SEISMIC MIGRATION ALGORITHMS ON PARALLEL COMPUTERS

Vijay K. Madisetti
David G. Messerschmitt

Electrical Engineering and Computer Sciences
The University of California at Berkeley
Berkeley, CA 94720

ABSTRACT

Seismic signal processing is computationally intensive and sequential algorithms in use do not exploit the concurrency inherent in data migration techniques. In this paper we study seismic migration algorithms with a view to present a general framework for analyzing such algorithms and propose some coarse-grained paradigms for computation on parallel computers. We analyze a few typical examples of the different migration techniques existing in literature, and discuss techniques to optimize their performance on distributed concurrently executing processors. The computational and communication requirements of the algorithms are discussed and diverse optimization techniques are proposed. The memory limitation, I/O bottleneck and computational tradeoffs on hypercube multiprocessors are analyzed [1,8,9].

1. Introduction

Migration of seismic data involves repositioning the measured data to determine accurately the topology of the subsurface reflectors. Migration is an inverse process in which the recorded waves are propagated back to their source by systematically solving the wave equation for each successive layer. There has been considerable study of stable algorithms for efficient solution to the wave equation, and seismic migration has been routinely used for interpreting seismic data for over two decades. Migration techniques range from simple finite-difference techniques to the more sophisticated frequency domain methods [5,7]. Regardless of the technique used, migration greatly facilitates accuracy in seismic interpretation and identification and in some cases is indispensable.

Seismic migration algorithms are computationally very intensive and require processing large amounts of data. *Stacking [5,7]* reduces the amount of data to be migrated and improves the signal to noise ratio, yet the tasks are remain computationally formidable. In addition, recent techniques for migration incorporate lateral velocity variations, and models for processing three-dimensional signals have also been proposed. These developments call for a dramatic increase in the computational requirements, often well above those provided by traditional computing architectures. At the same time, new architectures for high performance low cost computing have been developed, with exciting possibilities for handling computationally intensive problems [1]. Systematic analyses of seismic migration algorithms from the view point of such high performance systems has not yet been done. An objective of this paper is to reformulate solution of the wave equation to exploit the inherent space-time concurrency in the algorithms. We propose a few paradigms for exploiting this parallelism. For this purpose we review the spectrum of algorithms currently in use and propose methods in which they could be adapted for highly parallel signal processing architectures. A simple frequency domain parallel *phase shift* migration algorithm [2,3] was analyzed in the companion papers [1,9], and the performance indices for that algorithm discussed. In this paper we generalize those models for computation to incorporate alternate methods, and study their performance requirements. The algorithms will then be suitable for implementation on distributed memory and hypercube multiprocessors. The communication and computation tradeoffs, I/O bandwidth, and performance optimization criteria will also be examined.

In Section 2, we discuss the salient features of several migration techniques and propose parallel paradigms. Sections 3 and 4 will discuss the performance issues on shared memory and distributed memory multiprocessors. The rich interconnectivity of hypercube topologies will be used to advantage in reducing the communication overhead in distributed signal processing.

This research was supported in part by the Shell Development Co., Houston, TX, in part by the California Microelectronics Research Grant MICRO, and in part by the National Science Foundation.

2. Seismic Migration Algorithms

In this section we describe techniques, in use for migrating seismic record sections. For the purposes of clarity we outline the derivation of the general *wave extrapolation* equation. These techniques are clustered under the general name of *wave equation* migration methods. The theory is exact for laterally invariant velocities [2,5]. Assume the receiver coordinate is r, the source coordinate is s, the midpoint coordinate is $x = \frac{r+s}{2}$ and the offset coordinate is $h = \frac{r-s}{2}$. The variable z represents depth into the earth/ocean. The general wavefield can then be represented by $\psi(r, s, t, z_r, z_s)$ where z_r and z_s are the depth coordinates of the receiver and source respectively. The scalar wave equation is then

$$\frac{\partial^2\psi}{\partial t^2} = v^2\left[\frac{\partial^2\psi}{\partial r^2} + \frac{\partial^2\psi}{\partial z_r^2}\right] \quad , \quad \frac{\partial^2\psi}{\partial t^2} = v^2\left[\frac{\partial^2\psi}{\partial s^2} + \frac{\partial^2\psi}{\partial z_s^2}\right]$$

Fourier Transforming w.r.t r, s and t we have.

$$\frac{\partial^2\Psi}{\partial z_r^2} = -\frac{\omega}{v}^2\left[1-k_r\frac{v}{\omega}^2\right]\Psi$$

$$\frac{\partial^2\Psi}{\partial z_s^2} = -\frac{\omega}{v}^2\left[1-k_s\frac{v}{\omega}^2\right]\Psi$$

Here k_r and k_s are the spatial frequencies w.r.t r and s respectively. Ψ is the F.T. w.r.t all the variables (r, s, t, z_r, z_s).

These equations have two independent solutions, and choosing only the one which corresponds to the downward going waves, we have with $\kappa_r = \frac{k_r v}{\omega}$ and $\kappa_s = \frac{k_s v}{\omega}$ the following equations

$$\frac{\partial\Psi}{\partial z_r} = \frac{i\omega}{v}\left[1-\kappa_r^2\right]^{\frac{1}{2}}\Psi$$

$$\frac{\partial \Psi}{\partial z_s} = \frac{i\omega}{v}[1-\kappa_s^2]^{\frac{1}{2}}\Psi$$

With both the receiver and the source having common depth z we have

$$\frac{\partial \Psi}{\partial z} = \frac{\partial \Psi}{\partial z_r} + \frac{\partial \Psi}{\partial z_s}$$

$$\frac{\partial \Psi}{\partial z} = \frac{i\omega}{v}([1-\kappa_r^2]^{\frac{1}{2}} + [1-\kappa_s^2]^{\frac{1}{2}})\Psi$$

Then modifying the equation for zero-offset data and considering that $\kappa_r = (k_x + k_h)\frac{v}{2\omega}$ and $\kappa_s = (k_x - k_h)\frac{v}{2\omega}$ we can formulate a differential equation in $\Psi(k_x, \omega, z)$ as follows:

$$\frac{\partial \Psi}{\partial z} = \frac{2i\omega}{v}[1-(\frac{k_x v}{2\omega})^2]^{\frac{1}{2}}\Psi$$

This equation is the one-way wave equation for downward extrapolation of zero-offset data.

2.1 Migration in the (k_x, ω) domain.

Migration in the (k_x, ω) domain is called the *phase shift* algorithm [2,3]. We have from the extrapolation equation

$$\frac{\partial \Psi}{\partial z} = \frac{2i\omega}{v}[1-(\frac{k_x v}{2\omega})^2]^{\frac{1}{2}}\Psi$$

Given $\psi(x, t, z=0)$ we have to reconstruct $\psi(x, t=0, z)$ we obtain the stepping equation in z as follows

$$\Psi(k_x, \omega, \delta z) = \Psi(k_x, \omega, 0)\exp\{\frac{2i\omega}{v}[1-(\frac{k_x v}{2\omega})^2]^{\frac{1}{2}}\delta z\}$$

At this point we digress and describe in brief the architecture of the distributed multiprocessing system. There are a number of independent processors, each with its own memory, interconnected to other processors in the systems via a hypercube interconnection. Data is exchanged through messages, and the communication overhead is proportional to the length of the messages. Short messages are avoided to minimize the effect of communication set-up time for the communication protocol between nodes. There is a host processor, which oversees the operations of the multiprocessing system and can be used by the processors to execute commands which could be used by all the processors. Commands are defined as *global* if they need data elements from more than one processor for execution, and *local* if they can be executed by the processor itself on data elements within its own memory. A global command can be broken up into several local commands, each of which can be executed concurrently on the processors. If the operations are linear (as most matrix manipulations in migration are) a consistent result can be obtained by summing the results of local commands.

2.1.1 The Phase Shift Algorithm

The stratified-layer model of the earth [4] assumes that the earth is made up of horizontal layers extending downward in the depth coordinate z. Each layer of depth is characterized by a velocity c which is invariant in the horizontal coordinate x and is constant in z for that layer. Formulating the wave-equation for layer l, we have for $\psi(x, z, t)$ the wave-field

$$\psi_{xx}(x, z, t) + \psi_{zz}(x, z, t) - \frac{1}{c^2}\psi_{tt}(x, z, t) = 0$$

where ψ_{xx} is the second partial derivative w.r.t x, ψ_{zz} is second partial w.r.t z, and ψ_{tt} is the second partial w.r.t t. Fourier Transforming the equation with respect to x and t we have on rearranging

$$\Psi_{zz}(k_x, z, \omega) - k_x^2\Psi(k_x, z, \omega) + \frac{\omega^2}{c^2}\Psi(k_x, z, \omega) = 0$$

$$\Psi_{zz}(k_x, z, \omega) = -(\frac{\omega^2}{c^2} - k_x^2)\Psi(k_x, z, \omega)$$

This is a second order differential equation in z. If the initial condition is $\Psi(k_x, 0, \omega)$, then

$$\Psi(k_x, z_0, \omega) = \exp^{j\sqrt{\frac{\omega^2}{c^2} - k_x^2}z_0}\Psi(k_x, 0, \omega)$$

Therefore, given $\psi(x, 0, t)$ we evaluate $\Psi(k_x, 0, \omega)$, and then use the above equation to evaluate $\Psi(k_x, z_0, \omega)$. The required migration section is $\psi(x, z_0, 0)$, which we obtain from $\Psi(k_x, z_0, \omega)$ as follows,

$$\psi(x, z_0, 0) = \iint \Psi(k_x, z_0, \omega)e^{jk_x x}dk_x\, d\omega$$

Therefore, to obtain the migrated section, the algorithm first evaluates the two-dimensional Fourier Transform of the data, calculates the $\Psi(k_x, z_0, \omega)$ for required depth z_0, integrates over frequency and inverse transforms w.r.t. k_x.

In practice, we perform a discrete version of the above algorithm. $\psi(x, 0, t)$ is sampled at points $(x_j, 0, t_i)$ for $j = 1, 2, \cdots, N_x$ and $i = 1, 2, \ldots, N_t$ to give an N_t by N_x matrix, $\psi(x_j, 0, t_i)$. The two dimensional DFT w.r.t to x and t is then an N_t by N_x matrix, $\Psi(k_{xj}, 0, \omega_i)$, where j and i take the values mentioned earlier. In the discrete domain the solution to the migration equation, assuming the sampling theorem conditions are satisfied, is

$$\psi(x, z_0, t=0) = \sum_i \sum_j \Psi(k_{xj}, z_0, \omega_i)\, e^{jk_{xj}x_l}$$

The Sequential Phase Shift Algorithm.

The sequential algorithm proceeds in the following manner:

Step 1: $\Psi(k_{xj}, 0, w_i)$ is computed from $\psi(x_j, 0, t_i)$. $\Psi(k_{xj}, 0, \omega)$ is a N_t by N_x matrix. This step involves calculation of N_x N_t-point FFTs and N_t N_x-point FFTs.
Step 2: $\Psi(k_{xj}, 0, \omega_i)$ is multiplied by $E(j, 0, i) = e^{j\sqrt{\frac{w_i^2}{c^2} - k_{xj}^2}z_0}$ for all i and j.
Step 3: $\Psi(k_{xj}, z_0, \omega_i)$ from Step 2, is summed across ω_i.
Step 4: $\psi(x, z_0, 0)$ is then obtained by an IFFT w.r.t k_x.
Step 5: $z_0 = z_0 + \Delta z$
Step 6: if $z_0 \le L\Delta z$ then go to Step 2.
Stop.

For large N_x, N_t and L, the algorithm is computationally very expensive. We exploit the inherent parallelism as described in the next paragraph.

The Parallel Phase Shift Algorithm.

Let us assume we have N independent processors, each having its own CPU and local memory, capable of communicating with other processors, via messages, in a hypercube interconnected network. A central host processor performs tasks whose results may be used by all the independent processors.

We now describe the tasks required by each processor p to migrate data according to the phase shift algorithm. The processor is indexed by p which can take the integer values 1 through N. Since the processors have limited local memory, each processor can only process a row or a column at a time. There is an additional index to $E(j, l, i)$ to incorporate the varying velocity c_l in each new layer l. We also assume $N = N_x = N_t = L$ for simplicity of analysis.

Each processor p executes the following sequence:

Step 1: Receive from host, the N_x length vector, $\psi(x_j, 0, t_p)$ for $j = 1$ to N_x.
Compute its FFT w.r.t to x.
Send $\Psi(k_{xj}, 0, t_p)$ back to host
Step 2: Receive N_t length vector $\Psi(k_{xp}, 0, t_i)$ for $i = 1$ to N_t from host.
Compute its FFT, $\Psi(k_{xp}, 0, \omega_i)$.
Step 3: Multiply $\Psi(k_{xp}, 0, w_i)$ by $E(p, l, i)$ for all i.
Step 4: $\sum_i \Psi(k_{xp}, 0, \omega_i)$.
Step 5: Communicate with other processors,(in a parallel fashion) to sum over p
Invert the vector w.r.t k_x to obtain the next layer.
Step 6: Repeat steps 3-5 for the new layer, until L layers are migrated.
Stop

A multi-step parallel algorithm

A multi-step algorithm follows Steps 1, and Step 2 as above , but evaluates Steps 3 onwards as given below:

Step 3: For each $\Psi(k_{xp}, 0, \omega_i)$ compute the L by N_t matrix as follows:
Do for j = i to N_t
Do for l = 1 to L
$\Psi(k_{xp}, l, \omega_i) = \Psi(k_{xp}, l, \omega_i) \cdot E(p, l, i)$
Step 4: Sum over i to form $\sum_i \Psi(k_{xp}, l, \omega_i)$.

Step 5: Communicate with other processors to obtain the N_t length vector in k_x
Invert FFT w.r.t to k_x to obtain a migrated layer.
Stop

The hypercube topology, enables parallel communications in log$N$ steps, therefore, we can transmit from one node to the furthest node in atmost log$N$ communication steps. The multi-step algorithm is efficient in the sense that the parallelism is completely exploited. We explain this as follows. In the single step parallel algorithm , only one N_t point FFT need to be inverted to migrate the next layer, this implies that $N-1$ processors are idle. By processing L steps at a time, we can keep all the processors busy and balance the load equally. each processor migrates a different layer.

The parallel algorithm therefore reduces the $O(N^3)$ complexity of the sequential algorithm to an $O(N^2)$ algorithm, plus the communication costs. The multi-step algorithm keeps the load balanced across the processors, improving efficiency further.

With respect to conventional sequential computer the multi-step algorithm has another advantage. For each successive depth step, the sequential computer is penalized into accessing secondary storage, but our parallel algorithm, performs all the computations in the main memory for moderate memory penalty.

Practical considerations which limit this speedup are communication costs which are incurred in sending and receiving messages from the host, messages between the processors themselves across the spatial frequencies, I/O, memory limitation per processor resulting in fewer depth steps being processed in one pass and inefficient communication protocols for some commercial multiprocessors. A hypercube interconnection network between the processors can greatly speed up communications, and the broadcast facility from the host to the nodes decreases the communication costs incurred. In addition the communication time also depends on the size of the message being transmitted. In section 4 we examine the performance of this algorithm on two commercially available multiprocessors, the NCUBE and the Sequent [8,9].

2.2 Migration in the (x, ω) domain.

In the absence of horizontally varying velocity, the phase shift equation can be used to migrate seismic records efficiently. However, the equation is not valid when v varies with x and the square root has to be approximated with rational functions to solve the equation numerically. Different methods in use deal with truncated continued fraction expansion and are discussed in [2,3,4]. We will now examine an interesting stable two-step algorithm [4] that is a combination of the phase-shift algorithm and an interpolation scheme. Our interest in this scheme stems from two factors, firstly it arises from the exact phase shift scheme which is simple and stable, secondly, it generalizes to higher dimensions of space without significant increase in complexity. This cannot be said of most finite difference schemes which (using implicit methods) become expensive computationally when more than one dimension is considered. One of our objectives is to use parallel processing architectures for problems in higher dimensions, therefore, we will examine this two-step algorithm in some detail.

The method consists of two parts. In the first part, the phase shift equation is used to compute the next step, for a *number of velocities* which span a range of possible velocities. The next step then is interpolation, in which these reference wave-fields for different velocities are used to interpolate the exact wave-field for the next step in the migration. Though this two-part scheme is computationally expensive, it has very good dispersion properties, and we formulate a parallel algorithm in the following paragraph which minimizes the communication overhead. As will be explained in Section 3, any implementation which improves the computation to communication ratio will be preferred over others for the same algorithm. The interpolation part of the sequential program would contain routines for interpolating which would be defined globally. For efficient parallel implementation we break up this routine into *local* interpolation routines, and by trading communication in favor of computation the nodes spend a minimal time in communicating. The algorithm is structured as follows.

Step 1: Receive N_t length column from the host (FFT w.r.t k_x has been done).
Compute its N_t point FFT w.r.t, t , $\Psi(k_{xp}, z, \omega_i)$.
Step 2: For each of the velocities $v_1, v_2, \ldots, v_l$ march *locally* forward a depth step.
Sum over ω_i
Step 3: Communicate with other processors to march forward globally for l reference wavefields for the next layer. Compute IFFT w.r.t k_x of the l wavefields (in parallel).
Interpolate the definitive field $\Psi(x, z_0, \omega_i)$ for the next step.
Compute FFT of the definitive field, $\Psi(k_x, z_0, \omega_i)$.
Step 4: Repeat steps 2-3 for the next depth layer.
end

In a coarse-grained approach additional computation costs no extra time, if partitioned well. However, since communication costs come into the picture, we need to minimize the number of IFFTs. While methods to interpolate the definitive wavefield in the k_x domain would be of interest two problems arise; First a particular interpolation scheme may not be possible in frequency domain, and second, the velocity is explicitly a function of x only in the case of interpolation in (x, ω) domain. A compromise between the number of reference velocities and the simplicity of the interpolation scheme, would determine the actual efficiency of the implementation. In the next section we examine a *finite-difference* migration algorithm in the (x, w) domain, and observe that it lends itself easily for a distributed system of tridiagonal solvers.

2.3 Finite-difference in the (x, t) and (x, ω) domains.

In this section where we examine migration algorithms that migrate data in the time domain, the space component may be either in x or k_x Since we are trying to solve a two dimensional velocity problem, using one dimensional methods, the algorithms would not be exact for velocity which varies sharply with depth and space [6]. In this section we examine an algorithm in the (k_x, t) domain case when the velocity does not vary laterally. We will examine the parallel architectures for such implementation, and compare with the the other methods. These methods in the time domain approximate traditional methods for solving hyperbolic partial differential equations on grids. There is a wealth of literature on these methods, suitable for a finer-grained computation model than the one we are considering.

As described in [3,4], the phase shift equation which is exact for laterally invariant data cannot be used for laterally varying velocities. Instead the square-root expression is expanded in the form of a truncated series. The equation then becomes

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

This equation is solved by splitting it up into two separate equations applied for alternate steps. Cross multiplying and Fourier transforming with respect to k_x we have [3]

$$\frac{\partial \Psi}{\partial z} = \frac{i\omega}{v}\Psi$$

and

$$[1 + (\frac{v}{2\omega})^2 D_{xx}] D_z \Psi = \frac{iv}{2\omega} D_{xx} \Psi$$

D_x and D_z represent partial derivatives with respect to x and z . These equations are solved numerically with a set of algebraic equations involving a values of $\Psi(x, z, w)$ on a grid of points (j, n) referring to $(j\delta x, n\delta z)$, where Ψ is Ψ_{jn} .The equation then becomes

$$\Psi_{j,n+1} + (\alpha - i\beta)(\Psi_{j-1,n+1} - 2\Psi_{j,n+1} + \Psi_{j+1,n+1}) =$$
$$\Psi_{j,n} + (\alpha + j\beta)(\Psi_{j-1,n} - 2\Psi_{j,n} + \Psi_{j+1,n})$$

where $\alpha = (\frac{v}{2\omega\delta x})^2$ and $\beta = \frac{v\,\delta z}{4\omega\delta x^2}$.

This is a tridiagonal system of equations for j=1,2,3,....,N_x and must be solved for *all* N_t values of ω.

Our parallel model for finite-difference computation is one which solves a system of tridiagonal equations for ω_i on processor i . Let us examine the ways in which a tridiagonal system of equations may be solved on a processor. Two factors have to be considered; one is the limitation of memory per processor, and the second the parallelism in the computation for solving a single set.

Solving a tridiagonal set of equations is equivalent to multiplying the values on the grid by a template of four coefficient values, which moves across the grid, calculating the fourth unknown value on the grid using the coefficient values on the template. All the elements on the grid need not be stored and schemes exist which minimize the storage involved by ordering the update of values on the grid.

However, to continue with our discussion, finite-difference migration easily partitions into a coarse-grained paradigm for computation across the frequencies. Similar arguments can be made for finite difference in the (k_x , z , t^*) domain , where t^* is the transformed time in the "moving" coordinate reference axes [7]. Then each k_x is assigned a tridiagonal system of equations on a grid (z , t^*).

3. Performance Analyses

Our model for parallel processing consists of N independent processors, each with its own local memory, communicating with each other via messages. The interconnectivity is a hypercube topology, offering concurrency in communications. The cost of communication determines the efficacy of the parallel implementation. If the task is partitioned such that each independent processor takes t_{comp} for computation and the total time spent in non-overlapped communication is t_{comm} then the total time required for a parallel implementation is $t_{parr} = t_{comm} + t_{comp}$. The time required for a serial implementation of the algorithm would be Nt_{comp} , so an index of performance, I , would then be

$$I = \frac{t_{serial}}{t_{parr}} = \frac{Nt_{comp}}{t_{comm} + t_{comp}} = \frac{N}{1 + \frac{t_{comm}}{t_{comp}}}$$

The index of performance , I , compares the speed up of a parallel implementation with a sequential processor having an infinite memory. The actual speedup, then, in comparison with traditional sequential computers with secondary storage would be much higher. For example, a 512 processor NCUBE, would have an equivalent of 256 Mbytes of main memory, for parallel execution [1].

We now examine the indices of performance of the migration algorithms discussed to provide us with a measure of the performance expected from a distributed parallel implementation.

Migration in the (k_x , ω) domain.

The serial computation is about αA^2 time units, where A is the data array size ($A = N_x = N_t$) for a single step. The t_{parr} then is $\frac{\alpha A^2}{N}$ .The communication cost t_{comm} is $\beta A^2 \log N$ time units. α is a factor of proportionality which is a measure of the computation speed of each processor. β is proportional to the *effective* communication time per byte of data moved. The index of performance is then

$$I = \frac{N}{1 + \frac{N\beta \log N}{\alpha}}$$

The index depends on $\frac{\beta}{\alpha}$ (from [1], this ratio is about 10^{-2} to 10^{-3} for the phase shift algorithm). The memory requirements for the phase shift algorithm are minimal. If an L step scheme was used the memory required would be roughly $8LA$ bytes (we assume complex data).

Migration in the (x , ω) domain.

An interesting algorithm in this domain is the two-step phase shift plus interpolation algorithm discussed in section 2.2. The communication and the computation times are both directly proportional to the number of reference velocities l. If the number of reference velocities is less than the number of processors N then communication and computation may be overlapped. For this algorithm t_{comp} is approximately $\alpha l A^2$ and t_{comm} is $\beta l A^2 \log N$. Therefore the performance index is given by

$$I = \frac{N}{1 + \frac{N\beta \log N}{\alpha}}$$

Since the interpolation step adds some computation to the parallel system, with no additional communication , the index will be slightly better in practice than the phase shift algorithm. The phase shift algorithm with interpolation has the potential for being used when the data is in three dimensions, besides being unconditionally stable. The memory requirements are directly proportional to the number of reference velocities used and multistep algorithms used for the simple phase shift would be expensive.

Migration using Finite-Difference Schemes.

We considered a finite difference scheme in the (x , t) domain and the parallel algorithm consisted of partitioning the algorithm on the basis of frequencies. Each processor is assigned a tridiagonal system of equations. There are two communication steps, one when each node receives the row vector of data, and the second when the solution of the tridiagonal is completed and the solution summed across the nodes. For this algorithm t_{comp} = $\alpha_1 A + \alpha_2 A \log A$ since the solution of a tridiagonal system of size A takes $O(A)$ computations. The t_{comm} is again $\beta A^2 \log A$. We have considered a simple protocol in which the data is summed across the nodes in $\log A$ steps. The performance index is then

$$I = \frac{N}{1 + \frac{\beta A N}{\alpha_2}}$$

The data size A , affects the index of performance and we need more computation on each node. If more than one frequency is clustered on each node, then t_{comp} is increased and t_{comm} is reduced, giving the performance index

$$I = \frac{N}{1 + \frac{\beta N^2 \log N}{\alpha A \log A}}$$

The required performance index can then be computed for the best "cluster" value. in addition the memory requirements are moderate, since tridiagonal equations can be solved very efficiently in terms of storage.

Figures 1 and 2 illustrate the performance indices for the phase shift and the finite-difference methods respectively. They are plotted for A = 1024 , and for varying $\frac{\beta}{\alpha}$. The performance index of the phase shift algorithm monotonically increases with N, however the "knee" of the curve is soon reached. The onset of saturation is delayed by a small $\frac{\beta}{\alpha}$. The algorithm designer can use the curves to either estimate I or to determine the number of processors needed for the application. While the performance index is lower for smaller N, the efficiency is noticeably higher. The performance index for the finite difference methods increases to a limit and then decreases, the maximum being inversely proportional to $\frac{\beta}{\alpha}$. Therefore, to achieve peak performance $\frac{\beta}{\alpha}$ must be minimized, this may often depend on the communication protocols in use on the multiprocessor. So far we have not considered the constraint memory puts on the index of performance. The best cluster size can be determined for a given performance index, given A. But the limited memory per node may not allow such an implementation. Another factor to be considered is the I/O capabilities of the parallel processing machine. Pipelining I/O with computation would result in further enhancement of the performance. I/O and data path is a serious problem which must be considered before a dedicated hardware implementation of our proposed architectures can be considered. We have not considered broadcast communication facilities available on some commercial multiprocessors. In the next section we consider some performance indices on parallel computers.

4. Parallel Phase Shift Algorithm on Multiprocessors

In this section we study our implementation of the phase shift migration algorithm on multiprocessor computing systems. We consider two types of systems; shared memory systems and distributed memory parallel processor systems. The performance of our algorithm is studied on these two very different MIMD architectures.[8,9].

4.1 Shared Memory Multiprocessing System

Shared memory multiprocessors have a number of processors that share a common bus allowing them to access the system resources (I/O, memory, coprocessors). The program is partitioned into subtasks which are assigned to individual processors which should in principle execute them in parallel. One of the advantages of shared memory processors is that sharing data and variables is very easy, without expensive message based communications between processors, as only pointers need be exchanged. However, contention for common resources and the execution of critical sequential sections reduces the processor efficiency due to overhead incurred in queueing for resource allocation. Speed up is application dependent, but for many applications is linear in the number of processors for a while, with a saturation effect.

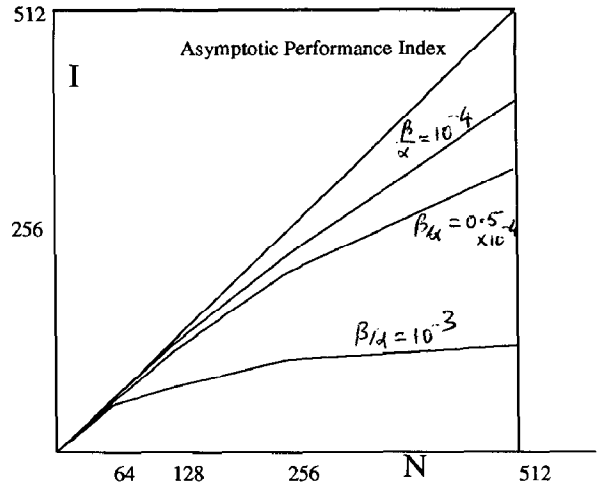
The **Sequent** multiprocessing system (with 12 processors) is a shared memory multiprocessor with an additional cache memory for each processor. The loops of the phase shift algorithm can be executed in parallel by the processors at the programmer's discretion. New processes are *forked* for concurrent execution. However, the shared and local variables used by the loops must be declared so that concurrent iterations do not violate the consistency of the program. Locks on resources need be set and freed by different processors, costing overhead.

Shared memory multiprocessors speed up execution of sequential programs mainly through **multitasking** facilities available, allowing a single application to be shared between closely cooperating processes. Speedup gained by multitasking is very application dependent. Contention arises when multiple processors access common resources, and in the Sequent this is minimized by having a private cache memory per processor. There is considerable overhead in creating, synchronizing and terminating multiple processes. The DYNIX operating system of the Sequent automatically balances load across the processes and schedules processes for optimal throughput.

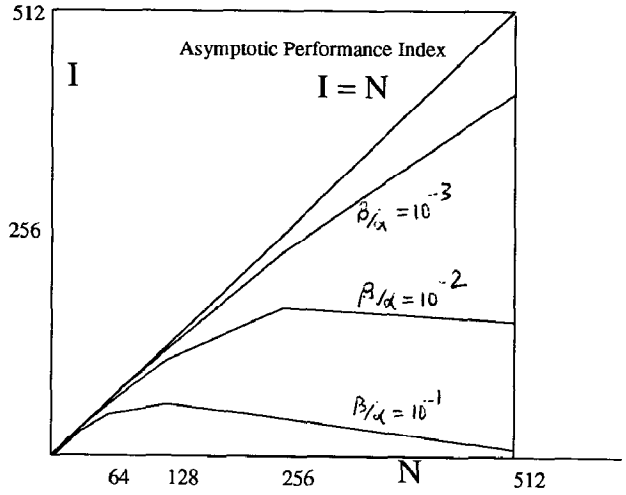
The initial speedup is almost linear in the number of processors but with an increase in the number of processors, the overhead associated with allocating and releasing resources penalizes performance and the speedup curve quickly saturates. The onset of saturation is delayed if each processor can execute a larger chunk of code in parallel and independently without conflict, memory management being a problem. The following table and Figure 3 summarize the performance of the algorithm for a relatively small data set migration (Case 1: $N_t = N_x = L = 128$, Case 2: $N_t = N_x = L = 32$).

Table 1

Phase Shift Migration on Shared Memory Multiprocessors					
Index	# of Processors	Speedup		% Utilization	
		Case 1	Case 2	Case 1	Case 2
1	1	1	1	99	76
2	2	1.96	1.5	96	67
3	4	3.4	3	91	55
4	6	4.4	3	86	53
5	8	4.8	3	82	50
6	10	5.5	3	79	49



Performance Curves for Phase Shift Methods
Figure 1



Performance Curves for Finite-Difference Methods
Figure 2

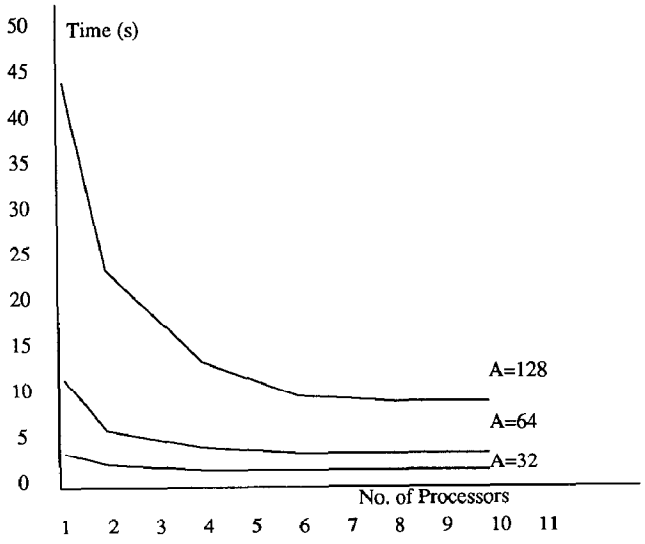


Figure 3

In our implementation we have not considered I/O in and out of the machine, and have not used any dynamic load balancing scheme to optimize performance.

4.2 Distributed Multiprocessor System.

A distributed computing system consists of an array of independent processors connected together through an interconnection network. Each node is an individual general processing system with its own private memory. The processors communicate with each other through a message communication protocol. An important difference between the distributed processor computing system and the traditional supercomputer is the need to divide both the algorithm and the data into smaller parts. This is very algorithm specific and may often be more difficult that using a shared memory multiprocessor having many programs running on a large shared memory. But if efficiently done, these highly concurrent processors can outperform vector supercomputers by over an order of magnitude.

Programming for the distributed application involves the following subtasks:

a). Partitioning the sequential algorithm, to distribute the computational load uniformly over the independent processors.
b). A communication protocol to enable efficient data and information transfer between the processors to yield results consistent with the sequential program.

Very often the communication overhead determines the performance to be expected. This is discussed in the following sections and also in the companion paper [6]. Further optimization of the performance is done by balancing the load dynamically, overlapping computation with communication, pipelining I/O and computation and by increasing the memory per processor.

The NCUBE/Ten Multiprocessor system interconnects 1024 32-bit processors each with 128 Kbyte private memory in a hypercube configuration [9]. Host processors are available to enable loose global control and synchronization. I/O channels lead directly to the processing nodes and the host through multiported memories. The communication protocol is a three-way handshake protocol, and data is transferred between nodes by DMA. The communication cost linearly increases with the size of the data chunk being transferred. Current models have enlarged memory available per node.

The parallel phase shift algorithm was implemented on the 64 processor model. Solution for data sizes for higher dimensions was implemented by breaking down the problem into smaller manageable tasks, and the performance was then projected for a higher number of nodes. For example it is the communications which need to be scaled up for a higher node model, the computation for the parallel task remains roughly the same. The modularity and the regularity of the node programs enables rapid porting onto higher order hypercubes.

4.3. Performance Analysis of Parallel Phase Shift Migration

In the implementation of parallel phase shift migration, each node works on a set of rows or columns at a time, and the communication routines handle the transfer of data and processing of intermediate results. The first communication algorithm we use is the *sequential communication protocol (SCP)*. The host is the center of the star of processors, each node communicates only with the host in receiving and sending data. The sequential algorithm thus penalizes the performance by not utilizing the parallelism inherent the highly interconnected hypercube configuration. However, it is conceptually simple, and its regularity will enable efficient implementation as a dedicated hardware processing system which is one objective of our study. The second communication algorithm, the *parallel communication protocol (PCP)*, overlaps communication between sets of nodes, reducing the *total* communication time t_{comm} . Both the protocols move the same amount of data and the times are proportional to A^2 where (A, A) is the size of the data array, though the constants of proportionality are an order of magnitude apart. The comunication cost is directly proportional to the bytes of data moved, and an array has A^2 elements. In addition using the hypercube topology PCP can sum across nodes in $LogN$ steps where N is the number of processors. It must be mentioned that the communication is dominated by the matrix transpose in the second step of the parallel algorithm.

Our basis of comparison is a hypothetical sequential machine (HSP) with infinite cache size, and the computation of the algorithm on this machine is the total computation time on the different parallel processors. Processor utilization of this machine is assumed to be 100 %. Therefore this hypothetical machine is many times more efficient than the usual sequential computer. For example, not using our function partitioning technique to compute L depth steps at a time, would penalize the traditional sequential computer, into taking L times longer than required for a single step. Likewise storing a large array columnwise can cause a page fault when computing across the rows on a large supercomputer (e.g. Convex). Another way of partitioning would be to assign different w to different nodes, this may facilitate I/O transfer (e.g. data from a geophone is sent directly to the node).

Table 2

Performance of Parallel Phase Shift Migration on NCUBE						
Index	# of Nodes	HSP (sec.)	SCP(sec)	PCP(sec)	I_s	I_p
1	64	252	9.05	6.1	27	41
2	128	252	7.2	4.2	35	60
3	256	252	6.3	3.36	40	75
4	512	252	5.6	2.76	45	91

The performance figures for the parallel phase shift algorithm are given in the Table 2 (For $N_t = N_x = 256$) and plotted in Figure 4 . If the computational task is partitioned such that each independent processor takes t_{comp} time for computation, and the total time spent in non-overlapped communication is t_{comm} , then the execution time on the parallel system is $t_{parr} = t_{comm} + t_{comp}$. The time required for a serial implementation would be Nt_{comp},so we can define an index of performance as

$$I = \frac{t_{serial}}{t_{parr}} = \frac{Nt_{comp}}{t_{comm} + t_{comp}}$$

An interesting observation is that the performance is relatively independent of the size of the data array being processed, given that the array is large enough to keep processors busy. Another observation is the strong dependence of the I on the communication protocol and the communication speed between nodes. The phase shift algorithm for one step takes roughly αA^2 time units and with N independent processors, t_{comp} is $\frac{\alpha A^2}{N}$ time units and t_{comm} is approximately $\beta A^2 \log N$ time units. Therefore I can be rewritten as

$$I = \frac{\alpha A^2}{\frac{\alpha A^2}{N} + \beta A^2 \log N}$$

β is a rough measure of the communication time for byte of data transferred, ideally β would be zero, leading to a performance index of N . Therefore we have;

$$I = \frac{N}{1 + \frac{N \beta \log N}{\alpha}}$$

and I is **independent** of the array size A to the first approximation.

We conclude that the communication speed coefficient β determines I. In addition, the performance improves with N as observed in Figure 4.

5. Summary and Future Work.

In this paper we have exploited the concurrency inherent in seismic migration algorithms to propose fast computing paradigms suitable for implementation on distributed parallel processors. Our analyses indicate that a coarse grained parallel implementation is well suited for seismic migration applications. We present some indices of performances for several important migration methods when implemented on a distributed

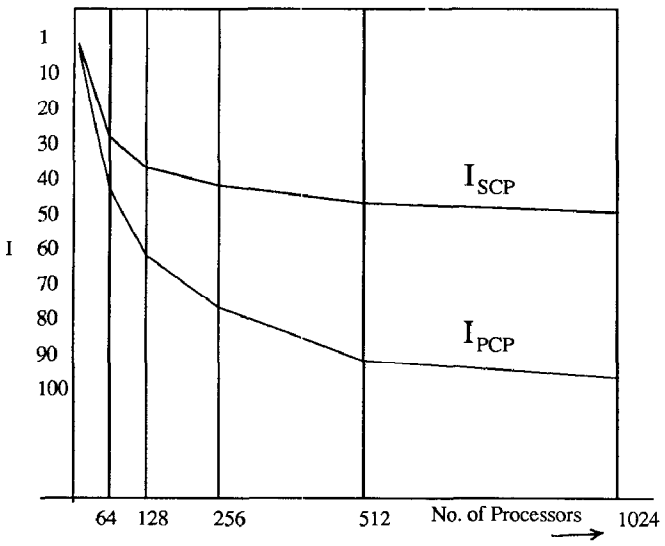


Figure 4

parallel signal processing architecture. A simple phase shift migration scheme was implemented on the NCUBE distributed multiprocessing system, and is described in [1], and illustrates the tradeoffs for a specific algorithm.

Analysis of I/O and data path requirements will be the next objective of our study. Efficient communication protocols for distributed multiprocessors lead to high indices of performance, and we are studying some schemes to improve speed up communication by overlapping processing and communication [8]. Load balancing and processor utilization have not been considered in our study and they need to be incorporated for enhancing the efficiency of implementation. The tradeoff between simplicity of implementation and speedup would prove to be useful in the design of dedicated signal processing hardware.

6. Selected References.

[1]. Madisetti, V.K. and D. G. Messerschmitt (1987), **Seismic Migration Algorithms Using the FFT Approach on the NCUBE Multiprocessor** *Proc. of IEEE-ICASSP 88, New York, 1988.*

[2] Gazdag, J. and P. Sguazero (1984), **Migration of seismic data,** *Proc. of IEEE, vol. 72, No. 10, October 1984.*

[3] Gazdag, J, and P. Sguazzero (1984), **Migration of seismic data by phase shift plus interpolation,** *Geophysics,* Vol.49, No. 2, 1984, pp 124-131

[4] Gazdag, J. (1978), **Wave equation migration with the phase shift method,** *Geophysics,* Vol. 43, 1978, pp1342-1351.

[5] Robinson, E. A. (1986), **Migration of Seismic Data by the WKBJ Approximation.** *Proc. of IEEE, Vol.74, March 1986, pp428-439.*

[6] Shimon Coen, *personal communication.*

[7] Claerbout, J.(1976), *Fundamentals of geophysical data processing,* McGraw Hill, New York. 1976.

[8] Madisetti, V., D. Messerschmitt (1987), **Efficient Message Communication Protocols for Distributed Multiprocessors.** *submitted for presentation.*

[9] Madisetti, V.K., D. G. Messerschmitt (1987), **Seismic Migration Algorithms on Multiprocessors,** *Proc. of IEEE International Conference on Acoustics, Speech and Signal Processing, New York, 1988.*

[10] Fox, G. and S.Otto, (1985), **Concurrent Computation and the Theory of Complex Systems,** SIAM Proc. of Conference on Hypercube Multiprocessors, Knoxville, TN 1986.

[11] Chun, J.H., and C. A. Jacewitz, (1981), **Fundamentals of Frequency Domain Migration,** *Geophysics,* Vol. 43, pp 1342-1351, No. 7.

[12] Schneider, W.A.(1976), **Integral Formulation For Migration in Two and Three Dimensions,** *Geophysics,* Vol. 43, pp 49-76, No. 1.

[13] **NCUBE Manual,** *NCUBE Corporation, Beaverton,* Oregon 1987.

[14] **Balance DYNIX Programmer's Manual.** *Sequent Computers Inc. 1986, Oregon.*

[15] Madisetti, V.K. and D. G. Messerschmitt, (1988), **Distributed Processing Paradigms: Definition, Analysis, Languages and Optimization,** *in preparation, 1988.*

[16] Madisetti, V., J. Walrand, and D. Messerschmitt,(1988), **Distributed Simulation of Discrete Event Systems** *under study.*