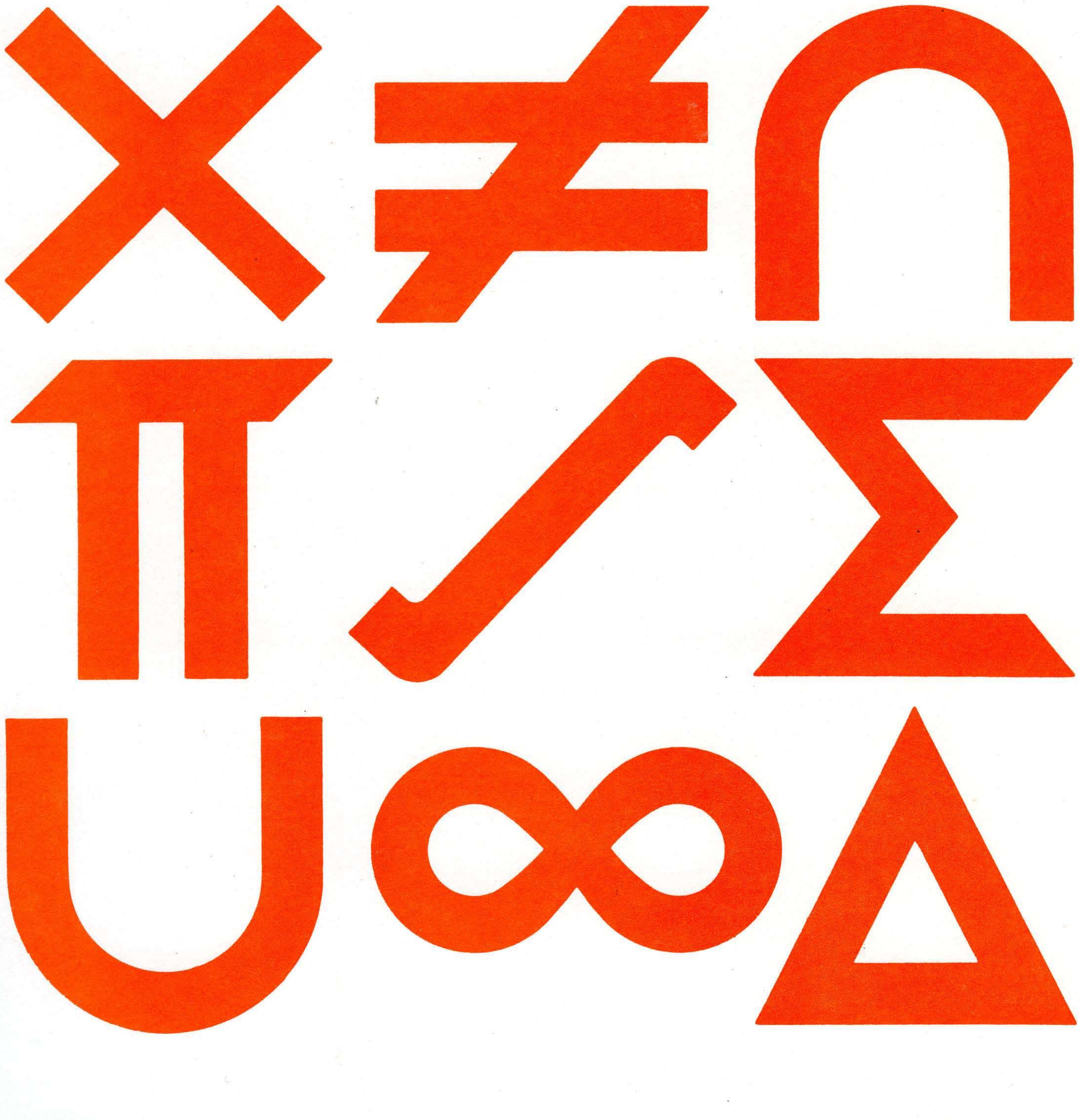
IBM PALO ALTO SCIENTIFIC CENTER
G320-3489, March 1986

Introduction to Vectorizing Techniques on the IBM 3090 Vector Facility

H. H. Wang



Introduction to
Vectorizing Techniques on the IBM 3090 Vector Facility

H. H. Wang

Document Number G320-3489

February 1986

IBM Palo Alto Scientific Center
1530 Page Mill Road
Palo Alto, California

Abstract

The advantage of vector processing is first reviewed. Then the IBM 3090 Vector Facility and software tools are briefly discussed. The major part of the report is devoted to the methodology of choosing algorithms on the target vector machine. We illustrate the method of evaluating algorithms and point out various good practices with examples in linear algebra problems commonly encountered in science and engineering.

Contents

Vector Processing Preliminaries

A vector computer is designed to speedup repetitive but independent computations applied to large arrays of data. The elements of these arrays are usually arranged regularly as vectors or matrices in the sense as understood in linear algebra. More specifically, in data processing terminology, a ***vector*** is an array of data whose successive elements are stored either in consecutive store locations or in locations separated by a constant stride. Thus, rows and columns as well as diagonals of a matrix are common examples of vectors. So are tabulated measurements of physical variables such as temperature, and pressure.

The main approach to gain speed in a vector computer is ***pipelining***. The term ***pipelining*** refers to the design technique that subdivides a basic operation into sub-operations each of which is performed by a special hardware in the same fashion as an assembly line in industrial manufacturing. The ***pipelining*** principle has been applied to memory access and instruction decoding since early 1960's. ***Pipelined*** arithmetic units were first included in machines like the System 360 Model 91 and others. However, these machines are scalar machines because their arithmetic instructions are executed with only one pair of operands. The first commercial vector machine was the Texas Instruments Inc. ASC (delivered in 1972) with hardware instructions which accept vectors as operands.

A typical floating-point add pipeline with four segments (compare exponents, align fractions, add fractions, and normalize) is shown in Figure 1.

When a vector instruction is issued, it initiates the stream of operands (assuming two source vector operands) to the pipeline, each segment of the pipeline accepts a pair of operands, performs its particular function, passes the result to the next segment, and receives the next pair of operands from the stream. At any instant during the execution, several pairs of operands are being processed concurrently in the pipeline. The net effect is that there is an initial delay to complete the first result, called the startup time, but each subsequent result follows quickly since it is only one segment away. Thus if the operands can be delivered to the pipeline in steady streams, the time for the completion of a vector instruction producing N results is given by

$$T_v = S + Nt, \qquad (1)$$

where S is the start up time, t is the time for each segment to complete its task. The startup time can be thought of as an overhead of a vector instruction. Its significance diminishes as the vector length N becomes large. To enhance the effective bandwidth of data flow to match the execution speed of the arithmetic unit, a high speed buffer memory (cache) is included as a part of the IBM 3090 VF, although a highly interleaved memory is often employed on other systems.

Another approach to gain in speed is the technique called ***chaining*** in which multiple pipelines are linked together and operate as a single long pipeline. ***Chaining*** is offered on the IBM 3090 VF in the form of

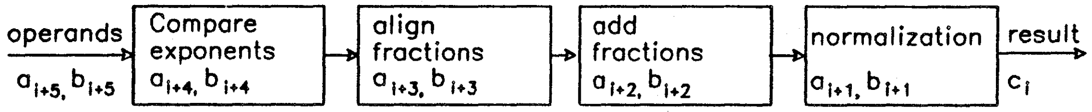


Figure 1: A floating-point add pipeline. Vector add: C = A + B.

compound instructions, ***multiply and add*** and ***multiply and accumulate*** instructions. Besides enabling the full use of the pipelines, vector instructions also substantially reduce the need to execute branch instructions.

The speed of a scalar computer is usually measured by the number of instructions executed per unit time, such as 'million instructions per second' (***MIPS***). For a vector processor, which is primarily designed for scientific computation, it is universally accepted to measure the speed by the number of useful (in the user's point of view) arithmetic operations performed per unit time, such as the use of million floating-point operations per second (***MEGAFLOPS***, or ***MFLOPS***). There is no fixed relationship between the two measurements. In general, it takes two to five instructions to perform a floating-point operation on a scalar machine.

For detailed information on various parallel/vector architectures, the reader is referred to the book by Hockney and Jesshope [1] and the book by Hwang and Briggs [2].

Vectorization

Vectorization means different things to different people. To a language developer, *vectorization* is designing array-like language extensions to FORTRAN such as VECTRAN [3]. To a compiler writer, it means analyzing the dependencies of the statements of the source code (DO-loops) and convert as much sequential operations to equivalent vector operations as economically feasible. To a user, *vectorization* means the introduction of hardware vector instructions into his program so that the high speed of these instructions can be effectively utilized. In general, a user has following three choices to do this:

1. He may choose to vectorize his application simply by recompiling his scalar FORTRAN code using the vectorizing compiler; or

2. he may obtain more benefit by first restructuring his code so as to assist the compiler to recognize more opportunities for generating vector code; or

3. he may recode completely his application by choosing or devising a new algorithm to reap the most benefit from the target vector machine.

We shall restrict our discussion on techniques of *vectorization* from a user's point of view.

Traditionally, on a scalar machine, the fastest algorithm is one that requires the least amount of arithmetic operations. This is not necessarily true anymore on a vector computer. An algorithm with higher operation count but which vectorizes well may outweigh the cost of extra arithmetic operations when executed in vector mode. An example of this is the tridiagonal system solver (discussed below), where the vector algorithm requires about twice as many operations as the scalar algorithm, but is faster than the scalar algorithm executing in scalar mode for sufficiently long vectors. The performance of a vector code also depends on the average vector length, and the startup time of the vector instructions. After an application is vectorized, it is natural to try to measure its performance in some way. The most commonly accepted measure is *speedup*, which is frequently defined as

$$P = \frac{\text{scalar execution time}}{\text{vector execution time}} \qquad (2)$$

This definition works fine when one vectorizes his application using the first two methods. However, ambiguity arises when vectorization is achieved using a different algorithm. In this case, he is more interested in comparing the execution speed of the scalar algorithm with that of the vector algorithm under consideration. So, a better definition of *speedup* is

$$P' = \frac{\text{scalar execution time using the best scalar algorithm}}{\text{vector execution time using the vector algorithm}} \qquad (3)$$

This shows that an execution using a highly vectorizable algorithm with a high ***speedup*** factor P does not necessarily mean a large improvement over the scalar execution.

We want to mention another definition of ***speedup*** which reveals clearly the role of scalar operations in a vectorized code. Let v denote the percentage of arithmetic operations in the code that can be executed by vector instructions, then the ***speedup*** is given by

$$P = \left[1 - v + \frac{v}{R}\right]^{-1} \tag{4}$$

where R is the ratio of vector speed over the scalar speed. If $v = 0.5$, or in another word, if the program is 50% vectorizable, then the speedup cannot exceed the value of 2 even if the vector operation is infinitely fast. This phenomenon was first discussed by Amdahl in 1967 [4] and is known as ***"Amdahl's law"***. Figure 2 shows the ***speedup*** as a function of v for several representative values of R. Experience show that for most applications the values of v fall between $0.5 - 0.9$. This means that the ***speedup*** for most programs will be limited in the range of two to ten no matter how fast the vector speed. However, if the cost of vector hardware is only a small percentage of the total cost of a complete scalar/vector processing system, then the case for the additional hardware which can double or triple the speed of many programs is easily made.

In what follows, we first briefly discuss, for completeness, the 3090 Vector Facility and software tools available. We then devote the rest of the report to our main purpose: to discuss the methodology of choosing algorithms on the target vector machine. We illustrate the method of evaluating algorithms and point out various good practices with examples in linear algebra problems commonly encountered in science and engineering.

IBM 3090 Vector Facility

The 3090 Vector Facility is an integral part of the CPU. Its main features are:

- 16 128-word vector registers (or 8 double-word vector registers)
  - For both standard and long floating-point data as well as 32-bit integer data.
- 128-bit long Vector Mask Register
  - To hold the results of vector compare and to serve as argument or result of other logical operations and to use as mask for mask controlled operations.
- 63 vector instructions (171 opcodes)
  - Many instructions with 3 operands and one of them can be from memory.
  - ***multiply and add*** and ***multiply and accumulate*** instructions to produce one multiply-and-add result per machine cycle.

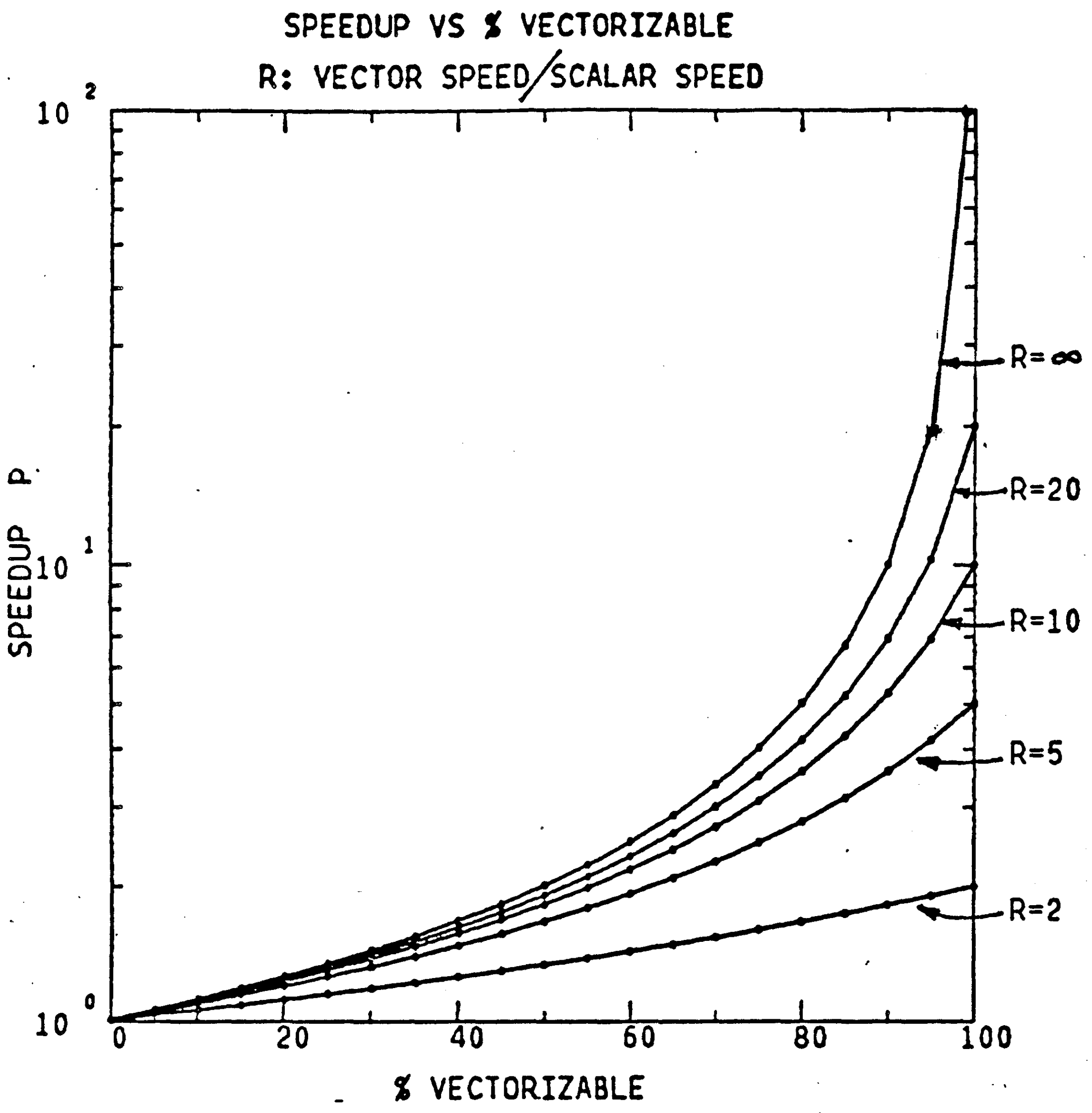


Figure 2: Speedup P as a function of % vectorizable v.

- Elements of a vector in memory can be in consecutive memory locations or can be separated by a constant stride. The stride can be any signed integer.

- *Gather* and *scatter* instructions to facilitate sparse vector operations.

Since the vector registers is of length Z = 128 on the 3090 VF, vector instructions can only process Z elements at a time, and vectors of length greater than Z must be processed by sections, in a process quite sectioning, the timing formula for a vector instruction of length N on 3090 VF is given by

$$T_v = s + r\left[\frac{N}{Z}\right] + Nt \ , \qquad (5)$$

where r is the startup time for processing each section, and s is the vector-sectioning-loop set-up time. The significance of s diminishes when there are more than one vector instructions in the sectioning loop. Note that here and afterwards the square brackets denote the ceiling function.

VS-FORTRAN Version 2 Compiler

Naturally, vector instructions can be incorporated into a program most directly by programming in assembler language. However, the preferred language in science and engineering is FORTRAN. Plus the fact that the vast amount of old FORTRAN programs makes the conversion to assembler impractical. Therefore, vast majority of the application programs must rely on the compiler for vectorization. The VS-FORTRAN Version 2 Compiler is designed to fulfill this need.

The VS FORTRAN Version 2 Compiler employs the state-of-the-art technique in producing highly optimized object code. When scalar compilation is requested, the compiler generates optimized scalar code. When vectorization is requested, the compiler, in addition, analyzes all the nests of DO loops and identifies in them the statements which may be vectorized, applies economic analysis and chooses those loops and statements which will execute fastest on the target vector hardware, and generates optimized vector code for them.

Vector Sectioning Loop

The basic result of vectorizing a DO-loop is to produce vector instructions that operate on groups of data elements. Since the vector instructions on the 3090 Vector Facility operate on Z elements at a time, the compiler converts the scalar loop into a loop over groups (sections) of Z elements. Thus, the loop:

```
    DO 1 I = 1, N
1   A(I) = B(I)
```

is converted by the compiler into:

```
    DO 1 I = 1, N, Z
    DO 1 II = I, MIN(N,I+Z-1)
1   A(II) = B(II)
```

The inner loop (loop with index II) is not actually present. It represents the actions of the vector instruction produced by the compiler.

For more detailed descriptions of the Vectorizing Compiler consult the manual [5] and the report [6].

Good FORTRAN Practice

Since the compiler generates both scalar and vector codes, it is important to follow good scalar practice as well as good vector practice in preparing the application code. Fortunately, experiences show that it is generally true that good vector practice implies good scalar practice and vice versa. It is not unusual that a well converted program intended for a vector machine runs faster even on the scalar portion of the machine. It is equally true that a good scalar program which make efficient use of the high-speed buffer memory (cache) will often result in good vector object code when submitted to a vectorizing compiler.

A notable exception to the general rule is the scalar practice of ***loop unrolling***. See [6] for detail.

The single most effective way to achieve optimal use of storage in vector code is to use stride-1 operations as much as possible. This means, for example, in matrix calculation, using column-oriented algorithm rather than row-oriented algorithm. When writing nested DO-loops for operation on multi-dimensional arrays, try to vary the subscripts in order of leftmost to rightmost.

When rows of a matrix must be accessed and the column dimension is even (especially when it can be divisible by a large power of 2), then it is sometimes advantageous to pad the column so that the column dimension becomes odd. To do this will make more effective use of the interleaved main storage and will also make fuller use of the cache. As a result, the execution will be speeded up at a small cost of storage space.

For more tips on writing good FORTRAN programs intended for both scalar and vector hardware please see [6]. The book by Metcalf [7] also contains many good FORTRAN practices.

Assembler H Version 2

For users who wish to obtain the maximum benefit from the scalar/vector hardware. See the manual [8].

Engineering and Scientific Subroutine Library (ESSL)

ESSL [9] is a set of mathematical routines that exploit the SYSTEM/370 Vector Hardware. It consists of a vector library and a scalar library. The vector library employs the state-of-the-art or new algorithms and fine tuned in assembly language for the vector facility to achieve optimum performance. Where feasible, the effects of paging, cache size, and vector size have all been taken into consideration when deciding on an algorithm. The execution rates of some of the routines in the library approach the theoretical maximum rate of the vector hardware. The scalar library, with its subroutines having the same calling sequences as their counterparts in the vector library, is provided so that one can develop the application code on any scalar facility that subscribes the library. ***ESSL*** subroutines can be called from VS FORTRAN programs as well as from Assembler programs.

The library consists of more than one hundred commonly used subroutines in both short and long precision. These routines fall into six areas of mathematical computation.

1. Linear Algebra

   The linear algebra subprograms consist of vector-scalar subprograms and matrix-vector subprograms. The vector-scalar subprograms contain a group of subprograms which perform the same functions and with the same calling sequences as those of BLAS [10].

2. Matrix Operations

   This group of subroutines provide computations for matrix addition, subtraction, and multiplication in assembly language codes closely tuned for vector hardware.

3. Simultaneous Linear Algebraic Equations

   The simultaneous linear algebraic equations subroutines provide factorizations and solutions to linear systems of equations for a real general matrix, a real banded matrix, a real symmetric positive definite matrix, and a real symmetric positive definite banded matrix.

4. Eigensystems Analysis

   This group of subroutines compute the eigenvalues and either all or selected eigenvectors for a real symmetric matrix and for a real general matrix.

5. Signal Processing

   The signal processing subroutines provide mathematical computations for Fourier transforms, convolutions, and correlations. They also provide programs for four IBM 3838 Array Processor Algorithms for signal processing application.

6. Random Number Generator

Vectorization Technique of Numerical Algorithms

Numerical methods designed for parallel or pipelined architecture started to appear in the literature around mid 1960's. Since the first of these machines became operational in early 1970's, the literature on parallel and vector computing has been increasing at a rapid rate. Besides the two books mentioned earlier, there have been numerous survey papers written on all aspects of parallel/vector computing. In the case of numerical methods, most of the early work were reviewed by Miranker in 1971 [11]. Heller [12] surveyed methods for linear algebra problems in 1978. Similar survey was given by Sameh in 1977 [13]. Recently, (1985) Ortega and Voigt [14] gave a complete account on solution of partial differential equations on parallel and vector computers.

It is appropriate, at this point, to clarify the use of the terms ***parallel*** and ***vector*** in reference to machines and algorithms. Since early 1960's, two contrasting architectural designs have been proposed to exploit the parallelism exhibited in many applications. One design employs array of processors operating simultaneously (or in parallel, hence the name parallel machine) under one central control. An example of such a machine is the Illiac IV. Another design makes use of pipelined arithmetic units and instructions which accepts vector operands (hence the name vector computer) such as the CRAY-1. Both designs fit Flynn's [15] classification as SIMD (single instruction stream, multiple data stream) machines. The vector design has been demonstrated to be more cost effective using the available technology. The methods developed for both types of machines are often interchangeable.

Recently numerous designs which fit Flynn's classification as MIMD (multiple instruction stream, multiple data stream) machines have appeared. These designs range from a few replications of the basic design to thousands of microprocessors operating in unison. Commercially available MIMD machines offer a limited parallelism. Examples are IBM 3084 and 3090 systems. The parallel methods developed for such machines work best with coarse granularity while vector algorithms concentrate on operations in nests of inner loops. In addition, the characteristics of each individual machine, such as the size of available memory, the accessibility of data, the instruction set, and so on, can greatly influence the applicability of a particular algorithm. The challenge for the user is to select or devise an algorithm and arrange the computation so that the architectural features of the target machine are fully utilized. In the rest of this paper, we limit our discussion to vectorizing techniques on the 3090 Vector Facility.

Techniques of vectorization

Many vector algorithms follow the principle known as ***reordering***. ***Reordering*** may involve restructuring the computational domain such as re-numbering the nodes in a grid so that more parallelism is revealed in the computation; or it may simply mean the rearranging the order of computation in order to increase the vector length or to decrease the stride or both.

An example of using the ***reordering*** principle is the implementation of the 3090 VF instruction ***multiply and accumulate*** where products of corresponding elements of two vectors are accumulated into four partial sums in the first four locations of a vector register. The instruction ***sum partial sums*** is next executed to obtain the total sum. The reason for arranging the computation in this manner is to allow the multiply and add pipelines working in tandem so as to produce one multiply-and-add result each clock period. We shall give additional examples of using the ***reordering*** principle below.

The above principle is applied to increase the percentage of a computation that can be vectorized. In order to execute the vector instructions at full speed, source vectors must be delivered to the arithmetic pipeline without any delay. This will happen if the data needed are already in the cache memory. Since a single vector instruction may specify large amount of computation, the requirement on data rate is more severe in vector processing than in scalar processing. One should exercise extra care in structuring the data so that cache memory and main memory are optimally utilized. This can be achieved by choosing stride-1 or low stride algorithms. Fortunately, many algorithms in scientific computation involve operations on vectors whose elements reside in consecutive memory locations.

To reduce the memory traffic, intermediate results of computation should be kept in the registers as long as possible. Outer-loop unrolling [16] is a programming technique to achieve this end.

Vectorization in Linear Algebra

Computational linear algebra is the most studied topic in numerical analysis because the majority of scientific computation can be formulated as basic matrix and vector operations. Large matrix problems are now routinely solved owing mainly to the existence of reliable software such as LINPACK [17] and EISPACK [18]. There is also a great wealth of literature on matrix calculations on vector computers. The reader is referred to the survey papers mentioned above for more information. A recent paper by Dongarra, Gustavson, and Karp [19] is particularly informative. They discuss how the performance can vary by simply reordering the loops in the program and suggest best algorithms for register oriented vector machines. The report [6] also includes a section on implementing well-known algorithms on the 3090 VF. We do not intend to give a complete review of previous works here. Instead, we shall illustrate the use of the principle of ***reordering*** and the method of evaluating competing algorithms by implementing the computation of

- Two basic vector operations
- Matrix-vector multiply
- Convolution
- Tridiagonal system solution
- Iterative solution of large sparse linear systems.

Two basic vector operations

1. Dot product, known as **SDOT** in **BLAS**

   For two vectors x and y each with n elements, the dot product is given by:

$$\sum_{i=0}^{n} x_i y_i \tag{6}$$

2. Scalar times vector plus vector, known as **SAXPY** in **BLAS**

   For two vectors x and y and a scalar α, this calculation is given by:

$$y + \alpha x \tag{7}$$

Both expressions can be computed efficiently on 3090 VF.

Dot Product

Dot product can be computed with the sectioning loop shown below:

```
*   assume vector length n in GR0
*   address of x in GR1
*   address of y in GR2
*
      VZPSD  VR0             zero partial sums
LOOP  VLVCU  GR0             load VCT and update
      VLD    VR2,GR1         load a section of x
      VMCD   VR0,VR2,GR2     multiply section of y to x
*                            and accumulate partial sums
      BC     2,LOOP          branch to LOOP if GR0>0
      SDR    FR0,FR0         clear FR0
      VSPSD  VR0,FR0         sum partial sums (dot pdt)
```

Allowing 50 cycles each for VZPSD, VLVCU, and VSPSD, and assuming 30 cycles of startup time for each section for VLD and VMCD and 5 cycles for each scalar instruction, we can estimate the time (in cycles) for the dot product to be:

$$t_{\text{dot}} = 2n + 115\left\lceil \frac{n}{128} \right\rceil + 105 \tag{8}$$

Dot product timing

Using the above formula, we can estimate the running time for some typical values of n.

n	128	256	512	1024
t_{dot}	476	847	1589	3073
Mflop	29	33	35	36

This compares favorably with the best scalar dot product time of 10 MFLOPS on the 3090.

Scalar times a vector plus vector: SAXPY

SAXPY can be computed with the following code:

```
*   compute y=y+ αx
*   assume GR0 has vector length n
*   address of x in GR1
*   address of y in GR2
*   α is in FR0
*
      LR     GR3,GR2        copy address y in GR3
LOOP  VLVCU  GR0            load VCT and update
      VLD    VR0,GR2        load a section of y
      VMADS  VR0,FR0,GR1    multiply a section of x
*                           by α and add to the
*                           corresponding y section
      VSTD   VR0,GR3        store a section in y
      BC     2,LOOP         branch back if GR0>0
```

Assuming the same start-up time (30 cycles) for each section of VMADS and VSTD as before, we can estimate the **SAXPY** time as:

$$t_{saxpy} = 3n + 145\left\lceil \frac{n}{128} \right\rceil \tag{9}$$

Using this formula, we obtain the following timing:

n	100	200	500	1000
t_{saxpy}	445	890	2080	4160
Mflop	24	24	26	26

Note that the MFLOP rates are not as high as those of the dot product. This is because of the I/O requirement of the SAXPY is **3n** words versus only **2n** for dot product.

In general, we can estimate the speed of a computation task by looking at its ***computation density*** which can be defined as the following ratio:

$$\frac{\text{total operations performed}}{\text{total data items required in and out of memory}} \tag{10}$$

In general, the higher the ***computation density***, the faster the processing rate. The ***computation densities*** of **dot product** and **SAXPY** are *1* and *2/3* respectively.

In next two sections, we shall look at more complex computations in which multiple SAXPY's are required and computation can be arranged so that intermediate results can be held in registers. Thus, store operations can be mostly eliminated resulted in much higher ***computation density*** than the single SAXPY.

Matrix-vector multiply

Consider the multiplication of a vector x by a matrix A:

$$y = Ax\,, \tag{11}$$

where A is a real matrix of dimension mxn, and x and y are real column vectors of dimension n and m respectively. Assuming that A is stored in column-major order, we compare the following three methods for computing y.

Horizontal Processing

In ***horizontal processing***, vector computations are performed horizontally from left to right in row fashion.

Assuming that y has been set to zero initially, the FORTRAN code represents the computation of y using ***horizontal processing***:

```
      DO 10 I = 1,M
      DO 10 J = 1,N
   10 Y(I) = Y(I) + A(I,J) * X(J)
```

The inner loop represents the **dot product** of i-th row of A and x. Since A is columnwise stored, the stride of the row vector is m.

An equivalent assembly code is:

```
*  GR0 = n, GR1 = GR7 = m
*  GR5 = GR6 = 1, GR11 = 8
*  add. A in GR2, x in GR3, y in GR4
```

```
*
LP2  LR     GR8,GR3          GR8 has address of x
     SDR    FR0,FR0          clear FR0
     VZPSD  VR0              clear VR0 for partial sum
     LR     GR10,GR2         GR10 get address of A
     LR     GR9,GR0          GR9 = n
LP1  VLVCU  GR9              load VCT and update
     VLD    VR2,GR2(GR1)     load a row section of A
     VMCD   VR0,VR2,GR8      multiply by x section
*                            and partial sum to VR0
     BC     2,LP1            branch back if GR0>0
     VSPSD  VR0,FR0          sum partial sums in FR0
     STD    FR0,0(GR4)       store dot product in y
     AR     GR4,GR11         update GR4 for next y
     AR     GR2,GR11         update GR2 for next row
     BXLE   GR5,GR6,LP2      branch
```

Using the same assumptions for each type of instructions as before, we get the timing formula for ***horizontal processing***:

$$t_h = m\left(2n + 115\left\lceil\frac{n}{128}\right\rceil + 140\right) \quad (12)$$

Note that this formula is valid only for infinite cache. The degradation for large matrices that cannot be held in the cache is severe due to large stride. The code requires m passes over the address space of the matrix.

Vertical Processing

In ***vertical processing***, vector computations are carried out vertically from top to bottom in column fashion.

If we simply switch the DO loops in the ***horizontal processing*** code, we obtain the code for ***vertical processing***. Thus:

```
      DO 10 J = 1,N
      DO 10 I = 1,M
  10  Y(I) = Y(I) + A(I,J) * X(J)
```

This is a stride-1 code. It requires only one pass over the address space of A. The inner loop represents a SAXPY which requires loading and storing once for each segment of y. We can see this clearly in the following assembly code.

```
*  assume y has been set to zero
*  GR0=GR7=n, GR1=m
*  GR5=GR6=1
*  GR10=8
*  GR2=address(A)
*  GR3=address(x)
*  GR4=address(y)
*
LP2  LD     FR0,0(GR3)     load x element in FR0
     LR     GR9,GR4        GR9 gets address of y
     LR     GR11,GR4       GR11 gets address of y
     LR     GR8,GR1        GR8=m
LP1  VLVCU  GR8            load VCT and update
     VLD    VR0,GR4        load a section of y
     VMADS  VR0,FR0,GR2    VR0=VR0+section(A)* x
     VSTD   VR0,GR11       store y section
     BC     2,LP1          branch back if GR8>0
     AR     GR3,GR10       GR3 points to next x elem.
     BXLE   GR5,GR6,LP2    branch
```

The timing formula for the above code is:

$$t_v = n\left(3m + 145\left[\frac{m}{128}\right] + 30\right) \qquad (13)$$

This formula gives good approximation of the computation time even for large matrices since the penalty for cache traffic is small for stride-1 operations. The shortcoming of this code is the need to load and store y vector $2n$ times.

Block processing

In ***block processing***, sectioning vector loop computations are performed from left to right and top to bottom in a combined horizontal and vertical approach.

A FORTRAN code represents the ***block processing*** is as follows:

```
      DO 10 I = 1 , M , Z
      DO 10 J = 1 , N
      DO 10 II = I , MIN ( M , I+Z-1 )
C
C   Z is the section size and = 128 on 3090VF
C
 10   Y(II) = Y(II) + A(II,J) * X(J)
```

By sectioning the outer loop, the y vector can be computed one section at a time (represented by the inner two loops). The results of the II loop can be held in a vector register until the final results are obtained (i.e., until the J loop has been exhausted). The code involves only stride-1 operations, although it does require $[m/128]$ passes of the address space of the matrix.

The following code employs the ***block processing*** approach.

```
*   GR0=GR11=n, GR1=m, GR7=8m, GR10=1
*   GR2=add(A), GR3=add(x), GR4=add(y)
*
LP2  LA     GR9,1           GR9=1
     LR     GR8,GR3         GR8=address of x
     VLVCU  GR1             load VCT and update
     VLZDR  VR0             clear VR0
     LR     GR5,GR2         GR5 gets address of A
LP1  LD     FR0,0(GR8)      load x element
     LR     GR6,GR5         GR6=current address of A
     VMADS  VR0,FR0,GR6     VR0=VR0+col. sec. of A*x
     AR     GR5,GR7         GR5 points to next col. of A
     LA     GR8,8(GR8)      update GR8 to next x element
     BXLE   GR9,GR10,LP1    branch back if not done
     VSTD   VR0,GR4         store a section in y
     LA     GR2,1024(GR2)   GR2 points to next block
     BC     2,LP2           branch back if GR1>0
```

Note that no vector load instructions are needed anywhere in the code.

The timing formula is given by:

$$t_b = nm + 2m + (55n + 135)\left\lceil \frac{m}{128} \right\rceil \qquad (14)$$

Again this formula gives a good estimate of the running time even for large matrices because only stride-1 vector operations are involved.

Comparison of horizontal, vertical, and block processing

Using the timing formulas, we obtain the following table:

n,m	**50**	**200**	**500**
t_h	17750	-----	-----
Mflops	15		
t_v	16250	184000	1055000
Mflops	16	23	26
t_b	5485	62670	361540
Mflops	49	69	75

We can reject outright the ***horizontal*** code because it causes inefficient use of the cache storage. The ***block*** code outperforms the ***vertical*** code due to the more efficient use of the vector register. Their ***computation densities*** (being 2 and 2/3 respectively) also show the right choice is ***block*** processing.

More choices of methods are available for **matrix multiply** and **solution of linear systems** by simply ***reordering*** the loops. They are discussed in [19].

Convolution

Given two sequences $u = (u_1, u_2, ..., u_n)$ and $x = (x_1, x_2, ..., x_{m+n-1})$. The convolution of x and u, $(x * u)$, is a sequence of length m given by: (for $i = 1, ..., m$)

$$(x * u)_i = \sum_{j=1}^{n} x_{n+i-j} u_j. \tag{15}$$

Convolution is actually a special matrix-vector multiply. For example, let n = 4 and m = 5, the above equation in expanded form is:

$$(x * u)_1 = x_4 u_1 + x_3 u_2 + x_2 u_3 + x_1 u_4 \tag{16}$$

$$(x * u)_2 = x_5 u_1 + x_4 u_2 + x_3 u_3 + x_2 u_4 \tag{17}$$

$$(x * u)_3 = x_6 u_1 + x_5 u_2 + x_4 u_3 + x_3 u_4 \tag{18}$$

$$(x * u)_4 = x_7 u_1 + x_6 u_2 + x_5 u_3 + x_4 u_4 \tag{19}$$

$$(x * u)_5 = x_8 u_1 + x_7 u_2 + x_6 u_3 + x_5 u_4 \tag{20}$$

Many algorithms (notably those based on FFT) have been developed to compute convolution at reduced number of operations. When the length of the filter, u, is small (say 50) compared to the length of the time series, x, then it is more economical to compute the convolution by straightforward matrix-vector multiply. However, convolution differs from general matrix-vector multiply in two ways:

1. no need to store a matrix
2. no large stride problem.

As a consequence, ***horizontal processing*** is faster than ***vertical processing*** in computing convolution. The best method, nonetheless, is still ***block processing***. In an actual test on the 3090VF [20], a speedup of more than 10 over the scalar execution was obtained using the ***block*** approach.

Tridiagonal Solver

The solution of tridiagonal system of linear equations are required when implicit schemes are used to solve differential equations by finite difference methods. The usual method for solving such a system on a serial computer is based on Gaussian elimination which is entirely recursive and does not lend itself conveniently to vectorization. In the past dozen years, a number of new algorithms have been proposed for solving tridiagonal systems on parallel and vector computers. Probably the best vector algorithm is based on the cyclic reduction method [21] first developed by Gene Golub and Roger Hockney and reported in Hockney [22]. For a system of n equations, the average vector length of cyclic reduction is n / p, where $p = \log_2 n$. This vectorization is achieved at a cost of approximately twice the arithmetic operations (as compared with Gaussian elimination). In what follows, we estimate the speedup of cyclic reduction over Gaussian elimination for solving one system of tridiagonal equations.

Gaussian Elimination

Consider the tridiagonal system of n linear equations: $Ax = r$, where A is a tridiagonal matrix with its i-th row denoted by $(..., c_i a_i b_i ...)$. The solution can be obtained in 2 steps:

1. Forward step.

$$d_1 = 1 / a_1 \tag{21}$$

$$d_i = 1 / (a_i - c_i d_{i-1} b_{i-1}), i = 2, ..., n. \tag{22}$$

$$y_1 = r_1 \tag{23}$$

$$y_i = r_i - c_i d_{i-1} y_{i-1}, i = 2, ..., n. \tag{24}$$

2. Backward step.

$$x_n = y_n d_n \tag{25}$$

$$x_i = (y_i - x_{i+1} b_i) d_i, i = n - 1, ..., 1. \tag{26}$$

In actual coding, array d can replace the main diagonal a, while y, and x can share the same space as r. A total of $9n$ arithmetic operations (including n divides) are required to obtain the answers.

FORTRAN code for Gaussian elimination

The following code computes the solution of one tridiagonal system.

```
      SUBROUTINE GAUSS3(A,B,C,R,N)
C
C  A is main diagonal, B is super-diagonal.
C  C is sub-diagonal.
C  R is the right side and result on exit.
C
      DIMENSION A(N),B(N),C(N),R(N)
C
C  forward step
C
      A(1)=1.0/A(1)
      DO 10 I =2,N
      A(I)=1.0/(A(I)-C(I)*A(I-1)*B(I-1))
   10 R(I)=R(I)-C(I)*A(I-1)*R(I-1)
C
C  backward step
C
      R(N)=R(N)*A(N)
      DO 20 I =N-1,1,-1
   20 R(I)=(R(I)-R(I+1)*B(I))*A(I)
      RETURN
      END
```

Note that a good compiler should only evaluate the expression: C(I)*A(I-1) once.

Cyclic Reduction

The approach in cyclic reduction is to eliminate in the even-numbered equations the coefficients associated with the odd-numbered variables by elementary row transformations. These transformations can be carried out simultaneously. The transformed even-numbered equations again form a tridiagonal system half of the original size. The elimination can be repeated on the reduced system. After p steps of such elimination, there is only one equation left. The solution of that equation is readily obtained. Other solutions can then be obtained by back substitution. The implementation of the algorithm as given in [23] requires a total of $20n$ arithmetic operations (including n divides).

As a vector algorithm to solve tridiagonal equations, **cyclic reduction** as described above has two weaknesses, namely:

1. The vector length changes by a factor of 2 each step. This causes more overhead.
2. The indices of the coefficients appearing in successive tridiagonal system become further separated at each step. This complicates the addressing scheme and results in less than optimal use of the cache storage.

A recent implementation by Kershaw [24] alleviates the second weakness mentioned above. In addition, the operation count of his implementation is reduced to $18n$ (with n divides) at the expense of some extra storage.

Speedup

In [25], a measurement of the 3090 vector time of the ***cyclic reduction*** was made based on an implementation of Kershaw's algorithm. Similar technique was used to measure the scalar execution time of the ***Gaussian elimination*** solution. The diagram of speedup versus the order of the matrix, n is reproduced here as Figure 3. [1]

Note that the vector execution is slower than the scalar execution for $n < 250$. Even for large n, the speedup is insignificant although the ***cyclic reduction*** code is almost entirely vectorizable. This is largely due to:

- The excellent scalar speed of the 3090 makes its vector to scalar speed ratio relatively moderate.
- ***Cyclic reduction*** requires twice the amount of arithmetic as compared with ***Gaussian elimination.***

However, quite often solutions of multiple independent tridiagonal systems are required (example: ***Line SOR*** method discussed in the next section) when finite difference methods are used. In these cases, the scalar ***Gaussian elimination*** algorithm can be applied simultaneously to all (or two halves in the case of ***line SOR***) the tridiagonal systems and the execution can be completely vectorized.

Iterative solution of large sparse linear systems

Large but sparse linear systems of equations arise when finite difference methods are used to solve partial differential equations. We choose the following model problem for discussion.

The Model Problem

The most studied elliptic boundary value problem is the Dirichlet problem, which is to find the equilibrium temperature distribution in the interior of a homogeneous solid when specified temperature is prescribed on its boundary. By physical principle, the temperature satisfies the ***Laplace equation.*** In two space dimensions, the ***Laplace equation*** takes the form

$$\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = 0 \qquad (27)$$

1 The vector speedups of the tridiagonal solvers included in the new release 2 ESSL (May, 1987) are noticeably better then those shown in Figure 3.

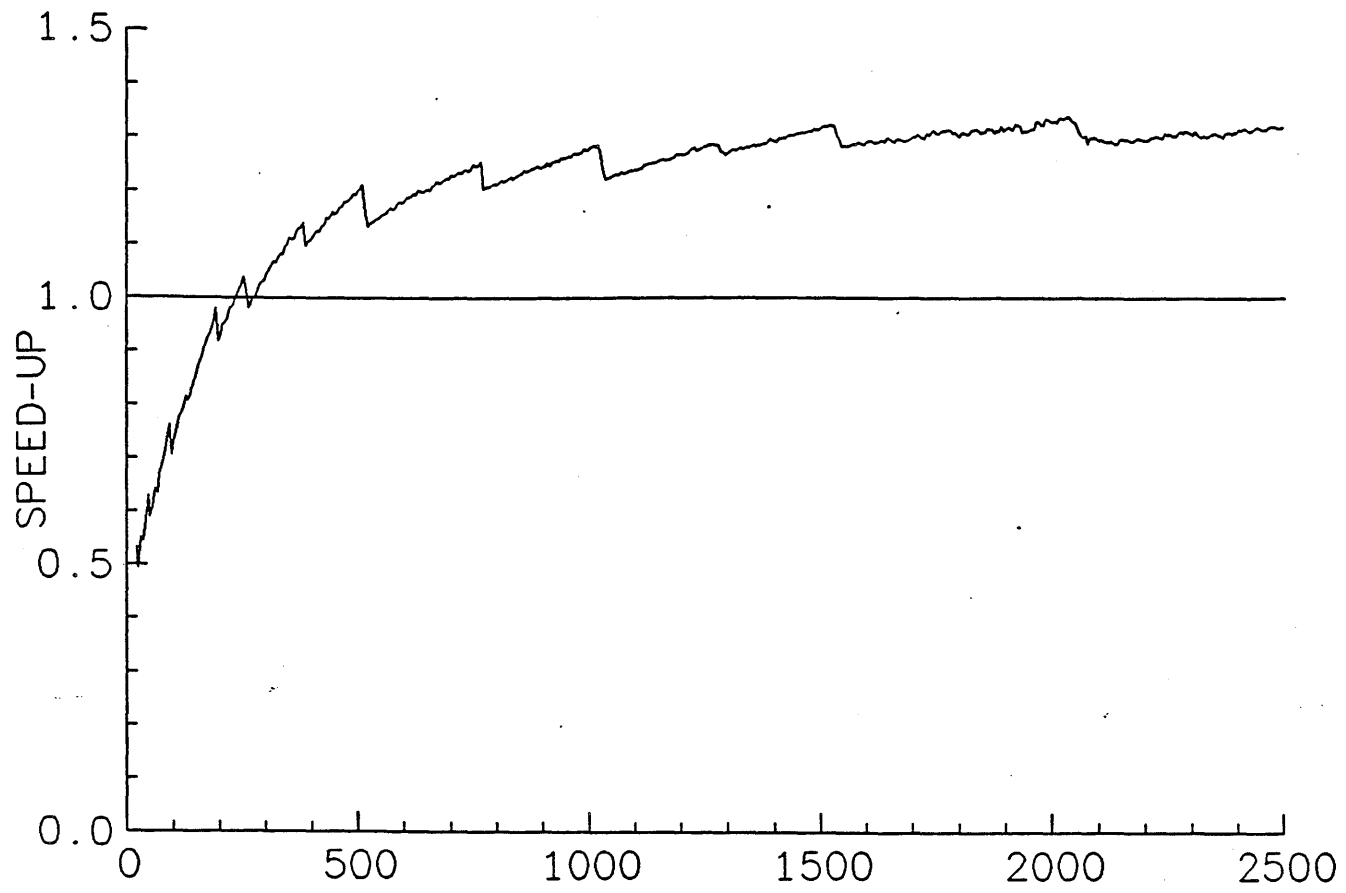


Figure 3: Estimated speedup of vector vs scalar algorithm for tridagonal system solvers as a function of *n*.

The model problem refers to the solution of the Dirichlet problem in a unit square region, $R: 0 < x < 1, 0 < y < 1$. That is: to find $u(x,y)$ which satisfies equation (27) in R and the boundary value, $g(x,y)$, specified on four sides.

In the numerical solution we impose a uniform grid $R(h)$ on R, where $R(h)$ is made of mesh lines:

$$x_i = ih, y_i = ih, i = 0, 1, ..., n + 1, \tag{28}$$

where $h = (n + 1)^{-1}$. For example, for $n = 4$, the grid is as shown in Figure 4. The intersections of the grid lines are called grid points. Those in the interior are called interior grid points (solid dots) and those on the boundary are termed boundary points. The standard five-point difference equation which approximates equation (27) is:

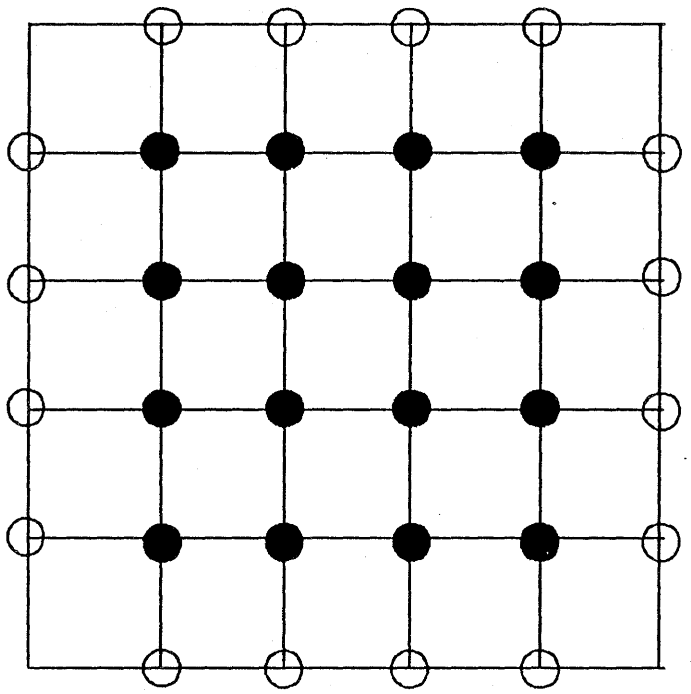


Figure 4: A grid with 4x4 interior grid points.

$$4u(ih,jh) - u((i-1)h,jh) - u((i+1)h,jh) - u(ih,(j-1)h) - u(ih,(j+1)h) = 0\ ,\ i,j = 1,\ldots n. \qquad (29)$$

The above formula is valid for all interior mesh points. Expressing in words, the formula simply means that the temperature at a point is the average temperature of its four nearest neighbors.

For instance, if we apply the above difference equation to the grid in Figure 4 in which the interior grid point are numbered in rowwise and left to right fashion (***natural ordering***) as shown in Figure 5, we obtain, for grid-point 1, the equation:

$$4u_1 - u_2 - u_5 = u(.2,0) + u(0,.2) = g(.2,0) + g(0,.2) \equiv r_1 \qquad (30)$$

and for grid-point 2:

$$-u_1 + 4u_2 - u_3 - u_6 = u(.4,0) = g(.4,0) \equiv r_2 \qquad (31)$$

and so on. Note that the right hand sides of both equations consist of known boundary values and we have used new symbols, r_1 and r_2, to represent them.

We can express all the equations in a single matrix equation:

$$Au = r, \qquad (32)$$

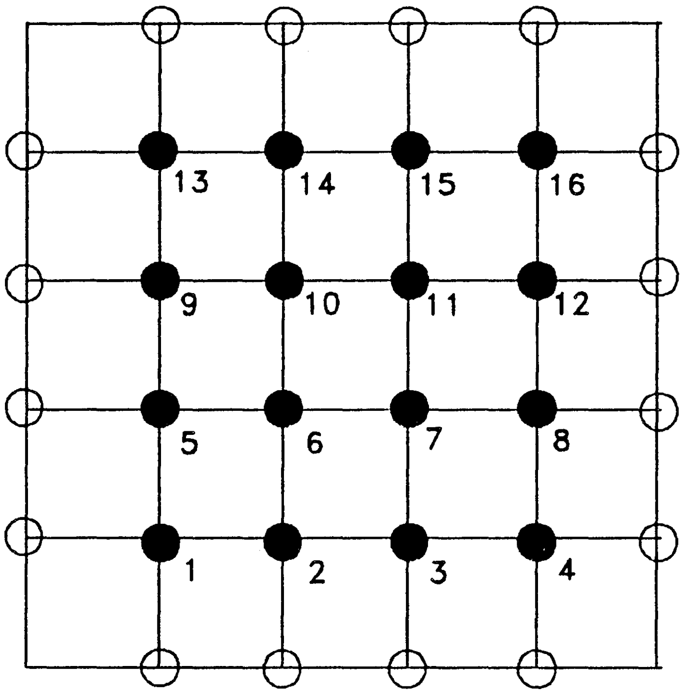


Figure 5: Natural ordering of interior grid points.

where A is the coefficient matrix, and u is the column vector of unknowns, and r is the column vector of known boundary values. In our 4x4 grid example, the matrix A takes the form as shown in Figure 6. Note that the matrix is sparse in view of the fact that the maximum number of nonzero element in a row or column is five, also note that the matrix is regularly structured in that all the nonzero elements lie on five diagonals. Experience shows that such a system can be very effectively solved by an iterative method.

Most iterative methods can be described as follows: Let M be a nonsingular matrix but otherwise arbitrary, then $A = M - B$ is a ***splitting*** of A and, associated with this ***splitting***, there is an iterative scheme for solving (32)

$$Mu^{(i+1)} = Bu^{(i)} + r, \tag{33}$$

where the superscript indicates the iteration number. The basic operations of numerical methods for solving (32) are matrix-vector multiplication and solution of linear system involving sparse but regularly structured matrix. For good vectorization, we like to choose an iterative method such that:

1. The Matrices M and B in (33) retain the sparse and diagonal form of A.

2. The matrix M is easily factored into triangular factors which are as sparse as A and possess partitions with diagonal blocks $(D_1, D_2, ..., D_m)$, where D_i are diagonal and m is small (preferably 1 or 2 or at least much less than the order of matrix).

$$\begin{bmatrix}
4 & -1 & & & -1 & & & & & & & & & & & \\
-1 & 4 & -1 & & & -1 & & & & & & & & & & \\
& -1 & 4 & -1 & & & -1 & & & & & & & & & \\
& & -1 & 4 & & & & -1 & & & & & & & & \\
-1 & & & & 4 & -1 & & & -1 & & & & & & & \\
& -1 & & & -1 & 4 & -1 & & & -1 & & & & & & \\
& & -1 & & & -1 & 4 & -1 & & & -1 & & & & & \\
& & & -1 & & & -1 & 4 & & & & -1 & & & & \\
& & & & -1 & & & & 4 & -1 & & & -1 & & & \\
& & & & & -1 & & & -1 & 4 & -1 & & & -1 & & \\
& & & & & & -1 & & & -1 & 4 & -1 & & & -1 & \\
& & & & & & & -1 & & & -1 & 4 & & & & -1 \\
& & & & & & & & -1 & & & & 4 & -1 & & \\
& & & & & & & & & -1 & & & -1 & 4 & -1 & \\
& & & & & & & & & & -1 & & & -1 & 4 & -1 \\
& & & & & & & & & & & -1 & & & -1 & 4
\end{bmatrix}$$

Figure 6: Matrix A corresponding to the natural ordering.

Condition 1 is important because matrix-vector multiply involving M and B can be computed with a few ***vector multiply and add*** instructions if they are stored by the diagonals. Condition 2 is essential for fast vector solution of linear systems involving M.

Basic Iterative Methods

It is convenient to express the matrix A as

$$A = L + D + U\,, \tag{34}$$

where L is the strict lower triangular part of A, D is the main diagonal of A, and U is the strict upper triangular part of A. In studying an iterative method, not only one should know how easy it can be computed, but also one must ask how fast does it converge?

1. The Jacobi Method

Perhaps the simplest iterative scheme is the Jacobi iteration in which $M = D$ in (33). That is

$$u^{(i+1)} = D^{-1}\left[r - (L + U)u^{(i)}\right] \tag{35}$$

It prescribes that the value of the new iteration at a point is the average value of the previous iteration at its four nearest neighbors. It is quite obvious that all components of the right side vector of (35) can be computed simultaneously and the method is completely vectorizable with long vectors (length = n^2) with the ***natural ordering*** of the grid points. However, this method is seldom used because of its slow convergence rate.

2. The Gauss-Seidel Method

It is intuitively clear that if we make use of the new values as soon as they have been computed in the iteration (33), the convergence will be faster. Indeed, we can double the rate of convergence if we use the Gauss-Seidel method where such strategy is employed by letting $M = L + D$ and the iteration can be written

$$(L + D)u^{(i+1)} = r - Uu^{(i)} . \tag{36}$$

Equation (36) represents a lower triangular system of linear equations. It is readily apparent that the Gauss-Seidel iteration using the ***natural ordering*** is not vectorizable since condition 2 discussed above is not met. Fortunately, it has been shown (see Young [26], for instance) that convergence will not suffer if we reorder the grid point in checkerboard fashion, or the so-called ***red-black*** ordering as shown in Figure 7. The matrix A corresponding to this ordering is shown in Figure 8 . Equation (36) can naturally be partitioned into the block form

$$\begin{bmatrix} D_1 & 0 \\ L & D_2 \end{bmatrix} \begin{bmatrix} u_1^{(i+1)} \\ u_2^{(i+1)} \end{bmatrix} = \begin{bmatrix} r_1 \\ r_2 \end{bmatrix} - \begin{bmatrix} 0 & U \\ 0 & 0 \end{bmatrix} \begin{bmatrix} u_1^{(i)} \\ u_2^{(i)} \end{bmatrix} \tag{37}$$

The above equation can be computed in two steps: i.e.,

$$u_1^{(i+1)} = D_1^{-1}\left[r_1 - Uu_2^{(i)}\right] , \tag{38}$$

and

$$u_2^{(i+1)} = D_2^{-1}\left[r_2 - Lu_1^{(i+1)}\right] . \tag{39}$$

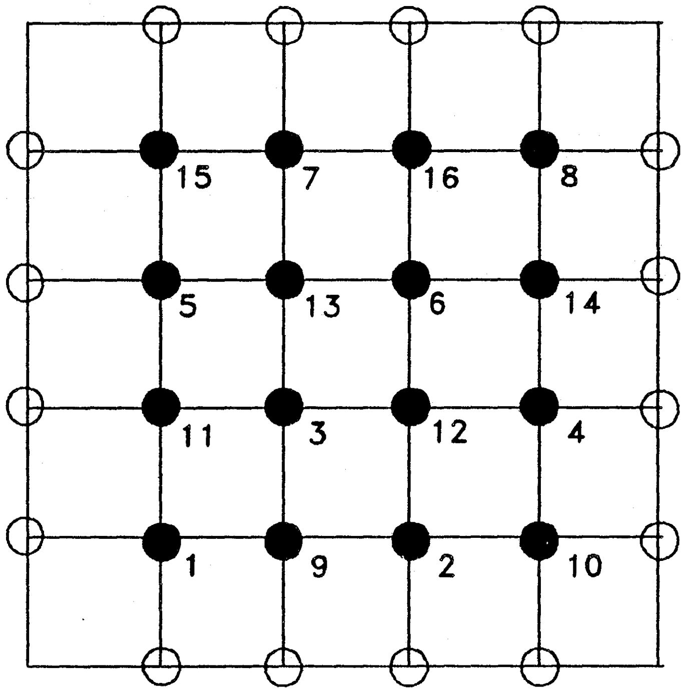


Figure 7: Red-black ordering of interior grid points.

These two equations look a lot like the Jacobi iteration and each involves vector operations with vector length $n^2/2$. We want to point out here that non-zero elements of L and U lie on five diagonals in our 4x4 example. In the case when the number of grid points on each line is odd, the non-zero elements would lie only on four diagonals. Therefore, one must be careful in computing the matrix-vector products involving L and U.

We have just demonstrated that by ***reordering*** the grid point in checkerboard fashion, we transform the serial Gauss-Seidel method into a very satisfying vector method.

3. The Successive Over-relaxation (SOR) Method

The convergence rate of the Gauss-Seidel method can be increased by an order of magnitude if the SOR method is used. The SOR method can be regarded as an extra extrapolation process from the Gauss-Seidel iterates. That is

$$u_{sor}^{(i+1)} = (1-\omega)u_{sor}^{(i)} + \omega u_{gs}^{(i+1)} \ , \tag{40}$$

where ω is the relaxation factor. The best value for ω is between 1 and 2. If $\omega = 1$ the method reduces to that of Gauss-Seidel. The SOR has been the method of choice for many applications involving sparse linear systems ever since Young's dissertation [27] was published in the early 1950's. The ***red-black*** ordered SOR method has also become the standard method for vector computation.

$$
\left[\begin{array}{cccccccc|cccccccc}
4 & & & & & & & & -1 & & -1 & & & & & \\
& 4 & & & & & & & -1 & -1 & & -1 & & & & \\
& & 4 & & & & & & -1 & & -1 & -1 & -1 & & & \\
& & & 4 & & & & & & -1 & & -1 & & -1 & & \\
& & & & 4 & & & & & & -1 & & -1 & & -1 & \\
& & & & & 4 & & & & & & -1 & -1 & -1 & & -1 \\
& & & & & & 4 & & & & & & -1 & & -1 & -1 \\
& & & & & & & 4 & & & & & & -1 & & -1 \\
\hline
-1 & -1 & -1 & & & & & & 4 & & & & & & & \\
& -1 & & -1 & & & & & & 4 & & & & & & \\
-1 & & -1 & & -1 & & & & & & 4 & & & & & \\
& -1 & -1 & -1 & & -1 & & & & & & 4 & & & & \\
& & -1 & & -1 & -1 & -1 & & & & & & 4 & & & \\
& & & -1 & & -1 & & -1 & & & & & & 4 & & \\
& & & & -1 & & -1 & & & & & & & & 4 & \\
& & & & & -1 & -1 & -1 & & & & & & & & 4
\end{array}\right]
$$

Figure 8: Matrix corresponding to red-black ordering.

Block Iterative Methods

The convergence rate of a basic iteration can be increased if we use the corresponding block iteration in which a group (***block***) of grid points are updated simultaneously. In the case of the popular ***line-SOR*** method where a line of grid values are computed via the solution of a tridiagonal system. Thanks to the work of Cuthill and Varga [28], line iteration does not incur additional computation cost over the basic point iteration. As in the case of point method, the line-SOR using the ***natural ordering*** (i.e., line by line order) is not vectorizable. However, it is obvious that all the odd (even) lines can be solved simultaneously. Therefore, the line method using the ***red-black*** or the ***zebra*** ordering is vectorizable with vector length $n/2$ (solving $n/2$ tridiagonal systems simultaneously). For large grid problem, this is the choice method for many applications.

Other Iterative Methods

In recent years, many iterative methods have been devised to compute the solution of large and sparse systems, most notable among them are a group of methods called the ***pre-conditioned conjugate gradient (PCG)*** methods. For a review of their effectiveness on modern vector computers, the reader is referred to the survey [14] and the references contained therein.

Summary

We have discussed three methods of **vectorization**:

1. Use the vectorizing compiler: **VS FORTRAN Version 2 Compiler**.
2. Restructure the application code before compile.
3. Write a new code based on a new vector algorithm.

We have given, via examples in linear algebra computation, some good vector practices:

- Choose stride-1 or low stride algorithms.
- Use ***block processing*** whenever feasible to optimize the vector register usage.
- ***Reorder*** the DO-loops often lead to good vectorization.
- Make calls to vector libraries such as the *ESSL*.

Bibliography

[1] R. W. Hockney and C. R. Jesshope, *Parallel Computers: Architecture, Programming and Algorithms,* Adam Hilger Ltd (Bristol, England, 1981).

[2] K. Hwang and F. A. Briggs, *Computer Architecture and Parallel Processing,* McGraw-Hill (1984).

[3] G. Paul and M. W. Wilson, *VECTRAN Language: An Experimental Language for Vector-Matrix Array Processing,* IBM Palo Alto Scientific Center Report G320-3334 (1975).

[4] G. M. Amdahl, Validity of the single processor approach to achieving large scale computing capabilities, *AFIPS Conference Proceedings* **30** (1967) pp. 483-485.

[5] VS-FORTRAN Version 2 Compiler, Library and Interactive Debug, General Information, GC26-4219, IBM Corporation.

[6] A. A. Dubrulle, R. G. Scarborough and H. G. Holsky, *How to Write Good Vectorizable FORTRAN,* IBM Palo Alto Scientific Center Report G320-3478 (1985).

[7] M. Metcalf, *FORTRAN Optimization,* Academic Press (1982).

[8] Assembler II Version 2 Application Programming Guide, SC26-4036, IBM Corporation.

[9] Engineering and Scientific Subroutine Library, General Information, GC23-0182, IBM Corporation.

[10] C. L. Lawson, R. J. Hanson, D. R. Kincaid, and F. T. Krogh, Basic Linear Algebra Subprograms for FORTRAN Usage, *ACM Transactions on Mathematical Software* **5** (1979) pp. 308-323.

[11] W. Miranker, A Survey of Parallelism in Numerical Analysis, *SIAM Review* **13** (1971) pp. 524-547.

[12] D. Heller, A Survey of Parallel Algorithms in Numerical Linear Algebra, *SIAM Review* **20** (1978) pp. 740-777.

[13] A. Sameh, Numerical Parallel Algorithms, A Survey in D.Kuck, D.Lawrie, and A.Sameh (editors), *High Speed Computer and Algorithm Organization, Academic Press* (1977) pp. 207-228.

[14] J. M. Ortega and R. G. Voigt, Solution of Partial Differential Equations on Vector and Parallel Computers, *SIAM Review* **27** (1985) pp. 149-240.

[15] M. J. Flynn, Very High-Speed Computing Systems, *Proc. IEEE* **54** (1966) pp. 1901-1909.

[16] J. J. Dongarra and S. C. Eisenstat, Squeezing the Most out of an Algorithm in CRAY FORTRAN, *ACM Transactions on Mathematical Software* **10** (1984) pp. 219-230.

[17] J. J. Dongarra, C. B. Moler, J. R. Bunch, and G. W. Stewart, *LINPACK User's Guide,* SIAM, Philadelphia (1979).

[18] B.T.Smith, J.M.Boyle, J.J.Dongarra, B.S.Garbow, Y.Ikebe, V.C.Klema and C.B.Moler, *Matrix Eigensystem Routines-EISPACK Guide, second edition,* Springer-Verlag (1976).

[19] J.J.Dongarra, F.G.Gustavson, and A.H.Karp, Implementing Linear Algebra Algorithms for Dense Matrices on a Vector Pipeline Machine, *SIAM Review* **26** (1984) pp. 91-112.

[20] G. Radicati, *privite communication,* IBM Scientific Center, Rome, Italy.

[21] H. S. Stone, Parallel Tridiagonal Equation Solver, *ACM Transcations on Mathematical Software* **1** (1975) pp. 308-329.

[22] R. W. Hockney, A fast direct method of Poisson's equation using Fourier analysis, *Journal of the Association for Computing Machinery* **12** (1965) pp. 95-113.

[23] J. J. Lambiotte and R. G. Voigt, The solution of tridiagonal linear systems on the CDC Star-100 computer, *ACM Transcations on Mathematical Software* **1** (1975) pp. 308-329.

[24] D. Kershaw, Solution of single tridiagonal linear systems and Vectorization of the ICCG Algorithm on the CRAY-1, in G. Rodrigue (ed), *Parallel Computations, Academic Press* (1982) pp. 85-100.

[25] J. Gazdag, G. Radicati, P. Sguazzero and H. H. Wang, Seismic Migration on the IBM 3090 Vector Facility, *IBM Journal of Research and Development* **30** (1986) pp. 172-183.

[26] D. M. Young, *Iterative Solution of Large Linear Systems,* Academic Press (1971).

[27] D. M. Young, Iterative Methods for Solving Partial Difference Equations of Elliptic type, *Transcations of American Mathematical Society* **76** (1954) pp. 92-111.

[28] E. H. Cuthill and R. S. Varga, A Method of Normalized Block Iteration, *J. Asso. Comput. Mach.* **6** (1959) pp. 236-244.

SCIENTIFIC CENTER REPORT INDEXING INFORMATION

1. AUTHORS: H. H. Wang	9. SUBJECT INDEX TERMS Vector IBM 3090 Linear Algebra Matrix SOR Methods
2. TITLE: Introduction to Vectorizing Techniques on the IBM 3090 Vector Facility	
3. ORIGINATING DEPARTMENT: Palo Alto Scientific Center	
4. REPORT NUMBER: G320-3489	
5a. NUMBER OF PAGES: 32 / 5b. NUMBER OF REFERENCES: 28	

6a. DATE COMPLETED	6b. DATE OF INITIAL PRINTING	6c. DATE OF LAST PRINTING
March 1986	March 1986	June 1987

7. ABSTRACT

The advantage of vector processing is first reviewed. Then the IBM 3090 Vector Facility and software tools are briefly discussed. The major part of the report is devoted to the methodology of choosing algorithms on the target vector machine. We illustrate the method of evaluating algorithms and point out various good practices with examples in linear algebra problems commonly encountered in science and engineering.

8. REMARKS