Seismic migration on the IBM 3090 Vector Facility

by J. Gazdag
G. Radicati
P. Sguazzero
H. H. Wang

Seismic prospecting aims at determining the structure of the earth from indirect measurements. Acoustic wave fields are generated at the surface, penetrate the earth, and are backscattered by the earth's inhomogeneities. The data recorded at the surface are processed in a complex sequence of steps among which seismic migration plays an important role. This is a wave depropagation process that permits the localization in depth of the origin of the diffraction events measured (in time) at the surface. This paper presents an overview of the major wave-equation migration methods. The most frequently executed algorithms or kernels on which the execution speed depends most crucially are given particular attention. The speedup resulting from scalar-to-vector formulation is presented over wide ranges of dimensionality for linear tridiagonal equation solvers, Fourier Transforms, and convolution operations. The vectorizability and resulting speedup are also addressed in the case of migration schemes known as the Phase-Shift Method and the Phase Shift Plus Interpolation (PSPI) Method. It is shown that Fourier domain migration based on the phase-shift concept lends itself conveniently to multilevel parallelism on the 3090 Vector Facility (VF): vectorization of the innermost loops and concurrent processing in the outer loops by means of the VS FORTRAN Version 2 Multitasking Facility.

Introduction

The role of seismic signal processing is to render the recorded data more easily interpretable by geologists. This discipline represents one of the most important areas of scientific/engineering computations. To meet "number-crunching" requirements dating back more than a decade, array processors (e.g., IBM 2938 and IBM 3838) were developed specifically for the seismic industry. In recent years, demands for more computing power have transcended the capacity of array processors. At the same time, general-purpose vector processors are gaining acceptance in the seismic industry. This is particularly true for processing steps involving two-, three-, and even four-dimensional data volumes, such as seismic migration, the analysis of which is the subject of this paper. The purpose of migration is to reconstruct the reflectivity map of the earth from the seismic data recorded at the surface.

The aim of this paper is to study the vectorizability of major migration algorithms. The solution methods under consideration fall into three major categories:

- Finite-difference methods in the space-time domain.
- Spectral methods formulated in the space-frequency domain.
- Spectral methods formulated in the wavenumber domain.

The numerically most intensive algorithms (executed many thousands of times within a migration job), henceforth referred to as "kernels," are linear tridiagonal equation solvers, Fourier Transforms, and convolution operations. The speed of migration depends crucially on the performance of the corresponding kernel. Consequently, it is of considerable importance to design, program, and use these kernels optimally for particular applications. The organization of the paper is as follows. First, the basic concepts of seismic data representation are introduced. Next, the theory of migration schemes is described. Then a general

overview of performance and programming considerations is presented. This is followed by performance analyses of the most important kernels and their vectorizability. Emphasis is laid on estimating and/or measuring the speedup of the vector algorithm versus the scalar one on the IBM 3090 Vector Facility. A summary of results and observations concludes the paper.

Background

• *Acquisition and representation of seismic data*

Reflection seismology is an echo-ranging technique. An acoustic source (shot) emits a short pulse and a set of recorders (geophones) register the reflected waves at the surface. The time series (sampled) data associated with a single shot and receiver are known as a trace. In a typical marine exploration (**Figure 1**), a boat tows a source and a streamer of receivers. As it moves half a receiver interval along a seismic line, it fires a shot and records the pressure at each receiver location. A trace is associated with each shot and receiver point. Let r be the horizontal coordinate of the receiver and s be the horizontal coordinate of the source. Both are measured along the seismic line. However, for mathematical reasoning, it is helpful to represent r and s on orthogonal axes, as shown in Fig. 1. We also define the midpoint coordinate between source and receiver as $x = (r + s)/2$, and the source/receiver half-offset coordinate as $h = (r - s)/2$. From these equations we see that x and h are another set of axes rotated 45° with respect to the axes r and s.

Seismic processing techniques have been developed for groups of traces, called gathers, aligned parallel with one of the four axes shown in Fig. 1. One such processing step preceding migration is known as *stacking*. It consists of the summation of the traces of each common midpoint (CMP) gather after correcting them to compensate for the offset between source and receiver. This is known as the normal moveout (NMO) correction. When the proper amount of time shift is applied to the traces, apart from a minor distortion effect, they appear as if they were recorded with coincident source and receiver, i.e., $h = 0$. The ensemble of summed traces is referred to as the CMP stacked section. A very important benefit of this operation is the significant improvement in the signal-to-noise ratio of the CMP section in comparison with the unstacked data.

• *Wave equation migration*

Migration calls for the numerical solution of partial differential equations which govern the propagation of the recorded signals from the surface to the reflector locations, in reverse time. These methods, generally referred to as wave-equation migration, consist of two steps: *wave extrapolation* and *imaging*. Downward extrapolation results in a wave field that is an approximation of the one that would have been recorded if both sources and recorders had been located at depth z. Thus, events appearing at $t = 0$ are at their correct lateral position. Therefore, the extrapolated zero-offset data at $t = 0$ are taken as being the correctly migrated data at the current depth. These data ($t = 0$) are then mapped onto the depth section at z, the depth of extrapolation. This mapping process is also referred to as imaging.

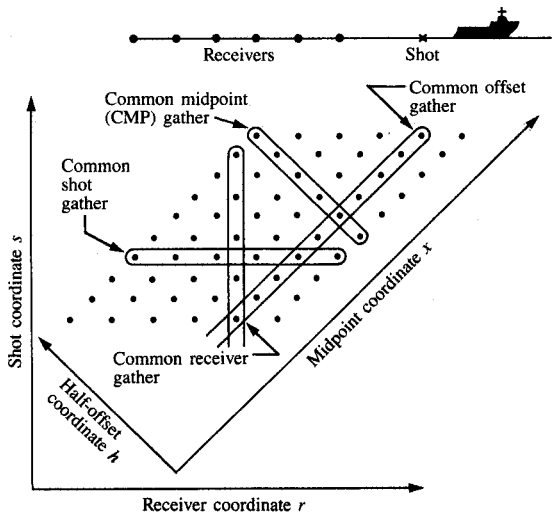

Figure 1

Relationship among the horizontal coordinates r,s,x, and h. All axes represent distances measured along the seismic line. Each dot on the surface corresponds to a seismic trace.

Let $p = p(x, z, t)$ be the CMP section (zero-offset pressure data), where x is the midpoint variable, z is depth, and t is two-way travel time. The downward extrapolation of zero-offset data is governed by the *one-way wave equation* [1]:

$$\frac{\partial P}{\partial z} = \frac{2i\omega}{v}\left[1 - \left(\frac{k_x v}{2\omega}\right)^2\right]^{1/2} P, \qquad (1)$$

where P is the Fourier Transform of p, v is the velocity, k_x is the wavenumber with respect to x, and ω is the temporal frequency. Equation (1) is expressed in the wavenumber-frequency domain (k_x, ω) and does not have an explicit representation in the midpoint-time domain (x, t).

Migration in the (k_x, ω) domain

We shall assume that within one extrapolation step, say from depth z to $z + \Delta z$, the velocity is constant. Then the solution of Eq. (1) can be expressed as

$$P(k_x, \omega, z + \Delta z) = P(k_x, \omega, z) \exp\left\{\frac{2i\omega}{v}\left[1 - \left(\frac{k_x v}{2\omega}\right)^2\right]^{1/2} \Delta z\right\}. \qquad (2)$$

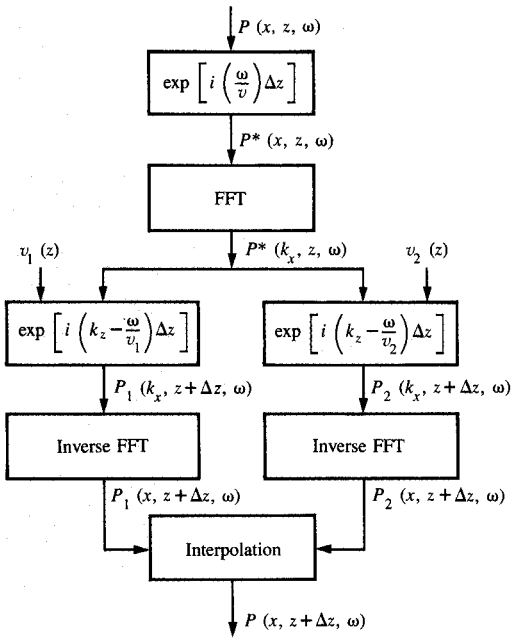

Figure 2

Computational diagram of the PSPI method, with two reference velocities.

This analytic solution states that P is extrapolated from z to $z + \Delta z$ by simply rotating its phase by a specified amount. Therefore, migration schemes based on this priciple are referred to as *phase-shift* methods [1].

The simple analytic solution expressed by Eq. (2) is not valid for velocity fields with lateral variations. In this case, the square-root expression in Eq. (1) must be approximated in some form—for example by a quadratic polynomial [2]. A second-order approximation of Eq. (1) can be written as

$$\frac{\partial P}{\partial z} = i\left[\frac{2\omega}{v} - \left(\frac{k_x^2 v}{4\omega}\right)\right] P. \tag{3}$$

When v has lateral dependence, P needs to be convolved with v or its inverse. To implement the migration algorithm efficiently, the mean velocity should be treated separately from the perturbation thereon. With regard to the mean velocity u, the wave extrapolation is accomplished by the phase-shift method, as expressed by Eq. (1) using u instead of v. The velocity perturbation about u is taken into account by what may be called the *correction term*, expressed as

$$\frac{\partial P}{\partial z} = 2i\omega P^* F\left[\frac{1}{v} - \frac{1}{u}\right] - i\left(\frac{k_x^2 v}{4\omega}\right) P^* F[v - u]. \tag{4}$$

In this expression F stands for the operation of Fourier Transform with respect to x, and * denotes convolution with respect to k_x.

Migration in the (x, ω) domain

Another rational approximation of (1) is done by truncated continued fractions [3] and splitting, which results in two extrapolators

$$\frac{\partial P}{\partial z} = \left(\frac{2i\omega}{v}\right) P, \tag{5}$$

which is known as the *thin lens* term, and

$$\left[1 + \frac{v^2}{16\omega^2}\frac{\partial^2}{\partial x^2}\right]\frac{\partial P(x, \omega, z)}{\partial z} = \left(\frac{iv}{4\omega}\right)\frac{\partial^2 P(x, \omega, z)}{\partial x^2}, \tag{6}$$

which is the *Fresnel diffraction* term. Advancing to greater depths is done by applying Eqs. (5) and (6) alternately in small Δz steps. To advance P with (6), for numerical stability considerations, an implicit Crank-Nicolson difference scheme is most often used.

An alternate approach to wave-field extrapolation in general media is described in [4] and [5]. At each Δz step the wave extrapolation is accomplished in two stages. In the first stage the wave field $P(x, \omega, z)$ given at depth z is extrapolated to $z + \Delta z$ by the phase-shift method using ℓ reference velocities $v_1, v_2, \cdots, v_\ell$ spanning the velocity range at depth z. This stage generates ℓ reference extrapolated wave fields at $z + \Delta z$, namely $P_1, P_2, \cdots, P_\ell$. In the second stage the definitive wave field $P(x, \omega, z + \Delta z)$ is constructed by interpolation of the ℓ reference wave fields. This *Phase Shift Plus Interpolation (PSPI)* method is unconditionally stable and has good dispersion relation properties worth its relatively high computational cost (for each frequency ω, and for each step in depth, $\ell + 1$ Fourier Transforms in the x direction are required). The computational diagram of the PSPI algorithm is shown in **Figure 2** for two reference velocities ($\ell = 2$).

Migration in the (x, t) domain

Equations (5) and (6) can be expressed in the *physical* (x, t) domain by Fourier-transforming both equations with respect to ω. In order to ensure stability, an implicit differencing scheme is used in extrapolating the wave field to greater depths. A comprehensive description of (x, t) domain migration techniques is given by Claerbout [6, pp. 90–102].

Migration in the (x, t) domain can also be formulated as a solution of integral equations. The Kirchhoff integral approximates the wave field at an arbitrary point as a weighted integral of the wave field recorded at the surface [5]. This approach to migration is not a *wave-equation method*, and its detailed analysis is outside the scope of this paper.

An example of migration of synthetic seismic data

As we have discussed earlier, migration is an inverse process. To test a migration scheme and evaluate its performance, we need a set of seismic data, e.g., a zero-offset section obtained from an idealized model with known reflectivity and

velocities. This is usually done by simulating the forward process using one of the standard forward modeling schemes [4, 5]. An example of such a modeling approach representing a set of dipping reflectors is shown in **Figure 3**. On the schematic of the model [Fig. 3(a)], the reflectors are indicated by thick lines to distinguish them from interfaces of velocity regions depicted by thin lines. Figure 3(b) illustrates the synthetic zero-offset section. **Figure 4** shows the migrated results obtained by a finite-difference method, based on Eqs. (5) and (6), and the PSPI method. Both migrated sections are of high quality, and they represent a good reconstruction of the reflectors of the model. An interesting feature superimposed on the migrated results is a circular pattern extending down to the $z = 5$ level. These are diffracted waves originating from the neighborhood of $(x = 8, z = 3)$, where the region in which $v = 4$ km/s narrows to a point.

Vector algorithms

• *General principles*

The number of operations within an algorithm that can be executed independently and therefore can be performed in parallel is referred to as the *parallelism of the algorithm* [7]. These independent operations can, in principle, be performed concurrently, or simultaneously, as one may wish to do in certain parallel architectures such as processor arrays. A pipelined computer is organized in a way similar to an assembly line. Operations are divided into subtasks that are executed by specialized hardware stages. Operands flow from one stage to the next, so that the various stages may concurrently process different operands. Successive tasks are executed in the pipeline in an overlapped fashion.

From the programmer's viewpoint, it makes little difference whether parallel operations are executed simultaneously or in a time sequence with considerable overlapping among the operations. The data are defined as vectors, and the operations correspond to vector instructions. Vector instructions, which operate on arrays of data rather than on single data items, serve the purpose of feeding data to the pipelined units at an optimal rate. If properly used, vector instructions can result in a tremendous improvement of system throughput.

To obtain sustained high performance on a vector computer, it is essential that the algorithms be designed specifically for the architecture of the computer under consideration. There are three approaches to implementing vector algorithms: (1) programming in FORTRAN, (2) programming in Assembler, and (3) using a set of routines optimized for the vector architecture under consideration. In the first case, while the program development is easier and more efficient, one needs to rely on the ability of the compiler to vectorize the FORTRAN program. In the second case, at the expense of working with a low-level

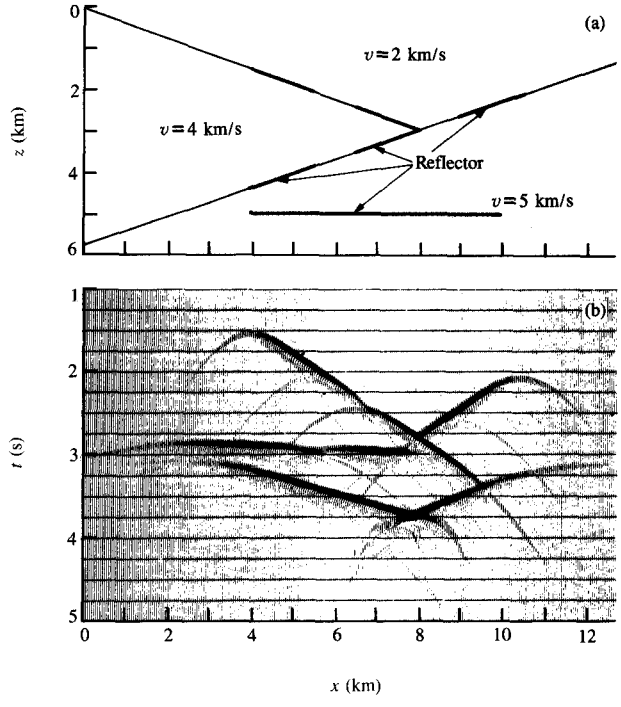

Figure 3

(a) A schematic model representing a dipping multilayer example. The thick-line segments denote where the reflector segments have been "turned on." (b) Synthetic zero-offset time section of the model shown in (a).

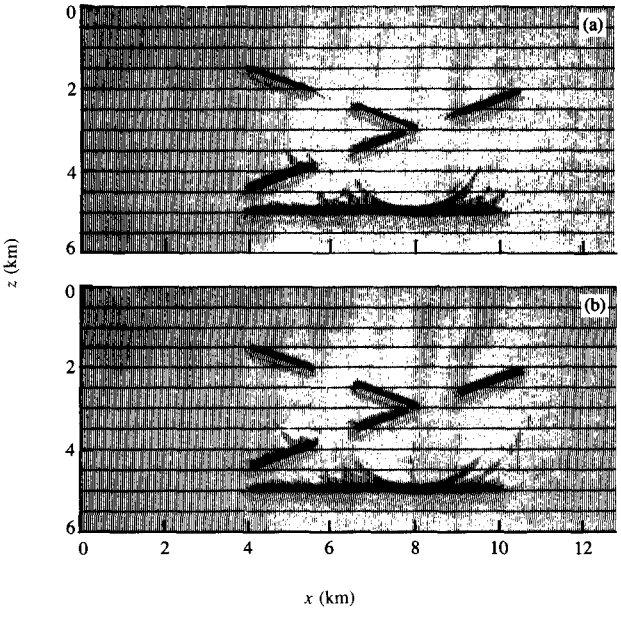

Figure 4

Depth migration sections obtained from the zero-offset section shown in Fig. 3(b) by means of (a) a finite-difference method and (b) the PSPI method.

language, one can select an optimal set of vector instructions to define the algorithm. A well-written assembler code tailored to a particular system is likely to outperform the code produced by any compiler. While this has always been true for serial computers, it is even more valid for vector computers. The ratio of performance between good and bad programs is substantially higher on vector computers than on serial computers. For the third approach a highly tuned subroutine library, the Engineering and Scientific Subroutine Library (ESSL) [8], has been developed for the 3090 VF. It contains routines for the solution of systems of linear equations, eigenvalue problems, Fast Fourier Transforms, etc.

• *Performance*

It is quite customary to measure the performance of a computer in millions of instructions per second (MIPS) or, in the case of a scientific computer, in millions of floating-point operations per second (MFLOPS). In this paper we are not interested in measuring hardware performance and we do not wish to make any comparison with any other hardware. Our primary interest lies in the relative performance of the 3090 in vector and scalar mode, in order to have a reasonably good appreciation for the incremental performance due to the vector feature. We define the performance of a computer program as the *inverse of the CPU time* required. This is not the only definition that could have been given. We could have taken into account the use of the memory and other storage devices. However, in the context of this paper we cannot address complicated performance measures of this type.

It should be noted that, on vector computers, minimum execution time is not necessarily synonymous with minimum number of floating-point operations. The gain in speed resulting from executing the program in vector mode may outweigh the cost of extra arithmetic operations. An example of this is the tridiagonal system solver, where the vector algorithm requires about twice as many operations as the scalar algorithm but is faster than the scalar version for sufficiently long vectors. Finally, the performance of a vector code may depend to a great extent on the average vector length, on the startup time of the instructions, and on such factors as how well the parallelism in the algorithm matches the parallelism of the computer.

Our primary goal in examining some of the most important kernels is to estimate the relative performance of the *best available vector code versus its scalar counterpart*. In the simpler algorithms, such as the *phase-shift operator*, the vector code is obtained from the scalar FORTRAN by using the VS FORTRAN Version 2 compiler. In the more complicated situations, the scalar and vector programs are based on different algorithms, as in the case of Fast Fourier Transforms (FFT), where we compare the performance of Short-Precision Complex-to-Complex Fast Fourier Transform (SCFT) from ESSL and a scalar program based on a different algorithm. In the case of *convolution* we compare the performance of different vector algorithms to underscore the importance of tailoring the code to the architecture under consideration. Thus our comparisons refer to the performance of specific scalar and vector algorithms. The relative performance improvement is measured as the ratio of the *scalar and vector* execution time, which we refer to as *speedup*. Speedup figures of different algorithms show considerable variation. Thus, for example, the measured speedup for convolution was more than 10, and less than 1.5 for tridiagonal system solvers.

• *The 3090 VF: An overview*

The 3090 VF, as seen by the programmer, is essentially a 370 machine with the addition of vector instructions and vector registers [9]. The length of the vector registers is model-dependent, and in the following discussion we assume the length to be 128, as on the machine on which we ran our tests. The vector instructions process at most 128 elements at a time; vectors of length greater than 128 must be sectioned, in a process that is quite similar to what happens on machines like the CRAY. In general, to maximize the performance, data loaded into a vector register should be held there for as many arithmetic operations as possible.

We now describe the formats of the vector instructions. In what follows, *FPR* stands for a floating-point register, and *VRn* stands for Vector Register *n*. The three basic floating-point vector instructions (multiplication, addition, and subtraction) have four different formats:

$$VR1 = VR2 \ op \ memory, \tag{7}$$

$$VR1 = FPR \ op \ memory, \tag{8}$$

$$VR1 = FPR \ op \ VR2, \tag{9}$$

$$VR1 = VR2 \ op \ VR3. \tag{10}$$

Floating-point division has only format (7). An input register may also serve as an output register. In (7) and (8) one of the vector operands comes directly from memory, without having to be loaded in a vector register, and may be accessed with stride (positive or negative). In all formats after a startup time of approximately 20 cycles, the vector instructions deliver one result per machine cycle. After the first 128 elements of a vector have been processed, the operations must be reinitialized on the next section.

There are also two compound instructions, *multiply and accumulate* (VMC) and *multiply-add* (VMA), which perform a multiplication and an addition at the same time and deserve special attention.

- In the *multiply and accumulate*, a vector register is multiplied by another vector and the result is accumulated in another vector register, as shown below:

$$VR0_1 = VR0_1 + \sum_k VR2_k W_k$$

$$\text{for} \quad k = 1, 5, 9, 13, \cdots, 125, \tag{11}$$

$$VR0_2 = VR0_2 + \sum_k VR2_k W_k$$

$$\text{for} \quad k = 2, 6, 10, 14, \cdots, 126, \tag{12}$$

$$VR0_3 = VR0_3 + \sum_k VR2_k W_k$$

$$\text{for} \quad k = 3, 7, 11, 15, \cdots, 127, \tag{13}$$

$$VR0_4 = VR0_4 + \sum_k VR2_k W_k$$

$$\text{for} \quad k = 4, 8, 12, 16, \cdots, 128 \tag{14}$$

In these equations, W represents data residing either in a vector register or in memory. Vector register *VR0* contains, in the first four positions, partial results. The final result is obtained by summing over the first four elements of *VR0* with the instruction *sum-partial-sums* (VSPSD).

- The *multiply-add* instruction has the following formats:

$$VR0 = VR0 + FPR \times memory, \tag{15}$$

$$VR0 = VR0 + FPR \times VR2, \tag{16}$$

$$VR0 = VR0 + VR2 \times memory. \tag{17}$$

After some startup time, these compound instructions deliver one result per machine cycle. This feature of the 3090 VF to perform multiple operations within one machine cycle is similar to the chaining concept of the CRAY computers.

When the vector length n exceeds 128, the vector operation is partitioned into

$$S(n) = 1 + \left[\frac{n-1}{128}\right] \tag{18}$$

sections, where $[\cdots]$ signifies the integer part of the bracketed expression. Each section requires some startup time β, typically 20 machine cycles, to set up the pipeline. However, when considering theoretical performance models, it is more realistic to assume the startup time to be approximately 35 cycles, to include some auxiliary scalar instructions (load address, etc.). After this overhead, the pipeline delivers one result every α cycles (α is 1 for most instructions and is larger for the more complicated instructions). Using these definitions, the time required for a vector operation is

$$\tau(n) = \alpha n + \beta S(n), \tag{19}$$

expressed in machine cycles.

Performance analysis of migration kernels

Our aim is to give the reader an estimate of the performance improvement that can reasonably be expected from the Vector Facility on the IBM 3090 system. The speedup curves we describe are derived from actual measurements of execution times. We make no effort to analyze a complete migration program in detail. Instead, we focus our attention on the most significant computational kernels of these programs, the optimization of which plays a decisive role in the global performance of migration algorithms.

- *Representative kernels*

Seismic migration codes are characterized by one or two algorithms that account for most of the floating-point operations or FLOPs. These algorithms, which may be a relatively small subset of the entire migration program, are executed hundreds of thousands of times within a job step. Consequently, their implementation plays a key role in determining the overall efficiency of the migration code under consideration. In what follows, these algorithms are referred to as representative *kernels*. In the migration methods discussed above, one can identify four such kernels, which are listed below.

Kernel 1: the phase-shift operator involves the evaluation of the exponent in Eq. (2) followed by a complex multiplication. This kernel is implemented for data corresponding to a fixed $\omega = \omega_j$ and $z = z_n$, i.e., for the vector $P(k_x, \omega_j, z_n)$. The vector elements corresponding to different k_x are independent of one another; thus they lend themselves conveniently to vectorization.

Kernel 2: the convolution of two functions $f(x)$ and $g(x)$ is defined as

$$h(x) = \int_{x-a}^{x+a} f(x - y)g(y)dy. \tag{20}$$

This is one of the most often executed algorithms in seismic data processing. While various approaches to seismic migration, e.g. Eq. (4), can be expressed in terms of convolution, this algorithm is used most often in *deconvolution*, a time filtering of seismic traces.

Kernel 3: Fourier Transform. The widespread use of the Fast Fourier Transform algorithm is well known in signal processing, and there is little need to emphasize its importance further. Its intensive use is most apparent in the PSPI migration algorithm, which is a spectral method requiring repeated Fourier Transforms of 1D or 2D arrays at all depths z and all temporal frequencies ω of interest.

Kernel 4: tridiagonal systems of equations are very important since they occur frequently in finite-difference approximations to differential equations, e.g., Helmholtz, Poisson, diffusion, and wave equations. The wave extrapolation equation (6) and (x, t) domain migration techniques are based on the solution of tridiagonal systems.

- *The phase-shift operator*

The major computation task in migration by the phase-shift method is the evaluation of Eq. (2). Numerically, this is

equivalent to multiplying each relevant complex Fourier coefficient by a complex number of unit modulus. The following FORTRAN DO-loop is the kernel of a phase-shift program:

```
      DO 10 IX = 2, LIM
         THETA = SQRT(1.0 - VPWSQ*KXSQ(IX))*DZ*WPERV
         ROTATE = CMPLX(COS(THETA), SIN(THETA))
C     APPLY PHASE SHIFT TO POSITIVE WAVENUMBERS
         P(IX) = P(IX)*ROTATE
C     APPLY PHASE SHIFT TO NEGATIVE WAVENUMBERS
         INX = NX - (IX - 2)
         Q(INX) = Q(INX)*ROTATE
   10 CONTINUE
```

The DO index limit, LIM, corresponds to the highest wavenumber k_x and is bounded by NX/2, and THETA is the quantity inside the curved brackets of Eq. (2). The VS FORTRAN Version 2 compiler vectorizes this DO-loop, generating calls to vector-valued intrinsic functions SQRT, COMPLX, SIN, and COS. We measured a scalar-to-vector speedup of 2.1 on the phase-shift kernel of a migration program.

• *Convolution*

Let $u = (u_1, u_2, \cdots, u_n)$ and $x = (x_1, x_2, \cdots, x_{m+n-1})$ be sequences of length n and $n + m - 1$, respectively. The convolution $u*x$ of these sequences is a sequence of length m, defined as

$$(u*x)_i = \sum_{k=1}^{n} u_k x_{n+i-k} \quad \text{for} \quad i = 1, \cdots, m. \tag{21}$$

This operation plays a very important role in signal processing. When both sequences are long, it is usually implemented by means of algorithms based on Fast Fourier Transforms. Here, we are concerned with a straightforward implementation of Eq. (21). As we shall see, the compound instructions discussed above are well suited to implementing (21) and result in very high vector performances.

There are two ways of implementing Eq. (21). The first one is described by the following FORTRAN loops, referred to as Algorithm 1:

```
      DIMENSION U(N), X(M + N - 1), Y(M)
      DO 2 I = 1, M
      Y(I) = 0.
         DO 1 K = 1, N
         Y(I) = Y(I) + U(K)*X(N + I - K)
    1    CONTINUE
    2 CONTINUE
```

The inner loop, over index k, is a *scalar product* of length n between u and sections of vector x. One can see that loops 1 and 2 can be interchanged without affecting the result. This second implementation is referred to as Algorithm 2:

```
      DIMENSION U(N), X(M + N - 1), Y(M)
      DO 3 I = 1, M
      Y(I) = 0.
    3 CONTINUE
      DO 5 K = 1, N
         DO 4 I = 1, M
         Y(I) = Y(I) + U(K)*X(N + I - K)
    4    CONTINUE
    5 CONTINUE
```

In this case the inner loop, over index i, is of the form $V1 = V1 + s \times V2$, where $V1$ and $V2$ are vectors and $s = u_k$ is a scalar.

Both algorithms can be implemented on the 3090 VF by making use of high-performance chained, or compound, instructions, which perform two floating-point operations per machine cycle. Using other vector instructions, such as addition, multiplication, etc., it is possible to perform only one floating-point operation per machine cycle. It is shown that the performance of Algorithm 2 is much better than that of Algorithm 1. This sort of loop reordering is common to many problems of linear algebra on vector processors. Problems of this kind are discussed, for example, by Dongarra et al. [10].

Algorithm 1

We now describe two implementations of the first algorithm using the vector instructions.

For $i = 1, 2, \cdots, m$,

Zero-partial-sums.
For $k = 1, 129, \cdots, n$,

Vector length $l = \min(128, 1 + n - k)$.
Load l elements of u to a vector register starting from u_k.
Multiply and accumulate for l elements over u and x starting from x_{n+i-k} with stride -1.

Sum-partial-sums.
Store result into y_i.

Assuming a startup time of 35 cycles for each vector instruction, the timing equation for this algorithm can be approximated as

$$\tau_1 = 2mn + m[90 + 105S(n)], \tag{22}$$

where $S(n)$ is the number of sections in a vector of length n as defined by (19). The first term accounts for the *load* of u and the *multiply and accumulate*, which are performed m times. The second term accounts for the *zero-partial-sums*, the *sum-partial-sums*, and the *store* of y_i. The third term in the equation accounts for the startup time for the three vector instructions in the main loop. This first implementation of Algorithm 1 is rather inefficient due to the repeated loading of vector u into the vector register.

When this shortcoming is corrected, the performance increases significantly. A second implementation of Algorithm 1 is based on loading one section of u and forming its scalar product with all the sections of x. Repeating this procedure for all the sections of u and summing the partial results yields the desired result. Each section of u is loaded only once, in exchange for doing a *zero-partial-sums*, a *sum-partial-sums*, a load, a sum, and a store of y_i, m times for each section of u. The timing equation of this second implementation is

$$\tau_2 = mn + m[1 + 120S(n)] + n + 70S(n) + 35S(m). \quad (23)$$

Algorithm 2

The vector implementation of Algorithm 2 can be summarized as follows:

For $i = 1, 129, \cdots, m$,

Vector length $l = \min(128, 1 + m - i)$.
Clear VR0 for l elements.
For $k = 1, 2, \cdots, n$,

Load u_k to *FPR*.
VR0 = *VR0* + *FPR* × l elements of x starting from x_{n+i-k} with stride 1.

Store l elements of *VR0* starting from y_i.

This algorithm is based on the *multiply-add* instruction. Note that no vector load instruction is ever used, and the sectioning is performed with respect to the index i. Overhead is minimized by performing the sectioning on the outermost loop. Assuming a startup time of 35 cycles for all the vector instructions, the timing equation for this algorithm is

$$\tau_3 = mn + 45nS(m) + 2m + 105S(m). \quad (24)$$

The first two terms account for the inner loop; the last two terms account for the clearing of the accumulation register, the *store* of vector y, and the computation of the vector length.

Comparison of performance

In order to illustrate performance improvements resulting from careful coding, the performance curves for the three implementations of the convolution operation above are shown in **Figure 5**. These plots represent floating-point operations per machine cycle versus m for $n = 64$ [Eq. (21)]. These curves were obtained by dividing $2nm$, the total number of floating-point operations in the algorithm, by the estimated execution time given in (22), (23), and (24). The cost of the multiple loads of vector u in Eq. (22) is obvious. For vectors with large m, Algorithm 2 is significantly better than Algorithm 1. The crossover point depends on n. In the case of $n = 64$, Algorithm 2 is more efficient for vectors with $m > 25$. Even for large n, Algorithm 2 is always more efficient if $m > 50$. The lower performance of Algorithm 1 is attributable to the loading of vector u and the performing of *sum-partial-sums* several times for each y_j. The effect caused by the sectioning of vector operations in Algorithm 2 is evident. The performance drops very sharply just after m reaches values that are integer multiples of 128. This is due to the overhead associated with the additional section containing only a few elements. Note that because of the startup time for vector instructions and the loads of u_k to floating-point registers, the asymptotic performance is 1.5 operations per cycle, rather than 2. The performance of Algorithm 1, shown in Fig. 5 for $n = 64$, increases monotonically with n up to 128. However, its maximum performance is about one floating-point operation per machine cycle. These curves illustrate the importance of tailoring the algorithm to the architecture of the computer.

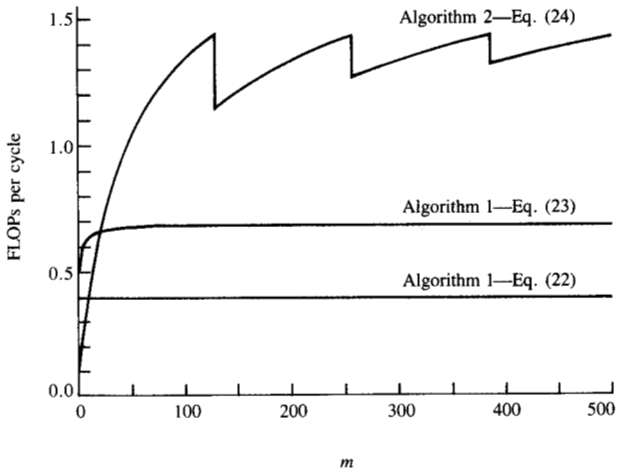

Figure 5

Number of floating-point operations per machine cycle as a function of m for convolution.

To illustrate the scalar-to-vector speedup for convolution, we consider Algorithm 2. First we measured τ_s, the execution time for a scalar convolution algorithm. The speedup τ_s/τ_3 is shown in **Figure 6** (solid curve), where τ_3 is given by Eq. (24). To verify these predictions, we also measured the actual vector execution time corresponding to τ_3. The measured speedup figures are depicted by the symbol + in Fig. 6. There is a good agreement between the predicted and the measured speedup values.

• *Fast Fourier Transform*

The FFT algorithm plays a very important role in signal processing. The motivation for this is twofold: convenience

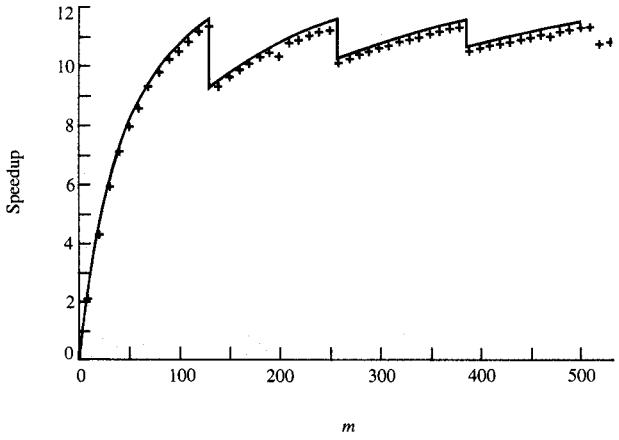

Figure 6

Scalar-to-vector speedup for convolution on the 3090 as a function of m. The vector algorithm is an implementation of Algorithm 2. The symbol + represents measured values; the continuous line is an estimate based on (24).

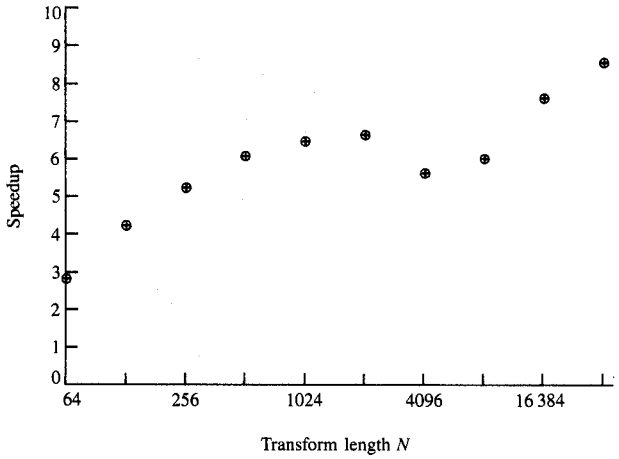

Figure 7

Speedup of the vector vs the scalar FFT.

and economy. The first category includes applications in which the operations on the data are more easily formulated and/or these operations are performed more accurately in the frequency or wavenumber domain. One example is directional or fan filtering of two- or three-dimensional data. Another example is migration by phase shift. Equations (1) and (2) are exact statements of wave extrapolation in the wavenumber-frequency domain, which cannot be expressed in the space-time domain. The second category comprises operations defined in the space and/or time domain, but the execution of which is carried out more economically by means of Fourier Transform techniques. Filtering of one-, two-, or three-dimensional data and numerical methods involving long linear operators serve as representative examples. Here one takes advantage of the convolution theorem (Brigham [11]) and the speed of the FFT algorithm.

For the FFT algorithm, our only objective was to measure the speedup from scalar to vector operations. For the scalar case we used the FFT routine by Singleton [12], the performance of which is comparable to that of the well-known HARM routine [13]. The vector FFT algorithm is described by Agarwal [14] and by Agarwal and Cooley [15] and is part of ESSL. The speedup curve (**Figure 7**) is based on timings of these routines performed on the 3090 VF. Note that the speedup decreases when passing from transforms of complex arrays of 2048 elements to 4096 elements. This problem is caused by the limited size of the cache [16]. For transforms of length 4096 and beyond, the data and the associated working and coefficient arrays do not fit in cache. Therefore, the problem is broken into pieces which fit in the cache; this requires multiple access of data and an intermediate two-dimensional array transposition. This degrades the performance by about 10%, but this drop in performance is not as much as in the case of a conventional scalar FFT for transforms which do not fit in cache (16 384 and beyond).

- *Tridiagonal solvers*

The solution of a tridiagonal system of linear equations lies at the heart of many scientific programs. 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 to vectorization. In the past dozen years, several new algorithms have been proposed for solving tridiagonal systems on parallel and vector computers [17, 18]. The most commonly used vector algorithms are based on a method originally proposed by Gene Golub and Roger Hockney and known as cyclic reduction [19, 20]. The vectorization is achieved at a cost of approximately twice the arithmetic operations required by Gaussian elimination. In what follows, we briefly describe the scalar and the vector algorithms and the speedup curves we measured.

Gaussian elimination

Consider the tridiagonal system of n linear equations

$$Ax = r, \tag{25}$$

where A is a tridiagonal matrix with its ith row denoted by $(\cdots, c_i, a_i, b_i, \cdots)$. The solution of (25) can be accomplished in the following two steps:

1. Forward step:

$$d_1 = 1/a_1, \tag{26}$$

$$y_1 = r_1, \tag{27}$$

$$d_i = 1/(a_i - c_i d_{i-1} b_{i-1}), \qquad i = 2, \cdots, n, \tag{28}$$

$$y_i = r_i - c_i d_{i-1} y_{i-1}, \qquad i = 2, \cdots, n. \tag{29}$$

2. Backward step:

$$x_n = y_n d_n, \tag{30}$$

$$x_i = (y_i - x_{i+1} b_i) d_i, \qquad i = n-1, \cdots, 1. \tag{31}$$

In the actual implementation, 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.

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 the original size. This process can be repeated on the reduced system. After $p = \log_2 n$ steps of such elimination, there is only one equation left, which can easily be solved. Other solutions can then be obtained by back substitution.

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

- The vector length changes by a factor of 2 in each step. In the last steps, even for large n, the vectors being processed are short.
- The large memory requirements and the complicated addressing scheme result in less than optimal use of the cache when solving large systems.

Scalar-to-vector speedup

We have compared the performance on a 3090 with Vector Facility of a Gaussian elimination code with that of an assembly language cyclic reduction code based on the algorithm described by Kershaw [20]. Note that neither the cyclic reduction algorithm nor the Gaussian elimination algorithm perform pivoting. **Figure 8** shows the speedup versus the order of the matrix, n. The speedup for this problem is small because of the relatively small vector-to-scalar speed ratio and because of the limited size of the cache.

Performance measurements of the PSPI code

The PSPI method was developed [4] as a response to the following shortcomings exhibited by finite-difference migration methods: (1) an inaccurate dispersion relation for steep dips, (2) errors due to difference approximations, and (3) predisposition to numerical instability. To keep the stability problem under control, implicit difference schemes are used to solve the wave extrapolation equations, e.g., Eq. (6). These implicit schemes call for the evaluation of tridiagonal systems of equations for two-dimensional data. However, Claerbout [6, p. 100] finds that "in space dimensions higher than one the implicit method becomes prohibitively costly." The PSPI migration scheme, on the other hand, has proved to be accurate, unconditionally stable, and applicable to data of any dimension. One of the aims of this study is to show that the PSPI algorithm is also easily vectorizable and that the scalar-to-vector speedup is significant.

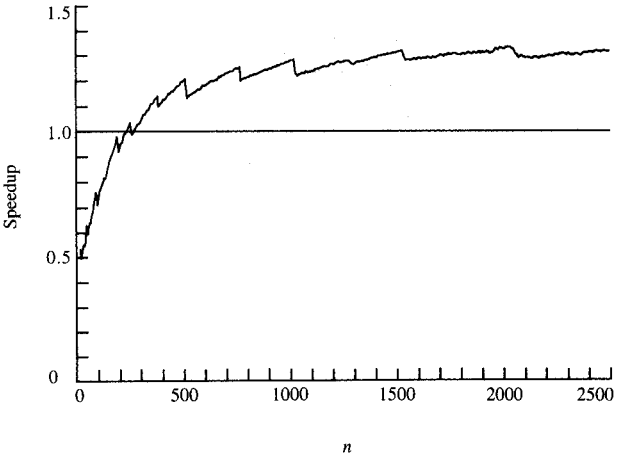

Figure 8

Scalar-to-vector speedup for tridiagonal linear system solvers as a function of n.

The vectorizability and speedup characteristics of the PSPI code were studied by running it and measuring execution times on the 3090 computer in scalar and vector modes. The problem under consideration consisted of a synthetic zero-offset section of size 512 × 512. The vectorized version of the code was developed from the original scalar code by introducing relatively minor modifications, e.g., loop reordering. The idea is to enable the VS FORTRAN Version 2 Compiler to recognize "independence" among a set of operations and to convert them into vector object programs. Only one subroutine was replaced by its vector counterpart: All calls to HARM were replaced by calls to SCFT from ESSL [8].

As one can see in Fig. 2, the PSPI algorithm consists of three major operations: Fourier transformation, complex multiplication, and interpolation between the wave fields. **Table 1** summarizes the performance data of the PSPI algorithm. The measurements do not include I/O, the initial transformation from the time to the frequency domain, parameter initialization, etc., operations that account for less

Table 1 Performance of the PSPI code.

	Scalar code		*Vector code*		*Scalar-to-vector speedup*
	CPU (s)	%	*CPU* (s)	%	
FFT	1181.1	61.8	180.6	39.1	6.5
Complex multiply	173.7	9.0	37.3	8.0	4.6
Interpolation	478.7	25.5	209.7	45.4	2.2
Other	67.5	3.9	34.4	7.4	1.9
Total	1910.0		462.0		4.1

than 1% of the total execution time in the scalar version of the code. Table 1 summarizes the results of our measurements by showing the CPU time, in seconds, for the main part of the code, the relative importance of the different portions of the code, and the scalar-to-vector speedup for each part. The best speedup is observed for the FFT, which does not come as a surprise, as SCFT is carefully tuned to the 3090 VF. It may be possible to improve the performance of the other parts. The overall speedup of 4 that we measured is a rather impressive figure.

As we mentioned earlier, the data and the computations associated with different frequencies ω are independent. We took advantage of this to run the loop over the frequencies in parallel on the two CPUs of the 3090 VF. The software tool we used to do this was the VS FORTRAN Version 2 Multitasking Facility [21]. This facility is a set of routines that can be called by a FORTRAN program and allows multiple MVS tasks to be started and synchronized. In the case of PSPI, we started two identical tasks that processed half of the frequencies each. We had only to make minor changes to the original scalar code. The overhead involved in this process is very small, particularly if compared to the coarse granularity of the problem. We measured a twofold speedup when running on a dedicated system using two CPUs, which is equivalent to an effective speedup of about 8 compared to the scalar uniprocessor version.

Concluding remarks

We have presented an overview of wave-equation migration of seismic data and analyzed the potential of their vectorizability on the IBM 3090 Vector Facility. The numerical solution of the one-directional wave equations can be carried out by finite-difference methods or by spectral methods based on Fourier Transform techniques. The finite-difference approach requires the repeated solution of tridiagonal systems of equations. In spectral methods Fourier Transforms and convolution operations are used extensively. The analysis of these representative kernels shows that the speedup gained by vectorizing tridiagonal solvers is practically nothing up to the order of 300, and very little above that. On the other hand, Fourier Transforms with SCFT of ESSL on the IBM 3090 VF are about six times faster than in scalar mode using HARM for a sequence of size 512 or greater, which is the range of interest in seismic data processing. Speedup figures for convolution were even more impressive. Compound vector instructions can be used to make the vector code more than ten times faster than its scalar counterpart.

The choice of finite-difference methods has always been influenced by considerations related to cost-effectiveness, and their development during the past three or four decades has been centered around the scalar machine. The *best* scalar algorithms were those that required the *fewest* operations. This, however, is not the only consideration for vector algorithms; their goodness depends also on how well they take advantage of the particular vector architecture under consideration. The relative speed of different migration methods may be significantly different on scalar and vector processors, and the effective adaptation of scalar algorithms to vector computers remains a challenge for some time in the future.

Spectral methods based on Fourier Transform techniques are characterized by an intrinsically parallel structure. This parallelism implies a certain degree of independence among the elements of some data set and operations performed thereon, and consequently the algorithm is more easily adapted to vector architecture. The Phase-Shift Method and the PSPI Method are excellent examples for demonstrating this inherent parallelism. In addition to their vectorizability with respect to the horizontal coordinate, x or k_x, all computations associated with different frequencies ω can also be performed independently of one another. This means that Fourier domain migration based on the phase-shift concept lends itself conveniently to multilevel parallelism on the 3090 VF: vectorization of the innermost loops and concurrent processing in the outer loops, e.g., by means of the VS FORTRAN Version 2 Multitasking Facility.

References

1. J. Gazdag, "Wave Equation Migration with the Phase-Shift Method," *Geophys.* **43,** 1342–1351 (1978).
2. J. F. Claerbout, "Coarse Grid Calculation of Waves in Inhomogeneous Media with Application to the Delineation of Complicated Seismic Structures," *Geophys.* **35,** 407–418 (1970).
3. R. Clayton and B. Engquist, "Absorbing Boundary Conditions for Acoustic and Elastic Wave Equations," *Bull. Seis. Soc. Amer.* **67,** 1529–1540 (1977).
4. J. Gazdag and P. Sguazzero, "Migration of Seismic Data by Phase Shift Plus Interpolation," *Geophys.* **49,** 124–131 (1984).

5. J. Gazdag and P. Sguazzero, "Migration of Seismic Data," *Proc. IEEE* **72,** 1302–1315 (1984).
6. J. F. Claerbout, *Imaging the Earth's Interior*, Blackwell Scientific Publications, Oxford, England, 1985.
7. R. W. Hockney and C. R. Jesshope, *Parallel Computers*, Adam Hilger Ltd., Bristol, England, 1981.
8. *Engineering and Scientific Subroutine Library, General Description*, Order No. GC23-0182, available through IBM branch offices.
9. *System/370 Vector Operations*, Order No. SA22-7125, available through IBM branch offices.
10. J. J. Dongarra, F. G. Gustavson, and A. Karp, "Implementing Linear Algebra Algorithms for Dense Matrices on a Vector Pipeline Machine," *SIAM Rev.* **26,** 91–112 (1984).
11. E. O. Brigham, *The Fast Fourier Transform*, Prentice-Hall, Inc., Englewood Cliffs, NJ, 1974.
12. R. C. Singleton, "Mixed Radix Fast Fourier Transforms," *Programs for Digital Signal Processing*, Digital Signal Processing Committee, Eds., IEEE Press, New York, NY, 1979.
13. *System/360 Scientific Subroutine Package Version III*, Order No. H20-0205, available through IBM branch offices.
14. R. C. Agarwal, "An Efficient Formulation of the Mixed-Radix FFT Algorithm," presented at the International Conference on Computers, Systems, and Signal Processing, Bangalore, India, December 10–12, 1984.
15. R. C. Agarwal and J. W. Cooley, "Fourier Transform and Convolution Subroutines for the IBM 3090 Vector Facility," *IBM J. Res. Develop.* **30,** No. 2, 145–162 (1986, this issue).
16. R. C. Agarwal, IBM Research Division, Yorktown Heights, NY, private communication, 1985.
17. R. W. Hockney, "A Fast Direct Solution of Poisson's Equation Using Fourier Analysis," *J. ACM* **12,** 95–113 (1965).
18. H. S. Stone, "Parallel Tridiagonal Equation Solvers," *ACM Trans. Math. Software* **1,** 289–307 (1975).
19. J. J. Lambiotte and R. G. Voigt, "The Solution of Tridiagonal Linear Systems on the CDC Star-100 Computer," *ACM Trans. Math. Software* **1,** 308–329 (1975).
20. D. Kershaw, "Solution of Single Tridiagonal Linear Systems and Vectorization of the ICCG Algorithm on the Cray-1," *Parallel Computations*, G. Rodrigue, Ed., Academic Press, Inc., New York, 1982, pp. 85–100.
21. *VS FORTRAN Version 2, General Description*, Order No. GC26-4219, available through IBM branch offices.

Received July 11, 1985; accepted for publication October 2, 1985

Jenö Gazdag *IBM Scientific Center, 1530 Page Mill Road, Palo Alto, California 94304.* Dr. Gazdag received his B.E. from McGill University, Montreal, Canada, in 1959, his M.A.Sc. from the University of Toronto, Canada, in 1961, and his Ph.D. from the University of Illinois, Urbana, in 1966, all in electrical engineering. From 1961 to 1962 he was employed by the National Research Council of Canada in Ottawa. In 1966, he joined IBM's Research Division. He is now manager of engineering and scientific applications at the IBM Scientific Center in Palo Alto. His main research interest is in the field of numerical solution methods for partial differential equations with applications in seismic data processing. Dr. Gazdag is a member of the European Association of Exploration Geophysicists, the Institute of Electrical and Electronics Engineers, the Society of Exploration Geophysicists, and the Society of Industrial and Applied Mathematics.

Giuseppe Radicati di Brozolo *IBM Italy, Scientific Center/European Center for Scientific and Engineering Computing, Via Giorgione 159, 00147 Rome, Italy.* Dr. Radicati received a doctorate in mathematics from the University of Pisa in 1981. He joined IBM in 1982 and since that time has been involved in seismic data processing. In 1985 he participated in the design and development of VPSS/VF, the simulator of the IBM 3838 Array Processor on the IBM 3090 Vector Facility. His current research interests are seismic data processing and algorithms for parallel and vector architectures.

Piero Sguazzero *IBM Italy, Scientific Center/European Center for Scientific and Engineering Computing, Via Giorgione 159, 00147 Rome, Italy.* Dr. Sguazzero received his doctorate in mathematics from the University of Trieste in 1969. He has held positions at the University of Trieste and at the National Council of Research of Italy. Since 1970 he has been with the IBM Italy Scientific Centers, first in Venice (1970–1978), where he worked in the area of numerical modeling in hydrodynamics and contributed to the development of two numerical models of the Lagoon of Venice, and then in Rome. His present interest is the development of algorithms and software for geophysical problems, in particular seismic migration and velocity analysis with special attention to the algorithmic implications of parallel and vector architectures. In 1974 and 1978 he was a visiting staff member at the IBM Palo Alto Scientific Center. Dr. Sguazzero is a member of the Society of Exploration Geophysicists.

Hsuan-Heng Wang *IBM Scientific Center, 1530 Page Mill Road, Palo Alto, California 94304.* Dr. Wang joined IBM in Poughkeepsie after receiving a Ph.D. in mathematics from the University of Texas, Austin, in 1964. He is currently a staff member at the Palo Alto Scientific Center. After joining IBM, Dr. Wang worked in the area of scientific computing and machine evaluation; more recently, he has been active in vector processing. He has received an IBM Outstanding Technical Achievement Award for his work on developing numerical algorithms for vector machines. He is a member of the Institute of Electrical and Electronics Engineers Computer Society and the Society for Industrial and Applied Mathematics.