Stanford Exploration Project, Report 70, December 18, 1997, pages 1–80

Wave-equation algorithms on massively parallel computers

Biondo Biondi[1]

keywords: *algorithm, computing, Fourier transform, phase*

ABSTRACT

The wave-equation algorithms used in reflection seismology are based on the combination of the following three methods: Fourier domain methods, finite differencing, and Kirchhoff integrals. These basic methods can be efficiently implemented on a massively parallel computer using simple algorithms. The proposed algorithms are easily expressed in the new Fortran 90 language. I implemented and ran the Fortran 90 codes on the SEP CM and measured good performances.

INTRODUCTION

Research in reflection seismology is dependent, as in all computational sciences, on the computer tools available. Seismology is an "experimental" science, that is, the "theory" (new algorithms) must be checked against the results of the "experiments" (real data results). Therefore, the speed of the available computers determines the limits of the computational cost of the new methods that is practical to investigate. Massively parallel computers hold the promise of expanding these limits by increasing the computational power available to researchers by some order of magnitude. But, to actually keep this promise, massively parallel computers must show themselves to be capable of solving the numerical problems that are of interest to Geophysicists, and to do so without requiring a large effort in programming from the users.

Among the seismic algorithms that are most widely used, and most computationally intensive, are algorithms related to the solution of the wave equation. Wave-equation algorithms are fundamental to seismic data processing, as well as of data modeling. The wave equation is usually solved or by one of the following basic methods, or by a combination of them: Fourier methods, finite-difference schemes, and Kirchhoff integrals. In this paper, I describe an efficient implementation of each of these three basic algorithms. The algorithms that I present are very simple but they are useful for illustrating the ideas underlying the efficient implementation of wave-equation algorithms on a parallel computer. They achieve high efficiency because

[1]**email:** not available

they respect the two rules of parallel algorithms: low communications overhead and load balancing.

I implemented the example algorithms on the Connection Machine (Hillis, 1985) that SEP recently acquired. The algorithms were coded in Fortran 90 (Brainerd et al., 1990); I show some code fragments from my programs.

PARALLEL PROGRAMMING IN FORTRAN 90

Fortran 90 is a new Fortran standard that has been implemented on a few massively parallel computers and a few vector computers. The main advance of Fortran 90, with respect to Fortran 77, is that arrays are treated as first-class objects. Arrays, or sections of arrays, can be referenced in a single statement, and a whole set of intrinsic functions are defined for complex manipulations of array objects. The operations on the data expressed by the array statements, or array functions, are intrinsically parallel, and thus they can be efficiently compiled for execution on a parallel hardware. Using Fortran 90, the Geophysicist can express mathematical relations in a high-level language, while the compiler and the system software take care of parallelizing the execution of his program. However, the algorithms and the programs must still be designed carefully if good performance is to be obtained.

The first issue to be addressed when implementing algorithms on a massively parallel computer is minimizing the cost of interprocessor communications. Massively parallel computers have the memory distributed among many processors; the processors are linked together by a communication network, which may have varying topology (hypercube, 2-D mesh, etc ...). Because computations can be performed only when all the operands are local to a processor, interprocessor communications are needed when the required data do not reside in the memory of one processor. If the ratio between the time spent performing floating point operations and the time spent moving data among processors is too low, the performance of a parallel computer may be disappointing. To minimize the time spent in communication the user can determine an "optimal" mapping of the data into the memory of the processors. The implementation of Fortran 90 on the Connection Machine allows the user to specify the data layout with a simple compiler directive. Figure **??** shows an example of how seismic data can be mapped into processors memory. The layout shown is the one used for the Kirchhoff migration algorithm described in the final section of this paper. The traces in a common-offset section are stored locally in the memory of each processor, while the midpoint direction is parallel. Because of this choice of data mapping the generalized moveout operation that is in the innermost loop of Kirchhoff migration can be performed without inter-processor communications, and the algorithm performs very well.

The most of the algorithms need inter-processor communication even if the data have been correctly mapped. When communication is needed, it is important that a fast type of communication is used. On the Connection Machine, there are three

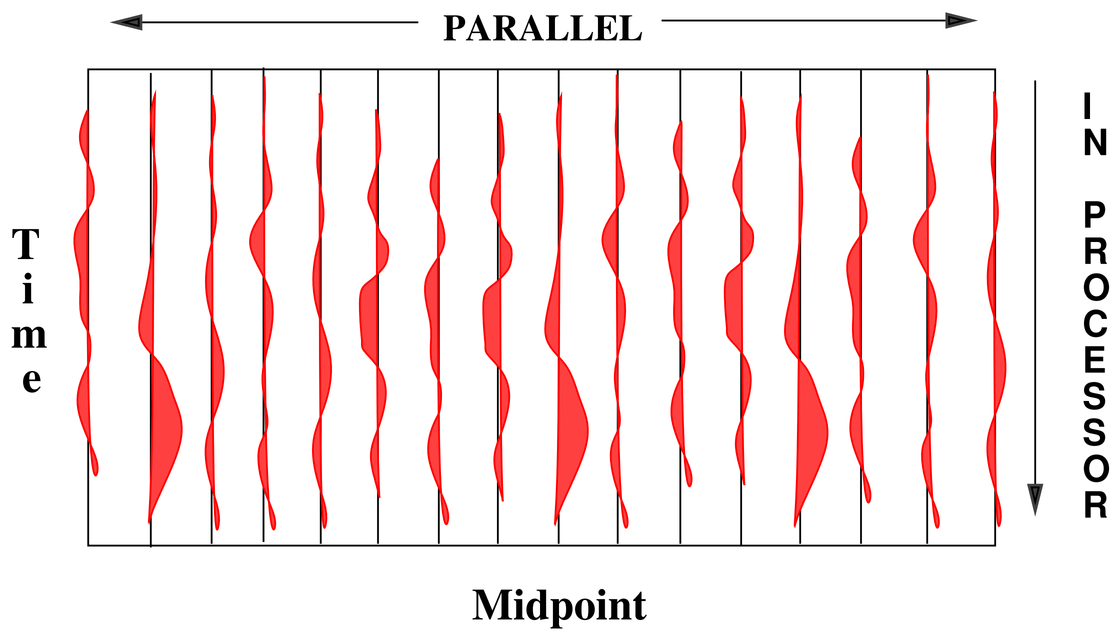

Figure 1: The "optimal" mapping of a common-offset section into the processors' memory for Kirchhoff migration. biondo-layout [NR]

types of communication between processors: nearest neighbor, global, and general communication. The nearest neighbor and the global communications are the fastest, and thus the most desirable to use. In Fortran 90, the communications are expressed at a high level as operations on the data arrays, or on elements of data arrays. The compiler takes care of translating these Fortran statements into the appropriate lower level communication functions. The type of communication that is performed for executing a specific Fortran 90 instruction depends on the data mapping, as well as on the instruction itself. Therefore, the user can control the type of communications performed during the execution of his program by selecting the "optimal" data layout and by using the "correct" array operations.

The second basic principle for writing efficient parallel programs is to keep the maximum number of processors busy at all times. The first way to avoid idle processors is to have a problem large enough compared to the size of the computer at hand. In general, massively parallel computers are not good at solving small problems. Recursion along one of the parallel axes can be a cause of severe load balancing problems. Performing special computations in the boundary regions of a finite-difference grid is another situation where processor time could be wasted, if the boundary conditions are not carefully designed. There is no general recipe for avoiding load unbalancing, but the most of the time the problem can be solved by changing the data mapping and/or modifying the algorithm. In extreme cases, the data can be redistributed among the processors to balance the computational load, at the expense of extra communications.

In the next section I begin my review of wave-equation algorithms with Fourier domain methods.

FOURIER DOMAIN ALGORITHMS

The Fourier domain algorithms are naturally parallel because the computations for the different frequencies (or wavenumbers) are usually independent. However, the execution of Fourier domain algorithms requires the transformation of the data by use of FFT's. Implementing efficiently FFT's on a parallel computer is difficult because FFTs are "global" algorithms, and thus they require a lot of data motions. There are many specialized algorithms for optimizing FFTs on different parallel architecture (Johnsson et al., 1989). Fortunately, many parallel computers has scientific library, of which FFTs are one of the main components (TMC, 1991). Geophysicists can thus avoid the troubles of coding efficient FFTs by themselves.

As an example of Fourier domain algorithms I discuss a prestack phase-shift migration (Gazdag, 1978). In a phase-shift migration the prestack data are first transformed into the frequency-wavenumber domain by a multidimensional FFT. Then, the transformed data are downward continued by multiplication with a complex exponential. Finally, the downward continued data are imaged at zero time and zero offset. The imaging at zero time and zero offset can be simply computed by summing the downward continued data for all frequencies, and all offset wavenumbers, without computing the inverse Fourier transform for all times and offsets.

The downward continuation is naturally parallel because the different wavenumber components of the data can be downward continued independently; they interact only when they are summed together for imaging the result at zero offset. This summation is easily coded in Fortran 90 by the intrinsic function `SUM`; the compiler takes care of issuing the correct instructions for the necessary interprocessor communications. Global communications, as the sum over one or more axes of an array, are usually very fast because they can use the whole bandwidth of the communication network linking the processors. Consequently, the time spent in communications is only a small percentage of the total run time (about 5-8 % in this particular case).

Not all the wavenumber components need to be downward continued, because some of them correspond to evanescent waves. The zeroing of the evanescent waves cannot be expressed with the usual serial `IF` statement, because all the wavenumbers are processed in parallel. Therefore the parallel conditional statement `WHERE` must be used for distinguishing between propagating and evanescent waves. A SIMD (Single Instruction Multiple Data) computer executes the two branches of a `WHERE` statement in sequence, one after the other, not in parallel. On the contrary, a MIMD computer (Multiple Instructions Multiple Data) could execute them in parallel. The sequential execution of the two branches of a `WHERE` statements is a potential disadvantage of SIMD computers, but it is not in this particular case, because the branch corresponding to evanescent waves simply sets to zero the data, and thus takes few CPU's cycles.

The kernel of phase-shift prestack migration is expressed in Fortran 90 by the following code fragment:

```
C
C loop over migrated time
C
        DO ITAU = 1,NTAU-1
C
C compute Vel*Vel*Ks*Ks and Vel*Vel*Kg*Kg
C
           VELSQ = VELTAU(ITAU)*VELTAU(ITAU)
           KSVELSQ = VELSQ*KS2SQ
           KGVELSQ = VELSQ*KG2SQ

           TAUSLICE = 0.

C
C loop over frequencies
C
           DO IW = 1,NW
                CMOMEGA = OMEGA(IW)

C
C downward propagate the non-evanescent waves
C
                WHERE((CMOMEGA .GT. KSVELSQ).AND.(CMOMEGA .GT. KGVELSQ))

                   KTAU = -DTAUD2*(SQRT(CMOMEGA-KSVELSQ)+SQRT(CMOMEGA-KGVELSQ))
                   DATA(IW,:,:) = CMPLX(COS(KTAU),SIN(KTAU))*DATA(IW,:,:)

C
C zero the evanescent waves
C
                ELSEWHERE
                    DATA(IW,:,:) = 0.
                END WHERE

C
C image at zero time
C
                TAUSLICE = TAUSLICE+DATA(IW,:,:)

          END DO

C
C image at zero offset
C
          MIGRES(ITAU+1,:) = SUM(TAUSLICE,DIM=1)

        END DO
```

Notice that in a phase-shift algorithm the different frequencies components could have also been processed in parallel, but, by serially looping along the frequency axis, I avoid the redundant evaluation, and storage, of some working arrays (KSVELSQ and KGVELSQ in my code).

Prestack Stolt migration (Clayton and Stolt, 1981) can be implemented similarly to phase shift; also for Stolt migration the wavenumber components can be processed in parallel. However, in an efficient implementation of Stolt migration, the frequency axis must be local to each processor for avoiding interprocessor communications during the Stolt transformation of the frequency axis. If the frequency axis is kept local, the Stolt transformation can be simply implemented with the usual interpolation algorithms used on serial computers.

FINITE-DIFFERENCE ALGORITHMS

Explicit finite-difference algorithms map well onto massively parallel computers. At each time step, or depth step, the values of the wavefield is computed by a linear combination of the values of the wavefield at the previous step in the neighboring points of the data grid. Furthermore, the regular data grid used in finite-difference algorithms can be mapped into an hypercube in such a way that the property of locality are preserved. Therefore, only nearest-neighbor communications are required between the processors of an hypercube for executing an explicit finite-difference scheme.

As an example of explicit finite-difference I present a simple subroutine that implements an explicit finite-difference algorithm for modeling acoustic waves. The code shows the subroutine for stepping in time of a simple 2-D second-oder in time and space finite-difference scheme. The values of the wavefield in in the neighboring points are accessed by shifting the data array by use of the Fortran 90 intrinsic function CSHIFT. The most of the neighboring values are stored in the memory of the same processors, and thus no communications is required for accessing them with a CSHIFT. The compiler takes care of issuing the instructions for the interprocessor communications necessary to access the few values that are not local.

```
      SUBROUTINE TIMESTEP (NT,NX,NY,NSOURCE,IXS,IYS,SOURCEV,
     1 WAVEITM1,WAVEIT,WAVEITP1,C1,C2,C3,C4,C5,C6)
      INTEGER NT,NX,NY,NSOURCE,IXS,IYS

      REAL SOURCEV(NSOURCE)
      REAL WAVEITM1(NX,NY),WAVEIT(NX,NY),WAVEITP1(NX,NY)
      REAL C1(NX,NY),C2(NX,NY),C3(NX,NY)
      REAL C4(NX,NY),C5(NX,NY),C6(NX,NY)

C
C time step loop
C

       DO IT=1,NT
C
C add point source at location (IXS,IYS)
```

```
C

               WAVEIT(IXS,IYS)=WAVEIT(IXS,IYS)+SOURCEV(IT)

C
C compute finite-difference star with spatially varying coefficients
C C1,C2,C3,C4,C5
C

               WAVEITP1 = C1*CSHIFT(WAVEIT,DIM=1,SHIFT=-1)
     1                 + C2*WAVEIT
     2                 + C3*CSHIFT(WAVEIT,DIM=1,SHIFT= 1)
     3                 + C4*CSHIFT(WAVEIT,DIM=2,SHIFT=-1)
     4                 + C5*CSHIFT(WAVEIT,DIM=2,SHIFT= 1)

C
C compute next time step
C
               WAVEITP1=WAVEITP1+C6*WAVEITM1

C
C update value for previous time step
C

               WAVEITM1=WAVEIT
               WAVEIT=WAVEITP1

       END DO

       RETURN
       END
```

The efficient treatment of boundary conditions can be a problem on SIMD machines, because the computations performed in the boundary regions may be different than the computations executed in the interior region. In this case, the computations in the boundary must be performed after the computations in the interior, and not in parallel . Often this problem can be solved by applying the same finite-difference star in the boundary region as in the interior region, but with different coefficients. Changing the coefficients makes the finite-difference equation solved on the boundary different than the equation solved in the interior, without requiring a different differencing star. In the example shown before the variability of the coefficients have been exploited for implementing free-surface boundary conditions at the top of the computational domain, and absorbing boundary conditions on the other three sides. Figure **??** shows a snapshot of the wavefield generated using this subroutine.

The efficient implementation of implicit finite-difference algorithms is not always as straightforward as explicit schemes, because implicit algorithms require the solution of a sparse system of linear equations. The linear systems can be solved by iterative algorithm like conjugate gradient (Nichols, 1991) The iterative solution are usually easy to parallelize because they require computations similar to the one required by explicit finite differences. On the contrary, the commonly used algorithms for direct solution of sparse linear systems are recursive. Important example in Geophysics is the solution of the tridiagonal system in finite-difference migration, that creates problems for vector computers as well as parallel computers. On a parallel computer the tridiagonal systems can be solved with methods similar to the ones used on vector computers, like cyclic reduction (Johnsson, 1987, Cole 1991). Or in some cases, as for example for finite-difference prestack migration, the particular structure of the problem can be exploited for devising a non-recursive parallel algorithm (Moorhead and Biondi, 1991).

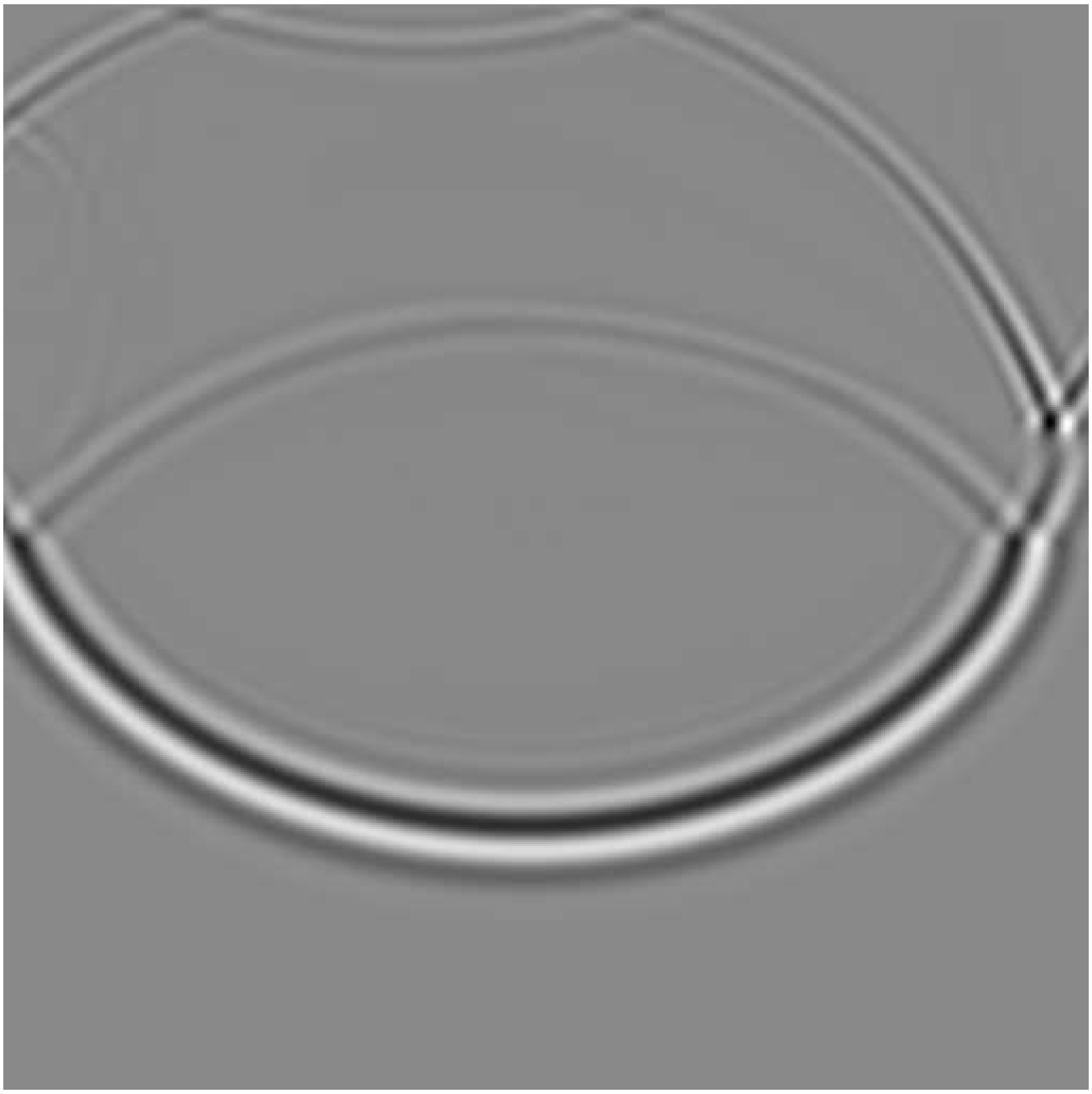

Figure 2: Snapshot of the wavefield captured from the CM FrameBuffer. Notice the absorbing boundary conditions on the sides and the free surface at the top. biondo-fdfig [NR]

KIRCHHOFF ALGORITHMS

Kirchhoff migration (Schneider, 1978) is applied to the data as a convolution with a space and time variant operator. The impulse response of Kirchhoff migration, although it may be very long, is continuos in space. By exploiting the spatial continuity of the impulse response, an efficient implementation of Kirchhoff migration on a massively parallel computer needs to use only fast nearest neighbor communications. The algorithm is very simple. The seismic traces are laid out local to each processor (Figure **??**); the input traces are passed around among processors while the output traces are kept fixed (it may be as well the contrary). At each step of this process, each input trace will be in the same processor of an output trace, and thus its contributions to the overlapping output trace can be added to the result by simple in-processor computations.

The following code fragments express in Fortran 90 a Kirchhoff prestack time-migration algorithm. In this example the data are assumed to be sorted in common-offset sections and the traveltimes are computed analytically assuming an RMS slowness function. Furthermore, for the sake of simplicity, I did not include the constant phase shift that should be applied to the results of Kirchoff migration.

First I show how the desired data layout is specified to the Fortran compiler.

```
C
C arrays declarations
C
      REAL DATAL(N1,N2)
      REAL DATAR(N1,N2)
      REAL MIGRES(NTAU,N2)

C
C arrays layout
C
C :SERIAL = local axis
C :NEWS = parallel axis
C

CMF$ LAYOUT DATAL(:SERIAL,:NEWS)
CMF$ LAYOUT DATAR(:SERIAL,:NEWS)
CMF$ LAYOUT MIGRES(:SERIAL,:NEWS)
```

Then, I show the main loop of the algorithm.

```
C
C initialize buffers for right-side half of operator
C
      DATAR=DATAL

C
C loop over operator lag
```

```
C
      DO IOPER=1,LOPER

C
C compute constants on the front-end
C
        DELMIDPOINT=(IOPER)*D2
        DELMH2=(DELMIDPOINT-HOFFSET)**2
        DELPH2=(DELMIDPOINT+HOFFSET)**2

C
C shift input data to the left and to the right
C
        DATAL=CSHIFT(DATAL,DIM=2,SHIFT=1)
        DATAR=CSHIFT(DATAR,DIM=2,SHIFT=-1)

C
C loop over output times
C
        DO ITAU=3,NTAU

C
C compute traveltimes on the front-end
C
           TIME=SLOWTAU(ITAU) *
     1          (SQRT(DEPTH2(ITAU)+(DELMH2)) +
     2          SQRT(DEPTH2(ITAU)+(DELPH2)))
           TIME=((TIME-O1)*INVD1)+1
           ITIME=INT(TIME)

C
C compute amplitudes (your favorite amplitude function can go here)
C
           AMP=1./TIME


           IF (ITIME .LE. (N1-3)) THEN

C
C compute interpolation samples and weights on the front-end,
C using the interpolator table TABINT
C
                ITAB=INT((TIME-ITIME)*LTABINT)+1

                INDEX1=ITIME
                INDEX2=ITIME+1
                INDEX3=ITIME-1
                INDEX4=ITIME+2
                INDEX5=ITIME-2
                INDEX6=ITIME+3

                INTER1=AMP*TABINT(0,ITAB)
                INTER2=AMP*TABINT(-1,ITAB)
                iNTER3=AMP*TABINT(1,ITAB)
```

```
               INTER4=AMP*TABINT(-2,ITAB)
               INTER5=AMP*TABINT(2,ITAB)
               INTER6=AMP*TABINT(-3,ITAB)

C
C compute contributions to the output and sum into output array
C

               MIGRES(ITAU,:)=MIGRES(ITAU,:)
     1              +  INTER1*DATAL(INDEX1,:)
     2              +  INTER1*DATAR(INDEX1,:)
     3              +  INTER2*DATAL(INDEX2,:)
     4              +  INTER2*DATAR(INDEX2,:)
     5              +  INTER3*DATAL(INDEX3,:)
     6              +  INTER3*DATAR(INDEX3,:)
     7              +  INTER4*DATAL(INDEX4,:)
     8              +  INTER4*DATAR(INDEX4,:)
     9              +  INTER5*DATAL(INDEX5,:)
     1              +  INTER5*DATAR(INDEX5,:)
     2              +  INTER6*DATAL(INDEX6,:)
     3              +  INTER6*DATAR(INDEX6,:)

          END IF


        END DO


      END DO
```

In this simple case each processor does exactly the same operation because the traveltimes are independent of the midpoint location. The same would be true if dip moveout were implemented instead of migration. On the contrary, in a depth migration algorithm the impulse response changes with the midpoint location, and thus each processor needs to perform a slightly different operation. More precisely, the indices of input time samples used in the summation differ from processor to processor. This requires the use of the indirect addressing capabilities of the processing hardware, and it usually leads to an increase in run time. Nevertheless, Kirchhoff depth migration can be efficiently run on a massively parallel computer (Van Trier, 1991). If the data are not evenly sampled, as in 3-D surveys, implementing a Kirchhoff migration algorithm would become even more complicated. However, also in this case, the basic idea of exploiting the spatial continuity of the impulse response can be used for deriving efficient algorithms.

TIMINGS ON THE SEP CONNECTION MACHINE

I implemented the algorithms presented in the previous sections on the SEP Connection Machine. The SEP CM has 8K serial bit processors, divided into two sections of 4K each (the largest CMs have 64K processors). For every 32 serial bit processors there is a Floating Point Unit (FPU), for a total of 256 FPUs (TMC, 1991). The most recent version of CM Fortran compiler (CMF 1.0) generates code executed directly by the FPUs, therefore the SEP CM is for the Fortran programmer a 256 processors computer, with 256 Kbytes of local memory per processor.

I timed the kernels of the algorithms on a 4K processors (128 FPUs) section of the machine and estimated the flop rates; all the algorithms presented run efficiently on the CM. When evaluating the performances of the tested programs it is advisable to take into account that they are just the kernels of simplified algorithms. They do not include I/O times (SEP has no DataVault, the CM's disk system) and other "real life" overheads. On the other hand, the CM at SEP has small memory; each processor has only 1/16th of the maximum possible memory (4 Mbytes per FPU). The small size of the processors' memory affects negatively the speed of the algorithms that require nearest neighbor communications (`CSHIFT`). The smaller the processors' memory is, the larger are the relative overheads of communications, because it becomes less favorable the ratio between the amount of data that is actually exchanged through the communication network and the amount of data that stays local in memory.

I run the Kirchhoff migration program on a constant offset section with 2048 traces and 1024 time samples per trace. I used a six point long interpolator, and a migration operator 200 traces wide in space. The total run time was 20.1 seconds, corresponding to 245 Mflop/s (Mflop = million of floating point instruction). Theoretically the speed of the same program on a 64K CM should scale up to 3.920 Mflop/s, but I did not tested it yet.

The finite-difference program takes .022 seconds for a each time step, when the dimensions of the computational domain are 512*512, this time corresponds to about 140 Mflop/s. For the Connection Machine there is also an experimental Fortran compiler, called the "Stencil Compiler", that generates faster code to perform convolutions of the kind required by finite-differences. When I compiled the five point differencing star with this new compiler each time steps took only .0084 seconds, equivalent to a computational rate of 373 Mflop/s (5.968 Mflop/s on 64K CM).

Finally, I timed the phase-shift migration. The data cube had 256 time samples, 256 shot locations, and 64 offsets. To perform the FFTs I used the complex-to-complex routine belonging to the Connection Machine Scientific Software Library (CMSSL). The transformation of a 128*256*64 data cube took 1.7 seconds on the SEP CM, correspondent to a flop rate of 129.5 Mflop/s. The downward continuation and imaging, not including the FFTs, took 47.9 seconds. In this case, the methodology used for converting the timing measures to flop rates must be more clearly explained because the inner loop of the phase shift migration includes square roots and trigonometric functions. The relative speed of these functions is hardware dependent; on the CM FPU the cycles taken to perform a square root are about about 8.4 times the cycles taken for a floating point operation (hardware square root). The trigonometric functions, that are computed in software, are equivalent to about 46 floating point operations. Using this conversion factors, the phase-shift migration runs at a flops rate of 330 Mflop/s (equivalent to 5.280 Mflop/s on a 64 CM).

CONCLUSIONS

The three algorithms presented in this paper, although simple, illustrate the basic ideas underlying the efficient implementation of wave-equation algorithms on massively parallel computers. The expression of these algorithms in Fortran 90 is straightforward, and very close to their mathematical formulations.

The measured performances on the SEP Connection Machine are very encouraging, and they demonstrate that Geophysical problems can be solved very efficiently on massively parallel computers.

ACKNOWLEDGMENTS

I would like to thanks Bill Moorhead of Thinking Machines Co., for sharing with me some of his wide knowledge on how to solve Geophysical problems with parallel computers; I also thanks Kirk Jordan, also at TMC, for having given me his finite-difference code.

REFERENCES

Biondi, B., Moorhead, W., 1991, Parallel implicit finite-difference prestack migration: Expanded Abstracts of the EAEG.

Brainerd, W., Goldberg, C.H., and Adams, J.C., 1990, Programmer's guide to Fortran 90: McGraw-Hill Book Co., New York.

Clayton, R.W. and Stolt, R.H., 1981, A Born-WKJB inversion method for acoustic reflection data: Geophysics, **46**, 1559-1576.

Cole, S., 1991, Porting a simple 2-D migration program to the Connection Machine: SEP–**70**.

Gazdag, J, 1978, Wave equation migration with the phase shift method: Geophysics, **43**, 1342-1351.

Johnsson, S.L., Krawitz, R.L., MacDonald, D., and Frye, R., 1989, A radix-2 FFT on the Connection Machine: Supercomputing 89.

Johnsson, S.L., 1987, Solving tridiagonal systems on ensemble architectures: SIAM Journal of Scientific and Statistical Computations, **8:3**, 354-392.

Hillis, D., 1985, The Connection Machine: MIT Press, Cambridge.

Nichols, D., 1991, 3-D depth migration by a Predictor-Corrector method: SEP–**70**.

Schneider, W.A., 1978, Integral formulation in two and three dimensions: Geophysics, **43**, 49-76.

Thinking Machines Corp, 1991, CMSSL from CM Fortran: Cambridge.

Thinking Machines Corp, 1990, Connection Machine Model CM-2 Technical Summary: Cambridge.

Van Trier, J., 1991, A massively parallel implementation of prestack Kirchhoff depth migration: Expanded Abstracts of the EAEG.