2). At LEVEL(1), all statements within a loop must be vectorizable in order for the loop as a whole to vectorize. The LEVEL(2) option will cause the compiler to attempt to vectorize a loop, even if all statements within the loop cannot be vectorized. This may be accomplished through a combination of loop splitting, scalar expansion, IF conversion, etc.

## 5.1 Stride

On the System/370 Model 3090 Vector Facility, the performance of any single instruction is generally optimal when its data is referenced with stride-1 addressing. If a DO loop can refer to vectors with stride 1, performance will generally be much better than if a longer stride is used. However, many DO loops, and many nests of DO loops, must refer to vectors having a variety of strides. Many of the following examples will explore techniques for "improving" the stride of vector accesses.

One of the first considerations is for the user to determine where arrays are referenced at strides other than 1, and whether those references have to remain that way. The two examples in Figure 45 on page 41 will yield the same result; but in scalar mode, the innermost loop in the first example will reference arrays with a stride of N*M, while the innermost loop in the second example will reference the arrays in storage order (stride 1). However, the *outermost* loop in the first example will reference the arrays with a stride of 1, while the outermost loop in the second example will reference the arrays with a stride of N*M.

```
      DO 99 I = 1, N
        DO 99 J = 1, M
          DO 99 K = 1, L
            A(I,J,K) = B(I,J,K) + C(I,J,K)
               ...   =   ... + A(I,J,K) + ...
               ...   =   ... * A(I,J,K) * ...
   99 CONTINUE

      DO 99 K = 1, L
        DO 99 J = 1, M
          DO 99 I = 1, N
            A(I,J,K) = B(I,J,K) + C(I,J,K)
               ...   =   ... + A(I,J,K) + ...
               ...   =   ... * A(I,J,K) * ...
   99 CONTINUE
```

**Figure 45. Loops With Different Strides**

Depending on the dimensions of the arrays and the number of iterations of each loop, any of the three DO loops could be selected for vectorization. In general, however, it is best to avoid making the loop with the largest iteration count control the vectors with the longest stride, and to enhance the use of large iteration counts to control vectors with short stride.

Thus, the first example in Figure 45 is best if N is large compared to M and L, and the first dimension of each array is large. Conversely, if the second and third dimensions of the arrays are also large, the second example may be best. Which form will lead to the best performance in any given application must be determined by CPU timings.

### 5.1.1 Stride and Recurrences

Suppose there is a recurrence in the subscript controlled by the innermost DO loop, as shown in Figure 46.

```
      DO 99 J = 1, M
        DO 99 I = 2, N
          A(I,J) = A(I-1,J) + B(I,J)
   99 CONTINUE
```

**Figure 46. Loop With Dependence in One Dimension**

The compiler cannot vectorize the inner "I" loop due to the recurrence, but the outer "J" loop is eligible, and may be selected for vectorization depending on the economic analysis. Unless the actual value of "N" is known, we do not know if the resulting performance will be acceptable. If "N" and the first dimension of A and B have a value of the order of 1000, performance may well be disappointing, while if "N" is of order 10 the performance may be quite acceptable.

### 5.1.2 Stride Minimization

Next, we consider an example of what can be done to remove some of the references at strides other than 1, without harming the computation. In Figure 47 a type of reduction operation in "I" and "J" is found.

```
       DO 99 I = 2, N
         DO 99 J = 2, M
           DO 99 K = 1, L
             A(K) = 1.0 / X(I,J,K)
             B(K) = X(I-1,J,K) * A(K) + X(I,J-1,K)
             C(K) = X(I,J+1,K) - B(K) * X(I-1,J,K)
    99 CONTINUE
```

**Figure 47. Loop With Many Long-Stride References**

This means that the order of the loops may not be changed. The compiler may determine that the operation will be most efficiently performed in scalar mode.

An interesting aspect of this procedure is that each of the operations controlled by the innermost DO processes almost the entire range of of the array X. Almost all of the values will be used, even though the stride may be large. Since most of the values will be involved and they will be utilized many times in a number of statements, it may be profitable to take the "penalty" of referencing the array "X" with large stride once, and create a temporary copy ("XX" in Figure 48) in which the data is reordered so that the several other operations address the data in storage order, with stride 1.

In order to achieve maximum re-use, all references to the variable "X" in the main nest of DO loops are replaced with references to the copy, "XX", as shown in Figure 48.

```
       DO 97 I = 2, N
         DO 97 J = 2, M
           DO 97 K = 1, L
    97       XX(K,J,I) = X(I,J,K)
       DO 99 I = 2, N
         DO 99 J = 2, M
           DO 99 K = 1, L
             A(K) = 1.0 / XX(K,J,I)
             B(K) = XX(K,J,I-1) * A(K) + XX(K,J-1,I)
             C(K) = XX(K,J+1,I) - B(K) * XX(K,J,I-1)
    99 CONTINUE
       DO 98 I = 2, N
         DO 98 J = 2, M
           DO 98 K = 1, L
    98       X(K,J,I) = XX(I,J,K)
```

**Figure 48. Copying Data to Minimize Long-Stride References**

The associated cost of introducing the extra copy operations may be more than compensated for by the speed improvement of the vector execution of this loop, which has now been made possible.

As a general rule, when the major operations in a DO loop must be performed with some stride other than 1, the code should be analyzed to determine if there are a sufficient number of references to the affected variable to warrant the extra cost of the copy operation.

## 5.2 Data Organization

Even if the DO loops are well organized logically and syntactically, and are obviously vectorizable, the *data organization* may not be appropriate to optimize the vector performance of the loop structure. If this is the case, the user should plan to organize the data so that its vectors have the longest possible length and the shortest possible stride. As with any vector machine, some overhead is required to initiate vector operations. The longer the vector, the more closely the performance approaches the maximum.

### 5.2.1 Reorganizing Data to Improve Stride

Suppose we have an application in which a basic "data element" is a 5x5 matrix, and we need to handle 10,000 of these matrices. This is illustrated in Figure 49.

```
      DIMENSION  A(5,5,10000)
      - - - -
      DO 99 K = 1, 10000
        DO 99 ICOL = 1, 5
          DO 99 IROW = 1, 5
            A(IROW, ICOL, K) = A(IROW, ICOL, K) + ...
   99 CONTINUE
```

**Figure 49. Inappropriate Data Organization for Vectorization**

The analysis of this situation proceeds as follows. Should the loops in Figure 49 be thought of as operating, one matrix at a time, on each of 10,000 matrices each of which is a (5x5) square array, or can the requisite operation be performed on one of the array elements at a time for each of 10,000 5x5 arrays?

The answer to this question will determine whether the data dimensioning is specified as (5,5,10000) or as (10000,5,5). When the compiler vectorizes the "K" loop (the others are too short), the vectors will be computed with a stride of 25 with the first choice, or a stride of 1 with the second choice.

Since maximum vector performance is achieved by addressing long vectors at a stride of 1, and since for this example, there is no difficulty in re-ordering the data, the order of operations may be arranged so that all

10,000 arrays have the same element computed as a single vector. This situation is shown in Figure 50 on page 44.

```
      DIMENSION  A(10000, 5, 5)
      - - - -
      DO 99 ICOL = 1, 5
        DO 99 IROW = 1, 5
          DO 99 K = 1, 10000
            A(K, IROW, ICOL) = A(K, IROW, ICOL) + ...
   99 CONTINUE
```

**Figure 50. Data Organization More Appropriate for Vectorization**

## 5.2.2 Using EQUIVALENCE to Improve Vector Length

One of the factors which affects vector performance improvement is vector length. As discussed earlier, increasing the length of the vectors generally improves performance. It is appropriate to consider at least one technique by which vector length may be improved without resorting to extreme measures.

The example shown in Figure 51 shows a 3-dimensional nested loop.

```
      DIMENSION A(N,M,L), B(N,M,L)
      - - - -
      DO 99 K = 1, L
        DO 99 J = 1, M
          DO 99 I = 1, N
 99         A(I,J,K) = A(I,J,K) + B(I,J,K)
```

**Figure 51. Loops With 3-Dimension Arrays**

Let us assume that the array dimensions are larger than the minimum required for vector execution, but not particularly large. The nest of loops in this example contains no dependences in any of the three DO indexes, and all three are eligible for vectorization. The compiler will only select one loop for vectorization, and by our assumptions, the vector length may not be long enough to achieve the full potential performance improvement.

We observe that the operations are all independent and vectorizable in the first two dimensions. We can then take advantage of Fortran's linear mapping of multiply-dimensioned variables to "collapse" the leading two dimensions into a single linear dimension whose length is equal to the product of the original two. The coding technique employed uses the EQUIVALENCE statement. As shown in Figure 52 on page 45, new variables are defined and individually EQUIVALENCEd to the original variables so that they share the same virtual storage.

```
      DIMENSION A(N,M,L), B(N,M,L), AA(NM,L), BB(NM,L)
      EQUIVALENCE (A(1,1,1),AA(1,1)), (B(1,1,1),BB(1,1))
      - - - -
      DO 99 K = 1, L
        DO 99 IJ = 1, NM
   99     AA(IJ,K) = AA(IJ,K) + BB(IJ,K)
```

**Figure 52. Loops With 2-Dimension EQUIVALENCEd Arrays**

The original two dimensions may now be indexed with a single linear index whose length (by our assumption) is significantly greater than either of the original two. The resulting vector operations now are more likely to approach optimal vector length performance. Although this example only shows the combination of two of the three dimensions, all three dimensions in this example might have been combined, with a resulting vector length of N*M*L.

While this technique may appear somewhat artificial, it is similar to the "effective" equivalence which occurs when multiply dimensioned arrays are passed as arguments to a subroutine which uses a single linear index to address the entire array in storage order. This is typical of the practice of using subroutines to provide a computational result based solely on the starting address and length for all input and output arrays. The use of EQUIVALENCE provides the same function, in this case, without the overhead of subroutine CALLs.

## 5.2.3 Data Restructuring

We next present an example of restructuring the data to promote vectorization. Suppose that the application program is designed to solve, handle or otherwise manipulate a small number of simultaneous equations independently at many points on a grid. Let us represent this process by a matrix transpose on a small (5x5) matrix, at each point of the grid (I,J,K) as seen in Figure 53.

```
      DIMENSION A(5,5,N,M,L), B(5,5,N,M,L)
      . . .
      DO 99 K = 1, L
       DO 99 J = 1, M
        DO 99 I = 1, N
         DO 99 ICOL = 1, 5
          DO 99 IROW = 1, 5
   99      B(ICOL,IROW,I,J,K) = A(IROW,ICOL,I,J,K) + ...
```

**Figure 53. Poor Loop and Data Structure**

Any vector operation over the matrix dimension using either of the two innermost loops is restricted to a length of 5. Operating on the many matrices by forming one of the matrix elements for all points in one of the grid dimensions would be performed at some stride other than 1 (25, at a minimum). By restructuring the data array and using this latter evaluation

ordering, we may process one matrix element (IROW,ICOL) for all such matrices in one grid dimension.

```
      DIMENSION A(N,M,L,5,5), B(N,M,L,5,5)
      - - - -
      DO 99 ICOL = 1, 5
       DO 99 IROW = 1, 5
        DO 99 K = 1, L
         DO 99 J = 1, M
          DO 99 I = 1, N
99         B(I,J,K,ICOL,IROW) = A(I,J,K,IROW,ICOL) + ...
```

**Figure 54. Improved Loop and Data Structure**

As Figure 54 shows, the combination of DO statement reordering and data reordering will provide a more efficient structure for vector execution.

Once written in this form, it is clear that we may attempt to collapse the array's dimensions to improve vector length, as discussed earlier. In this example, all three (grid) dimensions might be combined to provide a flexible scheme for transposing matrices on a grid whose dimensions and size may range up to a maximum (combined) length of N*M*L. Each element (IROW,ICOL) is moved for ALL points in the grid, providing what should be effective vector length performance.

# 5.3 Temporary Variables

It is a common practice to use temporary variables in a DO loop to hold intermediate values. The presence of such variables may adversely affect the vectorization of the loop in which they appear. The VS Fortran Version 2 Vectorizing Compiler can recognize and vectorize many uses of temporary variables; however, it helps to understand techniques that can be used to increase the likelihood of vectorization where it is not otherwise recognized by the compiler.

## 5.3.1 Scalar Temporaries

The example in Figure 55 illustrates the use of a scalar temporary variable which contains a partial result. In this context, if these were the only statements in the loop, the compiler would vectorize the loop as if it were written as on the right. (This technique is known as *scalar expansion.*)

```
*  SCALAR TEMPORARY           *  VECTOR TEMPORARY
      DO 1 I = 1, N                 DO 1 I = 1, N
        T = A(I) + B(I)               T(I) = A(I) + B(I)
1       R(I) = T + C(I)/T       1     R(I) = T(I) + C(I)/T(I)
```

**Figure 55. Scalar Temporary Becomes Vector Temporary**

There are two factors to consider in recoding to convert scalar temporaries into vector temporaries. First, a compiler-generated vector temporary can often be kept in a register and need not be stored. If the user creates a temporary array variable, the compiler must allocate storage in the program for the array, and then keep track of its usage. It may be difficult to detect whether a store is needed or not.

Second, the explicit presence of a vector temporary may affect the scalar performance of the loop, because the elements of the temporary array must be stored.

Conversely, the user should recognize that extensive use of such scalar temporaries within a single loop could eventually exhaust the compiler's ability to expand them. A single limit is not known, however, since it depends on the context in which the temporaries appear.

Thus, some judicious experimentation may be needed to assess the benefits of manual scalar expansion.

### 5.3.2 Scalar Array Element References

In Figure 56, the array element A(K) has been used on the left as a scalar array reference. Note that the loop uses A(K) for a temporary assignment, and that the loop variable is "J". When the loop is exited, A(K) has a single value. The last value it attained is B(M)*C(M). "Good programming practice" indicates that the user can save computational effort even in scalar form by removing that element from the loop and performing only the final computation. This approach produces higher quality vectorized code as well.

```
      DO 3 J = 1, M                      DO 3 J = 1, M
        A(K) = B(J)*C(J)           3       R(J) = (B(J)*C(J)-.5)**2
 3      R(J) = (A(K)-.5)**2              A(K) = B(M) * C(M)
```

**Figure 56. Scalar Array Element as a Temporary**

The new loop, on the right, does not have the scalar array element A(K) which it cannot expand since it is addressed using another subscript. As a result, the compiler will vectorize this new loop. The sense of the original loop is retained by establishing the final value of A(K) after completion of the loop. This example assumes that M is not zero, that is, the loop is not a zero-trip DO loop which is permitted in FORTRAN 77.

## 5.4 Subscript Considerations

The next topic involves the use of subscripts that are not simply the loop induction variable, the "DO" loop index. When array variables are subscripted with expressions or with auxiliary computed subscripts, it can be difficult for the compiler to determine unambiguously whether the statements in the loop are vectorizable. A previous example (Figure 36 on page 33) used the MOD function and division by an integer to demonstrate this difficulty.

### 5.4.1 Subscripts With Constant Increments

The example in Figure 57 shows that the constant increment "5" could be placed directly into the subscript of the variable itself.

```
      DO 7 I = 1, N                      DO 7 I = 1, N
        J = I + 5                    7     R(I+5) = 1.0
  7     R(J) = 1.0
```

**Figure 57. Auxiliary Array Subscript With Constant Increment**

Eliminating the auxiliary subscript "J" in the loop avoids the possibility of unnecessary stores. There is also a second advantage: if the code which follows this loop requires the final value of "J", then this loop might not vectorize, since vectorization could require that "J" be considered as an auxiliary induction variable whose final value must be calculated.

Removing the explicit computation of "J" may promote vectorization. If the value of J is required later, the assignment statement `J=N+5` could be placed after the loop to provide the final value.

The greater the complexity and use of auxiliary subscripts, the greater the potential for the compiler to be unable to analyze them for vectorization purposes. Thus, another approach to "good programming practice" is to avoid the unnecessary computation of auxiliary subscripts.

### 5.4.2 Computed Auxiliary Subscripting Variables

Figure 58 on page 49 illustrates a more complex subscript inside the I loop, but only a part of the computation is dependent on that subscript.

```
      DO 13 J = 1, M                    DO 13 J = 1, M
      - - - -                           - - - -
        DO 11 I = 1, N                    INC = (J-1) * N
          K = (J-1)*N + I                 DO 11 I = 1, N
11        A(I) = B(K)             11        A(I) = B(I+INC)
      - - - -                           - - - -
13    CONTINUE                    13    CONTINUE
```

**Figure 58. Computed Array Subscript**

The simplest action is to remove the "I"-independent part from the loop. The new variable "INC" is invariant in the "I" loop, and can be used to index "B".

As a general rule, the less the user creates complicated subscripting structures, the better off he is. The scalar performance might also improve since some operations have been removed.

### 5.4.3 Linearized Multi-Dimensional Subscripts

Subscript expressions of the form illustrated in Figure 58 are sometimes used to map a two-dimensional array onto a one-dimensional array. These "linearized" subscripts may be more difficult for the compiler to analyze than the equivalent multi-dimensional form.

For example, if the code segment in Figure 58 had appeared in a subroutine for which the arrays "A" and "B" and their dimensions were dummy arguments, then the code can be written in a more "natural" form in which the complex subscript expressions can be replaced by single subscripts, as shown in Figure 59.

```
      DIMENSION A(N), B(N,M)
      - - - -
      DO 13 J = 1, M
      - - - -
        DO 11 I = 1, N
11        A(I) = B(I,J)
      - - - -
13    CONTINUE
```

**Figure 59. Eliminating An Auxiliary Subscripting Variable**

The compiler-generated code to calculate the subscripts is essentially the same in the two cases. In this revised format, however, the compiler can more easily analyze both loops as candidates for vectorization.

### 5.4.4 Auxiliary Subscripts with Unknown Increment

```
      COMMON /COM/ INC
      - - - -
      J = 1
      DO 99 I = 1, 100
        A(J) = A(J) + 5.0
 99     J = J + INC
```

**Figure 60. Auxiliary Subscripting Variable With Unknown Increment**

The example in Figure 60 illustrates an ambiguity which might result in a loop which will not vectorize. In this case, J is initially well defined, but it is increased by an increment whose value is externally defined (it is in the COMMON block /COM/). In analyzing this loop, the compiler must be able to know that the value of INC cannot be zero before it will vectorize the loop. Because there is a possibility that INC could be zero, the compiler will assume that the elements of A are not data independent and that no vector operation is possible.

If the increment INC *is* known to be zero and the code is rewritten on that basis, the compiler *will* vectorize the loop! This is shown in Figure 61.

```
      COMMON /COM/ INC
      - - - -
      J = 1
      DO 99 I = 1, 100
        A(J) = A(J) + 5.0
 99   CONTINUE
```

**Figure 61. Same Loop with Constant Auxiliary Subscript**

This loop simply adds the constant 5.0 to A(J) 100 times, and the compiler will generate code to perform a reduction operation (which is discussed in 3.8.1, "Vectorization of Reduction Operations" on page 29).

## 5.5 Recurrence, Part 2

The loop presented in Figure 62 on page 51 shows that complex relationships may exist between elements of an array, and still not contain a recurrence.

```
      DO 99 J = 1,M
        DO 98 I = 1,N
          A(I,J)=A(I-1,J-1)+A(I-1,J+1)+A(I+1,J-1)+A(I+1,J+1)
   98   CONTINUE
   99 CONTINUE
```

**Figure 62. Loops Not Containing a Recurrence**

The diagram in Figure 63 demonstrates some of the dependences of one element upon the others.

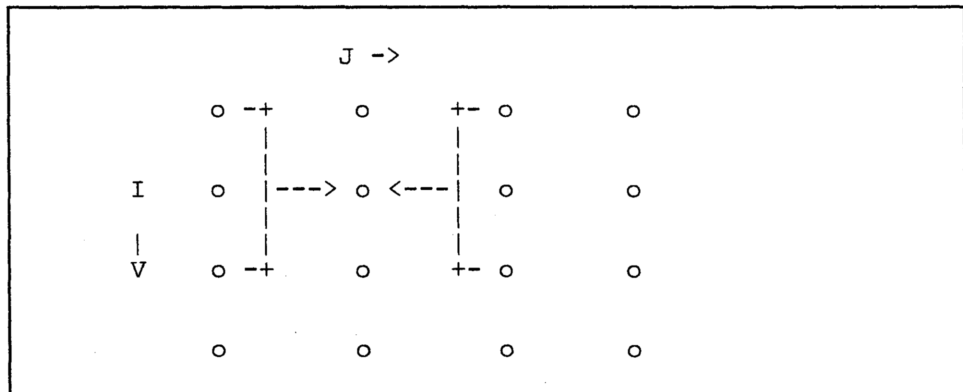



**Figure 63. Subscript Relationships in Previous Example**

Suppose we consider the vectorization of this loop in the "I" direction. Then the A(I,J) might be considered as the "J"th vector over "I". Similarly, A(I-1,J-1) represents a different section of the "J-1"st vector, and A(I+1,J+1) is still another section of the "J+1"st vector, all in the "I" direction. Thus there is no dependence between elements of the "J"th vector, which would be a recurrence; rather, there are relationships between one vector and another, which are not recurrences.

## 5.5.1 Hiding Recurrences

It is possible to effectively hide some recurrences through the use of other "aliasing" techniques such as passing arrays as subroutine arguments. The compiler cannot detect such recurrences, since the range of its analysis is limited to a single routine at a time. The Fortran-77 standard permits such aliasing only if the dummy arguments are "read-only". Thus, a standard-conforming program may not use these "aliasing" techniques to avoid or hide recurrences. (If the apparent recurrences *are* truly "read-only", no recurrence exists, and there is no need for a subroutine call to hide the recurrence!)

Furthermore, such "aliasing" techniques can cause overlaps and unsafe conditions that are not possible to detect since the user is performing them across module boundaries.

## 5.6 Unrolling Loops

Loop unrolling is a familiar technique employed to improve performance on selected scalar machine architectures. The intent of loop unrolling is to increase the proportion of computation in a loop compared to the overhead of the loop's "bookkeeping".

*Vector* computation already provides much more computation in a loop compared to the loop overhead; thus, vectorizing a loop that was unrolled to improve scalar performance could give far less improvement than would be possible if the loop had been left "rolled".

```
* With Unrolling                 *    Without Unrolling
    DO 99 I = 1, N, 3                    DO 99 I = 1, N
      A(I+0) = A(I+0) + B(I+0)             A(I) = A(I) + B(I)
      A(I+1) = A(I+1) + B(I+1)        99 CONTINUE
      A(I+2) = A(I+2) + B(I+2)
 99 CONTINUE
```

**Figure 64. Loop With Unrolling and Without**

The compiler will vectorize the unrolled loop shown in Figure 64 by generating three separate vector instructions, each with a stride of three (3) and a vector length of N. Each instruction will only operate on one third of the N values.

While these are valid vector instructions, improved vector performance may be obtained by recombining the statement sequence into the original single statement, which will be vectorized with a stride of 1 and a vector length of N, as shown on the right. This is an example of a scalar performance coding technique which does not pertain to vector applications.

In the example on the left in Figure 65, the two-dimensional operations are unrolled along the leftmost subscript, in the direction which should probably be vectorized.

```
      DO 99 J = 1, M                    DO 99 J = 1, M, 3
        DO 99 I = 1, N, 3                 DO 99 I = 1, N
          A(I+0,J) = ...                    A(I,J+0) =...
          A(I+1,J) = ...                    A(I,J+1) =...
          A(I+2,J) = ...                    A(I,J+2) =...
 99   CONTINUE                      99  CONTINUE
```

**Figure 65. Loop Unrolled Along Non-Vector Dimension**

Following the technique illustrated in the example in Figure 64, we can recombine the statements in the "I" direction to promote better vector performance. However, there is no reason why the user could not unroll the loop in the dimension controlled by the other index, "J", as shown on the right in Figure 65, since the new statement sequence defines three individual vectors of stride 1, each with length "N". Therefore, unrolling,

so long as it is performed in the non-vector dimension, remains an appropriate coding style for vector programming.

If either loop limit is very small, then is it probably worth unrolling the loop completely, by "expanding" it inline. Then, the DO statement for that loop index can be eliminated, and the compiler can make a better decision about vectorizing the remaining DO loop.

## 5.7 Loop Segmentation

If the computational work is unevenly distributed among the loops in a nest, the compiler might well select the "correct" loop for vectorization, but additional vectorization opportunities might not be accessible.

An example of loop *segmentation* is shown in Figure 66.

```
      DO 15 J = 1, M
      - - -
        DO 5 I = 1, N
        - - -
  5     CONTINUE
        - - -
 15   CONTINUE
```

**Figure 66. Nested Loops Available for Segmentation**

In this nest of two loops, the compiler could select either the inner "I" loop or the outer "J" loop for vectorization, but not both. As the ellipses indicate, considerable work may be performed in the "J" loop which would not be executed in vector mode if the inner, "I" loop were vectorized. Similarly, there might be statements in the inner "I" loop which we would also want to be vectorized even when the outer "J" loop is selected for vectorization.

To improve opportunities for vectorization, the user could manually segment the outer loop into three loops, *each* of which then becomes a candidate for vectorization, as shown in Figure 67.

```
      DO 115 J = 1, M
      - - - -
115   CONTINUE
      DO 215 J = 1, M
        DO 5 I = 1, N
        - - - -
  5     CONTINUE
215   CONTINUE
      DO 315 J = 1, M
      - - - -
315   CONTINUE
```

**Figure 67. Loops After Segmentation**

Of course, the operations within these loops must be sufficiently independent to allow this loop segmentation to be correct.

Loop segmentation must be applied with care, of course. There is always an identifiable cost in vector loop initiation and termination, and in sectioning the arrays within the loop. Thus, for example, it is possible that enough work in the inner loop is already vectorized on the outer loop's index "J" that segmentation would introduce enough extra overhead to cause slower execution! That is, vectorization could be increased, while program speed decreases. As with other vectorizations, timing measurements will reveal the relative merits of each change.

A more specific example of loop segmentation to enhance vectorization is shown in Figure 68. In this example, more work is being performed in the "J" loop than in the "I" loop.

```
      DO 10 J = 1, M                         DO 10 J = 1, M
        A(J) = S * A(J)                        A(J) = S * A(J)
        B(J) = S * B(J)                        B(J) = S * B(J)
        C(J) = A(J) + B(J)                     C(J) = A(J) + B(J)
        D(J) = S * C(J)                        D(J) = S * C(J)
                                          10 CONTINUE
                                             DO 20 J = 1, M
        DO 9 I = 1, N                          DO 9 I = 1, N
          E(I,J)=E(I,J)+D(J)                     E(I,J)=E(I,J)+D(J)
    9   CONTINUE                           9   CONTINUE
   10 CONTINUE                            20 CONTINUE
```

**Figure 68. Loops Before and After Segmentation**

It is clear that as long as the computation of the variable "D" is completed before the computation of the variable "E", these computations may be performed independently. Thus, two loops may be formed from the original outer loop, and the amount of work which is eligible for vectorization is increased.

# 5.8 Statement Reordering

Figure 69 shows an example of statement reordering. The VS Fortran Version 2 Vectorizing Compiler can detect most instances where statement reordering will permit vectorization, and will do the reordering automatically. However, because such vectorizations may not always be visible to the compiler, it may sometimes help to do such reorderings in the source code.

```
      DO 17 I = 1, N              DO 17 I = 1, N
        R(I) = A(I)                 A(I+1) = B(I)
   17   A(I+1) = B(I)          17   R(I) = A(I)
```

**Figure 69. Removing an Order Dependence**

The computation of "R" would appear to depend on the computation of "A" from the preceding trip through the loop. However, a little analysis shows that since the element of "A" which is being computed is the element that the computation of "R" will use on the next cycle of the loop, all of the "A"s could be computed before the "R"s. The computational order may be reversed, and the loop now is visibly vectorizable.

## 5.9 Loop Distribution

The compiler also has the ability to distribute or split a loop by considering the possibility of vectorizing Fortran statements within the loop separately, on a "statement by statement" basis, rather than analyzing the loop as a whole for vectorization. This compiler feature is part of the extra function obtained when the user selects the LEVEL(2) sub-option of the VECTOR compiler option. This process is illustrated in Figure 70.

```
      DO 15 I = 2, N
        AA(I) = AA(I) + B(I)**2
15      X(I) = X(I-1) + Y(I)
```

**Figure 70. Loop Suitable for Distribution**

In this loop, the statement involving the variable "AA" may be safely vectorized, but the statement involving the variable "X" contains a recurrence relationship, which may not. (Recurrences are discussed in 3.5, "Recurrences" on page 24.) The compiler can determine, however, that the two Fortran statements in the body of the loop are independent of each other, and could therefore be processed separately. The compiler then "splits" the original loop into two loops, "distributing" the original loop across the statements, as shown in Figure 71.

```
      DO 15 I = 2, N
15      AA(I) = AA(I) + B(I)**2
      DO xx I = 2, N
xx      X(I) = X(I-1) + Y(I)
```

**Figure 71. Original Loop Is Split Into Two Loops**

The loop containing the computation of "AA" may now be executed in vector mode, while the loop containing "X" remains in scalar mode.

Such vectorizable Fortran constructs may appear in a context which contains other vector inhibitors, or which does not provide enough information to the compiler to analyze the situation, or which is too complex. Thus it is advantageous to adopt a style which isolates the vectorizable from the non-vectorizable computations, in order to promote additional vectorization.

## 5.10 Indirect Addressing

The technique of copying data elements into a temporary vector illustrated in Figure 48 on page 42 might be used for indirectly addressed variables, as shown in Figure 72. Again, we assume that the number of elements, "M", and the amount of work involved in the computation loop between the two data motion loops, is sufficient to justify the copying loops.

It should be noted that this technique may be used only when the condition of no-duplication of values in the list of indirect addresses.

```
      DO 10 I = 1, N
        TEMPA(I) = A(INDEX(I))
   10 CONTINUE
      DO 15 I = 1, N
        - - - -
*       many operations on TEMPA ...
        - - - -
   15 CONTINUE
      DO 20 I = 1, N
        A(INDEX(I)) = TEMPA(I)
   20 CONTINUE
```

**Figure 72. Indirect Addressing With a Temporary Vector**

Indirectly addressed operations are commonly called a "gather" when data is fetched from an indirectly addressed array in storage into a contiguous vector, and a "scatter" when data is stored from a contiguous vector into an indirectly addressed array.

Even though we have discussed only the simplest one-dimensional case for both indirect addressing and conditional operations, these techniques are equally extensible to operations on arrays or matrices. As noted earlier, since Fortran stores arrays in column-major order, it is easy to apply these techniques to the column vectors of an array without further complicating the algorithm. In addition, experience with many applications has resulted in the observation that a common programming practice is to collapse a multi-dimensional array into a singly-dimensioned linear array in order to gain flexibility of use of the application for many array sizes. The required index pointers for this practice may be used to introduce the conditional execution technique as well.

## 5.11 Conditional Operations

We now consider the vectorization of conditional operations. The examples are intended to convey some indication of the difference between styles which work well in scalar mode compared to styles which might be more appropriate to vector execution.

The System/370 Model 3090 Vector Facility provides *masked* operations which operate under the control of "Vector Mask Mode". When Vector

Mask Mode is off, all elements of a vector are used; when Vector Mask Mode is on, any masked operation will operate selectively on designated elements of a vector. The selection mechanism is the Vector Mask Register, which contains a bit sequence whose length is the section size Z. These bits can be set on and off by comparisons and other operations; a masked operation will then operate on only those elements corresponding to "on" bits in the Vector Mask Register.

For example, consider the DO loop in Figure 73.

```
      DO 66 I = 1, N
 66     IF (A(I) .GT. B(I)) C(I) = D(I)
```

**Figure 73. Example of a Conditional Operation**

This loop would be vectorized by using the comparison of the arrays A and B to set the contents of the Vector Mask Register; then, those contents would be used to select the elements of arrays C and D for which the assignment operation is to be performed.

### 5.11.1 Conditional Operations and IF Conversion

The next topic we will consider involves conditional operations. That is, operations which take place under the control of the logical result of some comparison. This topic suggests a specific style of coding, since the vector relational is not currently supported in vector mode, but the resulting logical vector may be saved for use as a "mask" for later vector operations, as in Figure 74.

```
      DO 98 I = 1, N
        LCOND(I) = A(I) .LT. B(I)
   98 CONTINUE
      DO 99 I = 1, N
        IF(LCOND(I)) C(I) = D(I)
   99 CONTINUE
```

**Figure 74. Loops With Conditional Operations**

If the comparison is directly used to control a vector operation, then both the comparison and the conditional operation may be vectorized.

```
      DO 99 I = 1, N
        - - - -
        IF(A(I) .LT. B(I)) GO TO 99
        - - - -
   99 CONTINUE
```

**Figure 75. Loop With Conditional Control**

The example in Figure 75 will be converted from a control dependence to a data dependence, as if it were written as in Figure 76 on page 58.

```
      DO 99 I = 1, N
        - - - -
        IF(A(I) .GE. B(I)) THEN
        - - - -
        ENDIF
   99 CONTINUE
```

**Figure 76. Loop With Data Dependence**

In either case, the loop will be vectorized, and the conditional operation will be executed in vector "masked mode" under control of the bit mask generated by a vector "compare" instruction.

The last example, in Figure 77, shows a specific type of conditional operation which the compiler will not vectorize.

```
      DO 99 I = 1, N
        - - - -
        IF(A(I) .LT. B(I)) C(INDX(I)) = 0.0
        - - - -
   99 CONTINUE
```

**Figure 77. Loop With Conditional Control**

No hardware support is provided for indirectly addressed operations in vector masked-mode. The result is that this type of conditional computation will always be executed in scalar mode.

## 5.11.2 Writing Conditional Code

We will now examine several different examples showing different ways of expressing the same set of conditions. The object of the loop in Figure 78 is to compute a quantity which depends on the SQRT function, while avoiding the square root computation when its argument is negative.

```
      DO 25 I = 1, N
        X(I) = Y(I) + Z(I)
        IF(B(I) .LT. 0.) GO TO 15
        X(I) = X(I) + EXP(SQRT(B(I)))
        GO TO 25
15      X(I) = X(I) + 1.0
25    CONTINUE
```

**Figure 78. Loop Containing a Condition**

In this example, the loop contains a branch around part of the computation which is taken when the argument of the SQRT intrinsic function is

negative. This type of loop contains a *control dependence*, a dependence on the data for a transfer of control of execution. The VS Fortran Version 2 Vectorizing Compiler will use "IF Conversion" to try to turn the "control dependence" into a "data dependence". This results in code like the sequence shown in Figure 79.

```
      DO 25 I = 1, N
        X(I) = Y(I) + Z(I)
        IF(B(I) .GE. 0.) THEN
          X(I) = X(I) + EXP(SQRT(B(I)))
        ELSE
          X(I) = X(I) + 1.0
        ENDIF
   25 CONTINUE
```

**Figure 79. Control Dependence Changed to Data Dependence**

The resulting loop is now vectorized by using a vector comparison to set the mask in the Vector Mask Register, followed by operations under mask.

In fact, the original loop might also have been written as shown in Figure 80.

```
      DO 25 I = 1, N
        X(I) = Y(I) + Z(I)
        IF (B(I) .GE. 0.) X(I) = X(I) + EXP(SQRT(B(I)))
        IF (B(I) .LT. 0.) X(I) = X(I) + 1.0
   25 CONTINUE
```

**Figure 80. Data Dependence With Different Conditions**

In this case, a mask would be constructed for the result of each of the tests, and each of the conditional computations would be performed under a different mask.

Although each of the loops in the preceding examples (Figures 78-80) will provide the same result, the vector performance will vary depending on the coding style employed. CPU timings will help in selecting the style that leads to the best performance.

### 5.11.3 Improving Conditional Code

The next example, in Figure 81 on page 60 combines several of the techniques in the preceding discussions.

```
      DO 30 J = 1, JMAX
        DO 20 I = 1, IMAX
        IF(J.EQ.JMID.AND.(I.EQ.1.OR.I.EQ.IMAX).AND.FLAG)
     X    GO TO 20
          DO 10 K = 1, KMAX
            A(I,J,K) = B(I,J,K) + C(I,J,K)
10        CONTINUE
20      CONTINUE
30    CONTINUE
```

**Figure 81. Loops With Control Dependence**

The computation is representative of a practice that happens routinely in the simulation of physical phenomena of any kind. For some condition, the computation is to be bypassed for specific index values. This may be thought of as a boundary condition on a 3-dimensional grid (I,J,K), where the computation is not to be performed along part of the top and bottom planes of the grid (see Figure 82).

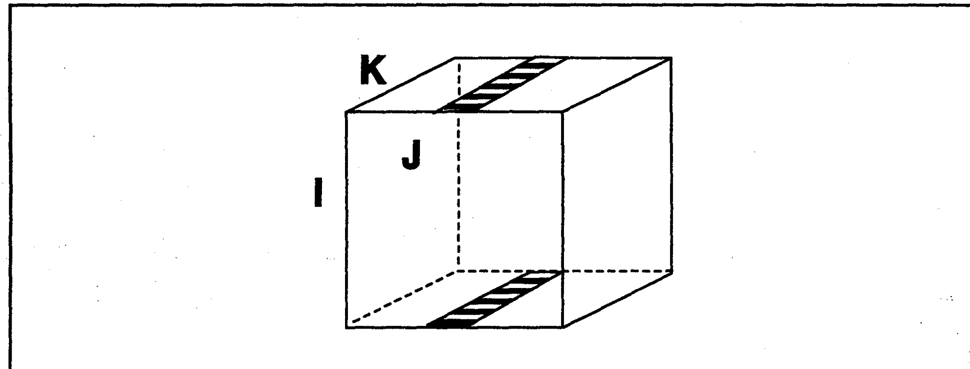



**Figure 82. Computation on 3-Dimensional Grid**

The compiler will analyze this loop and attempt to vectorize the (inner) "K" loop by changing the control dependence to a data dependence as in the example of Figure 79 on page 59. Depending on the dimensions of the arrays in the I,J directions, however, the economic analysis may indicate that the cost of this loop in vector mode, for vectors of length KMAX, potentially at a large stride (IMAX*JMAX), may exceed the cost in scalar mode, and select scalar execution mode as a result.

However, if we examine the condition, we find that part of the condition is independent of the "I" index, and is essentially used to select the limits on the "I" loop. Instead of testing each value of the index "I", we remove the "I"-independent part of the condition from the "I" loop and use it to set the "I" limits outside of the "I" loop. Now we observe that the "I" and "K" loops are order-independent, and may be interchanged. The resulting computation (shown in Figure 83 on page 61) is vectorizable, in the "I" direction.

```
      DO 30 K = 1, KMAX
        DO 20 J = 1, JMAX
          IBEGIN = 2
          IEND = IMAX - 1
          IF(J.EQ.JMID .AND. FLAG)
     X       GO TO 25
          IBEGIN = 1
          IEND = IMAX
25        DO 10 I = IBEGIN, IEND
             A(I,J,K) = B(I,J,K) + C(I,J,K)
10        CONTINUE
20      CONTINUE
30    CONTINUE
```

**Figure 83. Loops With Modified Control Dependence**

The vectors will be of length IMAX or IMAX-2, depending on the result of the "IF" test, and memory references will be in storage order (stride 1). Thus we have modified this loop so that the vector content is clearly "visible" to the compiler. An additional benefit of this modification is that the loop will execute more efficiently in scalar mode as well, since the "IF" test has been simplified, is executed fewer times, and the memory references have been re-ordered to a stride 1 addressing pattern.

The intent here is not to suggest that there is only one method for handling this operation. However, it does illustrate an example of how conditions on the boundaries of computations can be used to determine vector lengths, without introducing unnecessary tests within a loop which act to prevent vectorization. In the end, the user usually generates better scalar code at the same time.

## 5.12 Data Dependent Loops

Data dependent loops proceed until a computed value reaches some limit. While it is possible to vectorize such loops, it should be recognized that a performance gain may not necessarily be realized. For example, the loop in Figure 84 will not vectorize since it contains a branch to a statement outside the range of the loop.

```
      DO 98 I = 1, N
        - - - -
        X(I) = Y(I) - Z(I) * T(I)
        IF (X(I) .LT. 0.0) GO TO 99
        ROOT(I) = SQRT(X(I))
        - - - -
98    CONTINUE
99    ILAST = I - 1
```

**Figure 84. Data Dependent Loop With Branch Out**

However, it can be transformed into a vectorizable set of operations by segmenting the loop into three new loops, as illustrated in Figure 85 on page 62.

```
      DO 96 I = 1, N
        - - - -
        TEMPX(I) = Y(I) - Z(I) * T(I)
   96 CONTINUE
      DO 97 I = 1, N
        IF (X(I) .LT. 0.0) GO TO 98
   97 CONTINUE
   98 ILAST = I - 1
      DO 99 I = 1, ILAST
        X(I) = TEMPX(I)
        ROOT(I) = SQRT(X(I))
        - - - -
   99 CONTINUE
      IF (ILAST.EQ.N) GO TO 101
      X(ILAST+1) = XTEMP(ILAST+1)
  101 CONTINUE
```

**Figure 85. Vectorizable Version of Data Dependent Loop**

The first loop performs the evaluation of the variable to be tested, "X", and all of the work which leads to that computation. The second loop establishes the range of computation (loop limit or vector length) for the third loop which contains the rest of the computation. The ellipses (- - - -) represent some (potentially large) amount of computation, which we will assume will vectorize in the new loop sequence.

In vector mode, all "N" values of the arrays in the first loop will be computed. This means that more work may be performed in the vectorized loop than in the scalar version, which would have terminated at the appropriate condition. The speed improvement realized by executing in vector mode will be diminished by the time spent performing the extra work. In fact, a speed degradation may be encountered depending on the amount of work involved. As a rule of thumb, assuming a vector speed-up of a factor of two, the original loop would have to be performed for N/2 of the computations or more for a speed improvement to be realized. If fewer computations were performed, a loss in performance might result. This first order estimate applies only to this specific loop and ignores the speed improvement obtained from executing the rest of the computations (third loop) in vector mode. The third loop has the same range as the original, so that a speed improvement from vector execution would be expected for this section of code.

Note that a temporary has been introduced for the "X" result in the first loop This is because the original loop only modified the first ILAST+1 values of "X", but the vector loop changes all "N" values. If it is necessary to protect the values from ILAST+2 to "N", a temporary would be required to store the "N" values and the first ILAST of them would be copied into "X" later. This ensures the integrity of "X", except for the single negative value which triggered the original branch. This value is shown to be updated after the last loop completes. If this fix-up is not required, the temporary need not be introduced.

Overall, it is the combination of the extra work performed, the relative distribution of work between the first and last loops, and the vector speed-up which will determine whether an improvement in execution time will be realized. Clearly the greatest benefit will be obtained when the amount of work in the first loop is small, the point at which the original loop is exited is close to the loop limit, and the computation in the last loop is more extensive relative to that of the first.

## 5.13 Loops Containing External References

Loops containing non-intrinsic external references simply do not vectorize. If a CALL statement is present in a loop, the loop will not vectorize. Consider the example in Figure 86.

```
      DO 99 I = 1, M                       DO 97 I = 1, M
        - - - -                              - - - -
        A(I,J,K) = ...                 97    A(I,J,K) = ...
                                           DO 98 I = 1, M
        CALL SUBA (A, ...)                   CALL SUBA (A, ...)
                                       98  CONTINUE
                                           DO 99 I = 1, M
        - - - -                              - - - -
   99 CONTINUE                         99  CONTINUE
```

**Figure 86. Loop Containing a CALL Statement**

(Of course, when Fortran intrinsic functions are used, the compiler automatically provides links to the vector versions of the intrinsics[8], so long as the sub-option NOINTRINSIC is not specified).

Isolation of the CALL statement, as shown on the right in Figure 86, provides one method of improving the vectorization potential of the loop, if the subroutine coding permits. Otherwise, a technique described in the discussion in 6.2, "Incorporating Loops Across Modules" on page 72 may be used.

8 Consult the VS Fortran Version 2 Language and Library Reference (Form Number SC26-4221) for a list of the Fortran intrinsic functions.

## 5.14 Loops Containing Input/Output Statements

```
      DO 99 I = 1, N
        A(I) = . . .
        B(I) = . . .
        - - - -
        WRITE(10) A(I), B(I)
   99 CONTINUE
```

**Figure 87. Loop Containing WRITE Statement**

Input/Output operations do not vectorize. When a DO loop contains I/O statements in what would otherwise a vectorizable loop, as in Figure 87, the I/O statements should removed, as shown in Figure 88.

```
      DO 99 I = 1, N
        A(I) = . . .
        B(I) = . . .
        - - - -
   99 CONTINUE
      WRITE(10) (A(I), B(I), I = 1, N)
```

**Figure 88. Loop With WRITE Statement Moved**

The compiler can then analyze the loop, and vectorize it if it meets the necessary requirements.

## 5.15 Restating an Algorithm

One way to improve vector performance is to restate or reorder an algorithm so as to make optimal use of the data. This is contrasted with either modifying the algorithm to apply it in stages, reordering the data to remove dependencies, or changing the solution technique altogether.

The six possible orderings of the DO statements to perform a matrix multiplication are presented in Figure 89 on page 65.

```
   DO 1 I = 1, M                       |   DO 1 J = 1, P
   DO 1 J = 1, P                       |   DO 1 I = 1, M
    C(I,J) = 0.0                       |    C(I,J) = 0.0
   DO 1 K = 1, N                       |   DO 1 K = 1, N
    C(I,J) = C(I,J)+A(I,K)*B(K,J)      |    C(I,J) = C(I,J)+A(I,K)*B(K,J)
 1 CONTINUE                            | 1 CONTINUE
                                       |
------------------------------------|-------------------------------------
                                       |
   DO 1 I = 1, M                       |   DO 1 J = 1, P
   DO 1 J = 1, P                       |   DO 1 I = 1, M
    C(I,J) = 0.0                       |    C(I,J) = 0.0
 1 CONTINUE                            | 1 CONTINUE
   DO 2 K = 1, N                       |   DO 2 K = 1, N
   DO 2 I = 1, M                       |   DO 2 J = 1,P
   DO 2 J = 1, P                       |   DO 2 I = 1,M
    C(I,J) = C(I,J)+A(I,K)*B(K,J)      |    C(I,J) = C(I,J)+A(I,K)*B(K,J)
 2 CONTINUE                            | 2 CONTINUE
                                       |
------------------------------------|-------------------------------------
                                       |
   DO 2 I = 1, M                       |  DO 2 J = 1, P
   DO 1 J = 1, P                       |  DO 1 I = 1, M
    C(I,J) = 0.0                       |   C(I,J) = 0.0
 1 CONTINUE                            | 1 CONTINUE
   DO 2 K = 1, N                       |  DO 2 K = 1, N
   DO 2 J = 1, P                       |  DO 2 I = 1, M
    C(I,J) = C(I,J)+A(I,K)*B(K,J)      |   C(I,J) = C(I,J)+A(I,K)*B(K,J)
 2 CONTINUE                            | 2 CONTINUE
```

**Figure 89. All Six Ways to Multiply Two Matrices**

These orderings correspond to the data addressing patterns illustrated in Figure 90 on page 66.

**EXAMPLE**

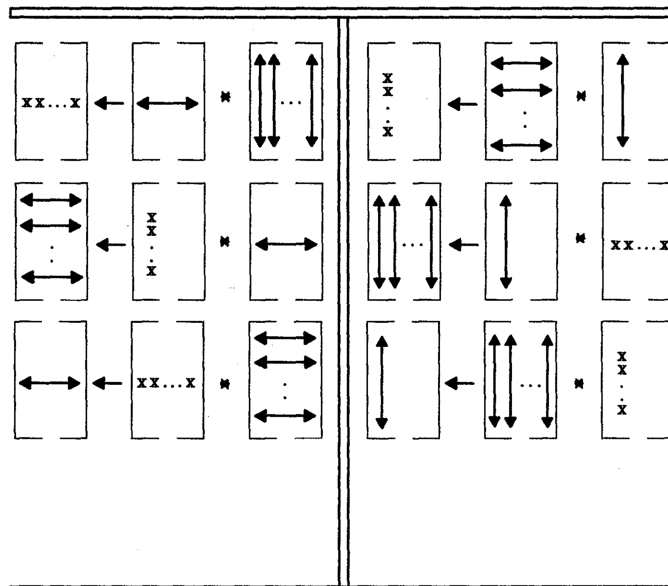



**Figure 90. Visualizing Matrix Multiplication**

In the upper left of each figure appears the standard row-column matrix multiply, with the (scalar) elements of the result matrix being developed in row order. The pattern on the top right is also a row-column matrix multiply, developing the (scalar) result in column order. The ordering appearing in the lower right is the preferred one. Here the product is performed as a scalar-vector multiply (with the scalars in column order), followed by a vector-vector add, to develop a column vector of the result matrix. This operation sequence can be mapped onto the vector-scalar "Multiply And Add" compound instruction, one of the fastest instructions in the System/370 Model 3090 Vector Facility.

## 5.16 Vector Optimizations

Optimization of code undergoing migration to vector execution should be performed only after the code migration and application validation have been completed. In addition to the usual scalar optimizations, there are a few vector-specific optimizations which may be helpful. Some of these considerations are:

- The usual scalar optimizations still apply:
  - Strength Reduction
  - Constant Propagation
  - Dead Code Elimination
  - Constant Sub-expressions
- Vector-specific optimizations

- Redundant operations on vector sections
- Re-computation of indirect index lists
- Dense vs. Sparse operations

### 5.16.1 Vector Sub-Sections

The concept of a vector section can have several meanings. One is, of course, the model-dependent hardware sectioning by which vectors of any length are processed. A second type of vector section might be called a "sub-section", a sub-vector which is contained within a vector.

For example, suppose that an operation such as multiplication is to be performed in two different places in a code. We assume that each multiplication ranges over a different sub-vector of some longer vector, but both use the same scalar multiplier. Figure 91 illustrates this situation.

```
      DO 10 I = 1, N-2
10      X(I) = A(I) * S
      - - - -
      DO 20 I = 3, N
20      X(I) = A(I) * S
```

**Figure 91. Subsets of Vectors**

In the first DO loop, only the first N-2 elements of "A" are multiplied and (presumably) used afterwards. In the second loop, the last N-2 elements of "A" are required. Since the two sub-vectors of "A" result from the same computation, and since they overlap (we assumed that N is large), some of the computations are redundant. A savings may be realized by performing the initial operation over all required values of "A", and addressing the result as needed, as shown in Figure 92.

```
      DO 10 I = 1, N
10      X(I) = A(I) * S
```

**Figure 92. Eliminating Subsets of Vectors**

The number of operations saved in this case would be N-4 multiply operations, which would represent a time savings in both scalar and vector modes.

Thus our focus has been changing from the scalar point of view, in which we consider individual elements, to a vector view, where the "object" of our attention is the string of elements called a "vector". This is the basis for the saying, "THINK VECTOR".

## 5.16.2 Indirect Addressing

Another optimization is to avoid the computation of indirect index lists, so-called "list vectors". This should follow from the observation that for many applications, the geometry, topology or connectivity of the problem is fixed for the duration of computation. A judicious ordering of the conditions which result in indirect address lists, from the broadest condition towards the more restrictive will promote a minimum of recomputation.

There are varying techniques associated with dense vs. sparse vector operations. Frequently different parts of a computation are required to be executed conditionally, affecting only selected elements of a given vector or set of vectors. These conditions may result from the physical problem itself, range of validity of a model of a process, the mathematics which describe be the process or the numerics of the computation.

The mechanisms of both indirect addressing and masked-mode computation are appropriate to the selection of those elements of a vector upon which to operate. Although indirect addressing is the more general indexing scheme, it is also the more expensive.

Indirect addressing can be used in situations where the list of elements is in some random order, where masked mode cannot. As a rule of thumb, if the condition can be applied to the elements of a vector in monotonic order, and the number of elements selected is a reasonable fraction of the total vector, then vector masked-mode operations are generally more efficient. Vector masked-mode operations result from conditional execution of Fortran computations in vector mode.

## 5.16.3 Improving Vector Density

The programming style used to perform conditional computations is a particularly good opportunity for the user to make use of information about the behavior of the application to improve performance beyond what would result from the "brute-force" or more direct expression of the conditional operations in Fortran. For example, consider a computational process in which many vector computations are to be performed conditionally, starting from a small number of input vectors and resulting in a small number of output vectors, and in which, say 10% of the elements are involved. Rather than perform all of the vector computations under a mask (the logical condition which controls the execution) perhaps it may be more efficient to create new, auxiliary vectors consisting only of the affected elements of the original vectors. Then the succeeding computations may be performed on stride-1 (storage order) vectors, whose length is (by our assumption) 10% of the original. Assuming that there is sufficient work to be performed, and that this shortened length is sufficient to provide effective vector utilization, then a performance improvement may be realized.

One way of determining whether this method should be selected is to count the potential length of the conditionally selected elements. An example of this technique is shown in Figure 93 on page 69.

```
      ICOUNT = 0
      DO 10 I = 1, N
        IF (logical expression) ICOUNT = ICOUNT + 1
   10 CONTINUE
```

**Figure 93. Counting Conditional Selections**

Once the number has been determined, and assuming that it is large enough, then the new vectors might be constructed as shown in Figure 94.

```
      J = 0
      DO 10 I = 1, N
        IF (logical expression) THEN
          J = J + 1
          X(J) = A(I)
        ENDIF
   10 CONTINUE
```

**Figure 94. Compressing Vector A Into Vector X**

The subsequent operations on the new vectors may then be said to act on them in "compressed" form.

Similarly, when the "compressed" operations are complete, results may be replaced in the original vector in one of two ways. The first method is an "expand" function: that is, the selected elements are expanded according to the condition on the original vectors, with the intervening elements set to zero. An example of this zero-fill expansion is shown in Figure 95.

```
      J = 0
      DO 10 I = 1, N
        A(I) = 0.0
        IF (logical expression) THEN
          J = J + 1
          A(I) = X(J)
        ENDIF
   10 CONTINUE
```

**Figure 95. Expanding Vector X Into Vector A, Zero Filler**

The second method is the replacement of the new selected element values according to the condition, without disturbing the intervening elements of the original vector, as shown in Figure 96 on page 70.

```
      J = 0
      DO 10 I = 1, N
        IF (logical expression) THEN
          J = J + 1
          A(I) = X(J)
        ENDIF
   10 CONTINUE
```

**Figure 96. Expanding Vector X Into Vector A, With Replacement**

# 5.17 Local Vectorization Techniques: Summary

This completes the discussion of purely local vectorization techniques. Many of the prevalent Fortran coding practices have been presented, although by no means all. It is useful to summarize the various practices in a list, to serve as a basis for the vector migration methodology we have been working towards.

- Isolate Non-Vectorizable Constructs
  - CALL
  - Recurrences
  - Input/Output
  - Relationals
  - Hazards
- Simplify Subscripts
- Reverse Unrolling
- Loop Segmentation
- Statement Re-ordering
- Loop Distribution
- IF Conversion
- Improve Vector Density
- EQUIVALENCE for Longer Vectors
- Opportunity to Use Vector Library (ESSL)

As the discussion has progressed, the list of local vector migration considerations has grown. Although it appears as the last item on the list, the practice of looking for the opportunity to use ESSL routines in place of scalar code should be continuous throughout the migration process. This provides an efficient and easy way of gaining vector performance for minimum effort. ESSL routines may be introduced at low functional levels through the use of the Basic Linear Algebra Subroutines, (the BLAS) such as SYAX or SAXPY, or at a more complex functional level such as a Real-to-Complex, 2-Dimensional Fast Fourier Transform (SRCFT2) or time-varying Recursive Filter (STREC).

# 6.0 Global Migration Considerations

The basic questions to be asked when considering more global modifications of an application program include:

- Is a global restructure of the application necessary?
- Will restructuring provide a performance improvement due to increased vectorization?
- Can restructuring be accomplished in a realistic time with a realistic effort?

In order to make a knowledgeable decision, it is necessary to have an understanding of the overall structure, the static and dynamic ordering of the logic, and the intent of the application.

What determines the possibilities for the data ordering which will exhibit the maximum vector content? Some of the more important choices include:

- the way in which the discrete values are represented (grid)
- the solution technique used (LSOR, L-U Decomposition, Gaussian Elimination, etc.),
- the algorithmic scheme,

and also includes consideration of such items as

- the resulting vector length, stride, and increased size of the application due to the usual expansion of scalar variables to vectors.

The following discussion presents some simple concepts which may be applied to application reorganization for vector migration. It is by no means complete; for example, no attempt has been made to cover the many numerical and algorithmic schemes and techniques in current use. These concepts are mentioned only to place them in their proper order in the migration process. The descriptions are somewhat general, taking in detail only some examples to illustrate particular code structuring. The art of the problem solving process still must remain with the user.

## 6.1 Global Restructuring

The original application and module organization may not have been designed to satisfy the objective of efficient vector execution, and it may be appropriate at this stage to consider the more global aspects of the migration process. This includes both module (subroutine) and data reorganization and may involve redistributing function between the routines. Reorganization implies that modules may have to be either combined or separated or both to make the required data independence visible to the compiler.

A reorganization of the modules may have implications in terms of the communication between the modules, such as common areas, argument lists, the data structure itself, and the order in which it is used. The task is to determine what controls the order in which the computation will take place.

- Is the computational order determined by the representation of the physical space being simulated?
- Is the numerical scheme explicit or implicit?
- Is the solution method setting the order or does the algorithm control?

The algorithm may be reordered for example, to perform computations at many points in stages, rather than completing the whole algorithm for one point.

On the other hand, constraints may be placed on the application by the algorithm, underlying mathematics and the like which do not permit reordering, or result in such sparse or short vectors that the techniques discussed here may have small effect. Thus, it is necessary that the basic message of this report, "Don't Give Up", should be properly interpreted in these situations, and expectations for vector performance improvement realistically adjusted.

## 6.2 Incorporating Loops Across Modules

One technique used is to incorporate a loop structure across several modules or subroutines. Incorporating a loop across modules may be required if a loop exists that calls for a series of subroutines and drives them over say, one or more dimensions. Incorporating the loop inside each of these subroutines may be preferred since the "DO" loops formed within each subroutine become eligible for vector analysis. As discussed earlier, mentioned, this is the situation when one particle, point or element is processed at a time, routine by routine, and function by function.

An example of incorporating loops across modules is illustrated in Figure 97 on page 73. This example is a loop which includes a CALL to (at least one) subroutine. That subroutine has the loop index (the induction variable of the loop) as one of its arguments. Within the subroutine, the items are addressed using that loop index, but only one at a time. The loop

within the subroutine is seen to contain a recurrence with respect to the second subscript, "J".

```
      DO 10 I = 1, N
        - - - -
        CALL SUB1 (..., I, X, Y, ...)
        - - - -
   10 CONTINUE


      SUBROUTINE SUB1(..., I, X, Y, ...)
      DIMENSION X(N,M,KM), Y(N,M,KM)
      - - - -
      DO 20 J = 1, M
        DO 20 K = 1, KM
          Y(I,J,K) = Y(I,J-1,K) * X(I,J,K)
   20 CONTINUE
```

**Figure 97. Loops Distributed Across Modules**

The loops over "J" and "K" are observed to be interchangeable, and the loop is vectorizable over "K". The vector operation over "K" would be performed at a stride of N*M.

An improvement might be realized by splitting the loop in the calling routine into three loops. In the absence of other vector inhibitors, the first loop should vectorize, since the CALL statement no longer appears within it. The last loop should also vectorize for the same reasons. The middle loop may then be incorporated into the subroutine, with appropriate dimensioning and adjustment of the argument list(s). The order of the loops is also reversed, so that the "I" loop is the inner-most, and the vector addressing will be at a stride of 1. The final result is shown in Figure 98.

```
      DO 10 I = 1, N
        - - - -
   10 CONTINUE
      CALL SUB1 (..., X, Y, N, ...)
      DO 20 I = 1, N
        - - - -
   20 CONTINUE


      SUBROUTINE SUB1(..., X, Y, N, ...)
      - - - -
      DO 20 J = 1, M
        DO 20 K = 1, KM
          DO 20 I = 1, N
            Y(I,J,K) = Y(I,J-1,K) * X(I,J,K)
   20 CONTINUE
```

**Figure 98. Loops Incorporated Into a Single Module**

If the function represented by the subroutine "SUB" in the example was essentially the same as provided by an ESSL function, then the loop splitting would permit the loop with the CALL to be replaced by a single a call to the ESSL routine without actually having to incorporate the loop into the subroutine.

## 6.3 Changing the Solution Method

Let us assume that the only opportunity for increased vectorization is through the replacement of the solution method itself. Before expending a great deal of time and energy a caveat is in order. Because a large effort has usually been invested in the selection of the solution method and supporting algorithm in the first place, the user leaves himself vulnerable to extra testing and validation if he chooses to use a new solution method or algorithm. The experience gained with the older algorithm and understanding of how it performs under various conditions will have to be re-learned with the replacement.

Consequently, considerable testing may have to be performed if the user changes the semantics, that is, the way in which the problem is solved. Even if a more vectorizable method is deemed appropriate, other properties of the new algorithm must be considered. For example, a more vectorizable method which converges to a solution more slowly than the older method may not provide any performance improvement at all! It is strongly recommended that the user make use of the large body of literature available before taking any extreme measures[9].

Lastly, it should be noted that many algorithms have been shown to benefit from application of the algorithm in stages (an easier task than replacing the solution technique) or from modifications to the data order. Such schemes as "even-odd", "red-black", "multi-color" or diagonal data orderings have proven, under appropriate circumstances to promote vectorization of selected solution techniques, accompanied by performance improvements due both to the vector execution and to improved numerical behavior.

9 See Appendix B, "References" on page 81.

# 7.0 Summary

All of these considerations can be summarized by a few major concepts which describe the vector migration methodology which we set out to define.

- Simple vectorization will work if the code
  - is well posed for vector,
  - expresses vector relationships simply,
  - has no dependences.
- Simple restructuring may benefit.
- Global restructure may benefit,
  - but is much more difficult!
- Understanding is required for restructuring.

Vectors are, from the total application program point of view, a micro-scale concept, since they are defined on the basis of individual "DO" loops. So-called "simple" vectorization, that is, merely applying the VS Fortran Version 2 Vectorizing Compiler with the VECTOR option to an existing Fortran program without modification, is only one method of achieving vector performance. This method will work if the code is already well-posed for vector execution, expresses relationships simply, and contains a minimum amount of data dependence. Simple loop modifications may help.

Coding style, data organization and module organization are all important factors which may require more global modifications to be performed. Changes to the over-all logic or data structure of the application, including modification or replacement of the solution technique or algorithm are all appropriate measures which can improve vector performance. It must be recognized, however, that they are more difficult than simple "DO" loop modifications, and will require a thorough understanding of the application program to accomplish.

It is important to note that all of these techniques are not usually required. Some combination of "simple" vectorization, and local loop modifications may be all that is required to achieve the desired performance. The more extensive modifications should be undertaken only when the potential benefit is sufficiently high to warrant the effort.

Finally, regardless of the level of modification attempted, we must continually keep in mind that while the VS Fortran Version 2 Vectorizing Compiler is the primary means by which we access the System/370 Model 3090 Vector Facility, programmer understanding and intervention is the key to a successful vector migration.

# Appendix A. Glossary of Terms and Concepts

**Basic Linear Algebra Subroutines (BLAS)**

The BLAS are public domain codes to perform standard Linear Algebra operations. They were originally implemented in scalar FORTRAN. The Engineering and Scientific Subroutine Library contains a subset of the BLAS which have compatible calling sequences with these scalar routines. Therefore, for programs that already use BLAS calls, the ESSL BLAS provide an easy migration path to vector exploitation.

**Column-Major Ordering**

In Fortran, arrays are stored in such a way that the leftmost subscript cycles most rapidly. Thus, A(6,7) is adjacent in storage to A(5,7).

**Computer Scalar** (See following "Scalar")

**Computer Vector** (See following "Vector")

**Data Independence**

The data elements of a computer vector are *data independent* if every element within the vector can be operated on independently of every other element.

**Engineering and Scientific Subroutine Library (ESSL)**

The Engineering and Scientific Subroutine Library is a set of high-performance mathematical programs which exploit the IBM Vector Facility for the 3090 processor. The library consists of 95 subroutines widely used in engineering and scientific computations.

**Performance Improvement (P)**

The ratio of a program's total execution CPU time in scalar mode to its execution CPU time when executed in mixed scalar-vector mode. (Note that this quantity is measurable and repeatable.) Also called "Job Speedup."

**Recurrence**

A relationship among the elements of a computer vector which prevents data independence, in which the value of an element computed later in the sequence depends on the value of an element computed earlier in the sequence.

**Recursion**

(1) Recurrence.

(2) A calling sequence which causes a routine to (directly or indirectly) call itself.

**Scalar**

(Webster) A quantity (as mass or time) that has magnitude describable by a real number.

**Computer Scalar**

A datum stored in a computer's memory. A variable with no DIMENSION declaration, or a single element of an array.

**Storage Order**

A computer vector is arranged in *storage order* if successive data elements are taken from a sequence of adjacent storage locations.

**Stride of a Vector**

The addressing increment between successive elements of a computer vector, divided by the element length.

In Fortran terms, the increment in the leftmost subscript position that would reference successive elements of the vector.

**Vector**

(Webster) A quantity that has magnitude and direction and that commonly represents magnitude and whose orientation in space represents the direction; broadly: an element of a vector space.

(ISO[10]) A quantity represented by an ordered set of numbers.

---

10 International Standards Organization.

**Computer Vector**

A set of scalar data items, all of the same type, stored in a computer's memory. Usually, the set is ordered, and the elements of the set are frequently arranged so as to have a fixed addressing increment between successive elements.

**Vector Facility Hardware**

Special registers and circuity to process computer vector data. Special Arithmetic-Logic Units (ALU's) are used to exploit the possibility of repetitive arithmetic execution ("pipelining") on the elements of a vector. (This capability is available on the IBM 3090 Vector Facility.)

**Vector Processor**

A computer (e.g., the IBM System/370 Model 3090 Vector Facility) with a set of vector instructions and Vector Facility Hardware.

**Vectorization**

(1) The vectorizing compiler's actions in analyzing Fortran programs and producing object code to execute on the System/370 Model 3090 Vector Facility.

(2) The activity of modifying and adapting an application program to assist the vectorizing compiler in exploiting the Vector Facility for that program.

**Vectorization Hazard**

A real or apparent lack of data independence which prevents (or may prevent) vectorization of a loop.

**Vectorization Ratio**

The fraction of a program's (scalar) execution CPU time which, following its vectorization by the vectorizing compiler, is then executed on the Vector Facility. (Note that this quantity is measurable and repeatable.)

**VS Fortran Version 2 Vectorizing Compiler**

An optimizing and vectorizing compiler that supports the IBM System/370 architecture, with or without the Vector Facility. The VS Fortran Version 2 compiler performs the analysis, vectorization, and code generation required to exploit the Vector Facility in an efficient manner.

## VS Fortran Version 2 Library

The Vector Fortran Library component of VS Fortran Version 2 is a high-function execution-time Fortran library which supports the System/370 Model 3090 Vector Facility.

# Appendix B. References

Calahan, D., Buning, P., Ames, W. "Sparse Matrix & Other High Performance Algorithm for CRAY-1", Dept 124, SEL, Univ. Mich. Jan 1979

Dongarra, J. "Performance of Various Computers Using Standard Linear Equations Software in a Fortran Environment" Argonne National Laboratory Report MSCD-TM-23, May 1985

Gajski, D. "An Algorithm for Solvıng Linear Recurrence Systems on Parallel and Pipelined Machines" IEEE Trans. on Comp., March 1981

Hwang, K.,Briggs, F. "Computer Architecture and Parallel Processing", McGraw Hill, 1984

Kogge, P.M. "Algorithm Development for Pipelined Processors" Proc 1977 International Cont. Parallel Proc., IEEE No 77 CH1253-4C Aug 1977

Kuck, D., Lawre, D., Samek, A. "High Speed Computer and Algorithm Organization" Academic Press, 1977

Kulisch, V.W., Miranker, W.L. "A New Approach to Scientific Computation" Academic Press, NY 1983

Nolan, J., Kuba, D., Kascic, M. "Application of Vector Processors to the Solution of Finite Difference Equations", AIME 5th Symp. Reservoir Simulation, Feb 1979

Paul, G. "Large-Scale Vector/Array Processors" IBM Research Dept. RC7306 Sept 78

Rice, J. "Matrix Computations and Mathematical Software" McGraw Hill NY 1981

Rodrique, G., Giroux, E., and Pratt, M. "Perspective on Large-Scale Scientific Computations" IEEE Comp, Oct 1986

Stone, H., "An Efficient Parallel Algorithm for the Solution of a Tridiagnoal Linear System of Equations" J.ACM V20, 1973

Stringer, J. "Efficiency of D4 Gaussian Elimination on a Vector Computer"

Note: Staples can cause problems with automated mail sorting equipment. Please use pressure sensitive or other gummed tape to seal this form.

Cut or Fold Along Line

Vectorization and Vector Migration Techniques — Technical Bulletin

# READER'S COMMENT FORM

**Order No. SR20-4966-0**

This form may be used to communicate your views about this publication. They will be sent to the author's department for whatever review and action, if any, is deemed appropriate.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation whatever. You may, of course, continue to use the information you supply.

Note: Copies of IBM publications are not stocked at the location to which this form is addressed. Please direct any requests for copies of publications, or for assistance in using your IBM system, to your IBM representative or to the IBM branch office serving your locality.

Name ____________________

Address ____________________

____________________

**Thank you for your cooperation. No postage stamp necessary if mailed in the U.S.A. (Elsewhere, an IBM office or representative will be happy to forward your comments.)**

SR20-4966-0

## Reader's Comment Form

Cut or Fold Along Line

Fold and tape    Please Do Not Staple    Fold and tape

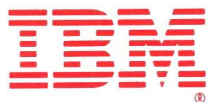
**BUSINESS REPLY MAIL**

FIRST CLASS    PERMIT NO. 40    ARMONK, N.Y.

POSTAGE WILL BE PAID BY ADDRESSEE:

International Business Machines Corporation
Publishing Services - Department 78L/Tower II
225 John W. Carpenter Freeway - East
Irving, Texas 75061

Fold and tape    Please Do Not Staple    Fold and tape



SR20-4966-0

Vectorization and Vector Migration Techniques Printed in USA SR20-4966-0