Stanford Exploration Project, Report 70, December 18, 1997, pages 1–28

Stanford Exploration Project, Report 70, December 18, 1997, pages 1–28

Porting a simple 2-D migration program to the Connection Machine

Steve Cole[1]

keywords: *algorithm, imaging*

ABSTRACT

As a learning exercise, I ported a simple 2-D implicit finite-difference poststack migration program to the Connection Machine. Getting the program up and running was not difficult, but figuring out how to get it to perform well was something of a challenge. Currently the program runs roughly 25 times faster than the equivalent well-vectorized program running on a Convex C-1. Since the algorithm itself is very familiar to most readers, those with no prior Connection Machine experience might find this review of my experiences a useful introduction.

INTRODUCTION

Figure **??** is a 2-D poststack implicit finite-difference depth migration program in the (ω, x) domain taken from *Imaging the Earth's Interior* (Claerbout, 1985). The program is short, the underlying algorithm is well-understood by geophysicists, and programs derived from this simple one are still widely used. For these reasons, I thought that this would be a good first program to bring to the Connection Machine. In addition to helping me learn about the machine, it is hopefully an example that will be of interest to others.

I had no experience with parallel machines prior to this, and still don't consider myself an expert, but I have gotten the program up and running, and it is performing (for problems large enough to take full advantage of the parallelism of the machine) roughly 25 times faster than a well-vectorized version of the program running on our Convex C-1, reaching a speed greater than 300 Megaflops/sec when using 4K CM processors (half of the 8K processor machine that is installed at SEP).

In this paper, I compare the code as I have it running on the two machines. The Connection Machine version of the code offers a brief introduction to Fortran 90, a new version of Fortran with parallel constructs, which we have been using on the CM. I also compare execution times, and discuss how I gradually improved the

[1]**email:** not available

```
#%  Migration in the (omega,x,z)-domain
real p(48,64), pi2, alpha, dt, dtau, dw
complex cp(48,64), cd(48), ce(48), cf(48), aa, a, b, c, cshift
integer ix, nx, iz, nz, iw, nw, it, nt, esize

nt= 64;   nz= nt;    nx= 48;      pi2= 2.*3.141592
dt= 1.;   dtau= 1.;  dw= pi2/(dt*nt);    nw= nt/2;
alpha = .25                                    #  alpha = v*v*dtau/(4*dx*dx)
do iz= 1, nz { do ix=1,nx { p(ix,iz) = 0.;    cp(ix,iz)=0. }}
do it= nt/3, nt, nt/4
        do ix= 1, 4                            # Broadened impulse source
                { cp(ix,it) = (5.-ix);     cp(ix,it+1) = (5.-ix) }
call rowcc( nx, nt, cp, +1., +1.)      # F.T. over time.
do iz= 1, nz {                         # iz and iw loops interchangeable
do iw= 2, nw {                         # iz and iw loops interchangeable
        aa = - alpha /( (0.,-1.)*(iw-1)*dw )
        a = -aa;      b = 1.+2.*aa;      c = -aa
# Compute right-hand side
        do ix= 2, nx-1
                cd(ix) = aa*cp(ix+1,iw) + (1.-2.*aa)*cp(ix,iw) + aa*cp(ix-1,iw)
        cd(1) = 0.;      cd(nx) = 0.
# Tridiagonal solver
        call ctris( nx, -a, a, b, c, -c, cd, cp(1,iw))
# Lens term
        cshift = cexp( cmplx( 0.,-(iw-1)*dw*dtau))
        do ix= 1, nx
                cp(ix,iw) = cp(ix,iw) * cshift
# Imaging
        do ix= 1, nx
                p(ix,iz) = p(ix,iz)+cp(ix,iw)   # p(t=0) = Sum P(omega)
        }}
esize=4
to history:     integer n1:nx, n2:nz, esize
call hclose()
call srite( 'out', p, nx*nz*4 )
return; end
```

Figure 1: A Ratfor program for migration in the (ω, x, z)-domain taken from Claerbout (1985).

performance on the CM.

FIRST ATTEMPT

Because the program is small, it took very little time to implement a first version on the Connection Machine. Here is a piece of my first program, equivalent to the two nested loops over frequency and depth in the program of Figure **??**:

```
  DO IZ=1,NZ
      A = SPREAD( -AA, DIM = 1, NCOPIES = NX )
      B = SPREAD( 1.+2.*AA, DIM = 1, NCOPIES = NX )
      C = SPREAD( -AA, DIM = 1, NCOPIES = NX )
C Compute right-hand side
      FORALL(J=1:NW) CD(1,J) = AA(J)*CP(2,J) + (1.-2.*AA(J))*CP(1,J)
      FORALL(J=1:NW) CD(NX,J) = AA(J)*CP(NX-1,J) + (1.-2.*AA(J))*CP(NX,J)
      FORALL(I=2:NX-1,J=1:NW) CD(I,J) = AA(J)*CP(I+1,J) +
     &   (1.-2.*AA(J))*CP(I,J) + AA(J)*CP(I-1,J)
C Tridiagonal solver
      CALL GEN_TRIDIAG_SOLVE( CD,A,B,C,ONE,TOL,IER)
C Lens term
      CP = CD * SHIFTS
C Imaging
      P(:,IZ) = SUM(REAL(CP), DIM = 2)
   ENDDO
```

Both programs have the four key ingredients of a 2-D implicit finite-difference migration: computing the right-hand side, solving the tridiagonal system, applying the thin lens term, and imaging.

A major difference between the two programs is that on the CM, there is no loop over frequencies. All frequencies are solved in parallel. Each CM processor sets up and solves the tridiagonal system of equations for one frequency. This mapping of the problem onto the processors of the machine is attractive because three of the four components in the algorithm (all except imaging) will then require no inter-processor communication, which should be avoided where possible on the Connection Machine. Figure **??** illustrates this mapping of the problem onto the hardware of the Connection Machine.

For best performance, we need to have enough frequencies to keep all processors busy. On a 4K (single bit) processor machine, there are 128 floating point processors. For hardware reasons, we need to have at least 4 times 128 or 512 frequencies to achieve best performance.

An alternative mapping of the problem is possible. We could make both the frequency and x axes parallel. This is a good thing to do for small problems, where we do not have enough frequencies to keep all processors busy with a (parallel, serial) ordering. A drawback to a (parallel, parallel) ordering is that all parts of the program

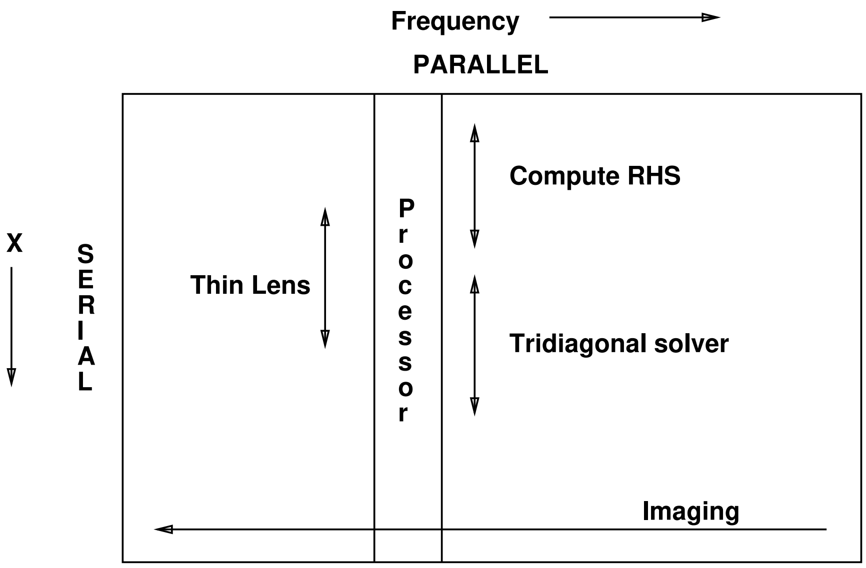

Figure 2: One mapping of the migration problem onto the Connection Machine. The downward-continued data for a given frequency and all x are stored in a single processor. Computing the right-hand side of the tridiagonal system, solving, and applying the thin lens term are fast – no inter-processor communication is required. Imaging requires a sum over all frequencies and is slower. steve1-ordering [NR]

will then require inter-processor communication. So if there are enough frequencies, it seems that the (parallel, serial) ordering is the way to go.

The mapping is specified by LAYOUT compiler directives. Here are the array declarations for the Fortran 90 code fragment shown above:

```
REAL P( NX,NZ )
COMPLEX CP( NX,NW )
COMPLEX A( NX,NW )
COMPLEX B( NX,NW )
COMPLEX C( NX,NW )
COMPLEX CD( NX,NW )
COMPLEX AA( NW )
COMPLEX SHIFTS( NX,NW )
```

and here are the associated layout directives:

```
CMF$LAYOUT P(:SERIAL,:NEWS)
CMF$LAYOUT CP(:SERIAL,:NEWS)
CMF$LAYOUT A(:SERIAL,:NEWS)
CMF$LAYOUT B(:SERIAL,:NEWS)
CMF$LAYOUT C(:SERIAL,:NEWS)
CMF$LAYOUT CD(:SERIAL,:NEWS)
CMF$LAYOUT SHIFTS(:SERIAL,:NEWS)
CMF$LAYOUT AA(:NEWS)
```

Changing all instances of `SERIAL` in the layout directives to `NEWS` would effect a (parallel, parallel) ordering.

The tridiagonal solver routine `GEN_TRIDIAG_SOLVE` solves (in this case) `NW` tridiagonal systems, each of length `NX`, in parallel. We supply `NW` copies of the three diagonal matrices. The SPREAD intrinsic function creates the necessary copies. The routine `GEN_TRIDIAG_SOLVE` is a parallel tridiagonal solver from the Connection Machine Scientific Software Library (CMSSL). On the Convex, I did not use the routine `ctris` shown in the Ratfor code example, but rather a well-vectorized tridiagonal solver from the Convex VECLIB vector processing library. This insures as fair a comparison between the two machines as possible.

Performance

The following tables describe the performance of the algorithm on problems of varying sizes. The elapsed and Connection Machine time and the speed (in Megaflops/sec) are listed for the loop over depths, for each of the key components of that loop, and for the program as a whole (time only). The speeds are computed by counting the number of floating point operations performed in each part of the code, then dividing by the time taken. Probably the most useful number is the speed of the loop, which is printed in boldface. Also shown is the cpu time on our Convex C-1 for the equivalent well-vectorized problem. `NX` is the number of points on the x axis, `NW` is the number of frequencies.

NX=64 NW=64			
	Elapsed Time (sec)	CM Time (sec)	CM Mflops/sec
Compute RHS	1.34	1.34	4.3
Tridiagonal solver	0.278	0.278	80
Lens term	0.052	0.052	30
Imaging	0.94	0.94	0.6
Loop total	3.29	3.29	**9**
Program total	6.23	5.01	
Convex cpu time (sec)	8.2		

NX=128 NW=128			
	Elapsed Time (sec)	CM Time (sec)	CM Mflops/sec
Compute RHS	5.19	5.12	9
Tridiagonal solver	1.44	1.11	162
Lens term	0.206	0.206	61
Imaging	3.98	3.98	1.1
Loop total	13.4	12.7	**19**
Program total	23	19	
Convex cpu time (sec)	52		

NX=256 NW=256			
	Elapsed Time (sec)	CM Time (sec)	CM Mflops/sec
Compute RHS	18.6	18.6	20
Tridiagonal solver	4.45	4.45	326
Lens term	0.819	0.819	123
Imaging	13.7	13.7	2.5
Loop total	47.4	47.0	**42**
Program total	71.5	69.6	
Convex cpu time (sec)	504 (8:24)		

NX=128 NW=1024			
	Elapsed Time (sec)	CM Time (sec)	CM Mflops/sec
Compute RHS	63.9	63.9	46
Tridiagonal solver	18.0	17.8	648
Lens term	3.27	3.27	246
Imaging	42.4	42.3	6.3
Loop total	158	158	**98**
Program total	239	235	
Convex cpu time (sec)	3284 (54:45)		

Not surprisingly, the program performs best when the number of frequencies is large, and we are able to keep all the processors busy. For the 1024x128 problem, the CM is roughly 14 time faster than the Convex.

A `(:NEWS,:NEWS)` version of the program is marginally faster on smaller problems because it uses more of the available processors. It is slower on the 1024x128 problem because it requires a lot of inter-processor communication that the `(:SERIAL,:NEWS)` version avoids. For a problem such as this, where I know that most inter-processor communication takes place along one axis, Fortran 90 allows me to "weight" the axes based on the amount of communication expected along them. Giving a weight of five to the x axis gives speeds of 18, 32, 45, and 74 Mflops/sec for the depth loops on the four problems presented above.

The tridiagonal solver performs quite well. The poorest performer is imaging. This is not too surprising, since imaging is the one place in the code where the frequencies are not dealt with independently. Inter-processor communication must take place in the imaging step, in order to sum the data from different frequencies together. This communication is the cause of the poor performance.

The speed of 98 Mflops/sec on the 1024x128 problem, though 14 times faster than the equivalent Convex program, was somewhat less than I had hoped to obtain. I tried various ways to speed up the program. These modifications are described in the following section.

IMPROVEMENTS

After a fair amount of work, I have managed to improve the program to where it is now roughly 25 times faster than the Convex versus 14 times originally.

Here is what the code looks like now:

```
   DO IZ=1,NW
      A = ACOPY
      B = BCOPY
      C = CCOPY
C Compute right-hand side
      CD = RA*CSHIFT(CMCQ,1,-1) + RB*CMCQ + RC*CSHIFT(CMCQ,1,1)
C Tridiagonal solver
      CALL GEN_TRIDIAG_SOLVE( CD,A,B,C,ONE,TOL,IER)
C Lens term
      CP = CD * SHIFTS
C Imaging
      P(:,1) = SUM(REAL(CP), DIM = 2)
      P = CSHIFT(P,2,1)
   ENDDO
```

In this section, I describe the modifications that I made, and show the improvement obtained by each. I show the timings at each step for the 128x1024 problem.

First, notice in the original CM code that I re-compute the diagonal matrices for the tridiagonal solver at each depth step. The reason I did this was because the solver overwrites these matrices. A less costly alternative to re-computing is to save a copy of the matrices and restore from this saved copy each time. These copies are the matrices `ACOPY`, `BCOPY`, and `CCOPY` in the new program. Here is the performance after this modification:

NX=128 NW=1024			
	Elapsed Time (sec)	CM Time (sec)	CM Mflops/sec
Compute RHS	63.9	63.9	46
Tridiagonal solver	17.8	17.8	648
Lens term	3.27	3.27	246
Imaging	42.3	42.3	6.4
Loop total	132	132	**117**
Program total	212	209	

The improvement is all in the "Loop total", since the computation of the diagonal matrices was not included in any of the parts of the loop timed individually.

The computation of the right-hand side in the old version, using `FORALL` statements, was not particularly efficient or easy to read. Fortran 90 provides functions `CSHIFT` and `EOSHIFT` to shift an array along a particular axis. `CSHIFT` performs a circular shift (with wraparound) while `EOSHIFT` does an end-off shift. `EOSHIFT` is what we want. It gives, in effect, zero-value boundary conditions. After trying it I noticed the following improvement:

NX=128 NW=1024			
	Elapsed Time (sec)	CM Time (sec)	CM Mflops/sec
Compute RHS	32.1	32.1	**92**
Tridiagonal solver	17.8	17.8	648
Lens term	3.27	3.27	246
Imaging	42.3	42.3	6.4
Loop total	101	101	**154**
Program total	180	177	

Then, when Biondo Biondi suggested that `CSHIFT` is faster than `EOSHIFT` in the current CM compiler, I used it instead and noticed an even larger improvement:

NX=128 NW=1024			
	Elapsed Time (sec)	CM Time (sec)	CM Mflops/sec
Compute RHS	10.8	10.8	**274**
Tridiagonal solver	17.8	17.8	648
Lens term	3.27	3.27	246
Imaging	42.2	42.2	6.4
Loop total	79.3	79.3	**196**
Program total	158	156	

Using `CSHIFT` gives circular boundary conditions. But I eliminated the unwanted wraparound in the shift by placing zeroes at the ends of the RA and RC right-hand side tridiagonal coefficient matrices, so the effect is the same as using `EOSHIFT`.

It may be that this code is still not optimal. Generally speaking `EOSHIFT` and `CSHIFT` operations should be used only on parallel axes. When used in the serial direction, as they are here, they may not be as efficient as a plain Fortran loop. For the problems tried thus far, however, I have found this method to be the fastest.

Finally the speed of imaging was improved by two more modifications suggested by Biondo. First, I had the imaging step always sum into the z=1 level of the output image rather than summing into the correct place. The fact that the destination of the sum is a constant location helps the compiler to optimize inter-processor communication. I then perform a `CSHIFT` to cycle the depth levels around to make sure that the z=1 level is free for the next imaging step. This operation is summarized in Figure **??**.

Along with this modification, I changed the size of one of the axes of the output image from the number of depths to the number of frequencies. This change forces the compiler to align the output image with the other arrays whose size depends

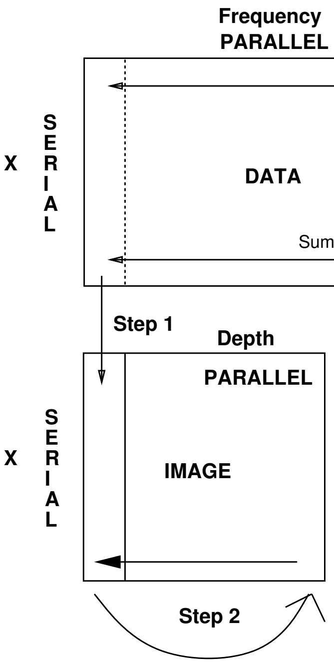

Figure 3: The speed of imaging is improved if, instead of summing into the correct depth level, we always sum into the first level. A circular shift of the image after each imaging step insures that the final image is ordered correctly. steve1-newimaging [NR]

on the number of frequencies (most of the arrays in the program). The resulting performance:

NX=128 NW=1024			
	Elapsed Time (sec)	CM Time (sec)	CM Mflops/sec
Compute RHS	10.8	10.8	274
Tridiagonal solver	17.8	17.8	648
Lens term	3.28	3.28	246
Imaging	13.0	13.0	**21**
Loop total	50	50	**311**
Program total	129	127	

The code is now running at more than 300 Mflops/sec, and is roughly 25 times faster than a well-vectorized Convex version. The imaging step is by far the slowest, and accounts for more than one fourth of the time spent in the depth loop.

With these fixes, there is now essentially no difference, for large problems, between the speed of the `(:SERIAL,:NEWS)` version discussed here and a `(:NEWS,:NEWS)` version where the axes are weighted to favor communication along the x axis (with a weighting factor of five). With this weight, the `(:NEWS,:NEWS)` achieves the same 311

Mflops/sec on the 1024x128 problem. On the 256x256 problem (as well as smaller problems) it is slightly faster. It runs at 158 Mflops/sec for the 256x256 problem, compared to 136 Mflops/sec for the `(:SERIAL,:NEWS)` version. On the smallest problem studied here, 64x64, the `(:NEWS,:NEWS)` version runs at 52 Mflops/sec, compared with 33 Mflops/sec for the `(:SERIAL,:NEWS)` program.

It may seem puzzling at first that the performance is poorer for smaller problems when using a `(:NEWS,:NEWS)` scheme. For a 64x64 problem, the product of `NX` and `NW` is 4096, far more than the number of available processors. Surely all the processors are being used, unlike with a `(:SERIAL,:NEWS)` ordering, where only 64 processors would be used. If so, why then are the speeds so similar? The answer is that even though all processors are being used in the `(:NEWS,:NEWS)` case, much time is being spent on inter-processor communication. In the 256x256 case, since there are 128 floating point processors, each one of these will contains 512 data points. Because of the axis weighting, these will likely be adjacent points in x for some particular frequency. The effect is much like a `(:SERIAL,:NEWS)` ordering, so the performance is similar. With a 64x64 problem, there are only 32 points per physical processor. The smaller problem is "spread out" over all the processors. This means that, compared to the larger problem, a greater fraction of the time is spent on inter-processor communication. Thus the resulting speed is slower, as is the speed we would get if we packed all the points into fewer processors in a `(:SERIAL,:NEWS)` ordering. For small problems, you just can't win.

My conclusion is that for this problem, a `(:NEWS,:NEWS)` specification with the proper weight on the x axis is the best general layout. It performs as well as a `(:SERIAL,:NEWS)` ordering for large problems, and slightly better for smaller problems.

My emphasis on the problem with 1024 frequencies and only 128 points in x may seem inappropriate. I would guess that a typical migration has as many or more points in x than ω. My reasons for choosing it had to do with memory limitations I have experienced on the CM, some of which are still mysterious. I have found myself limited to a total problem size of no more than 1024x128 or 128K points (though in theory the current program should have room for around 1024x1024 points on our CM). Within this limitation, the best speeds are obtained if we have more points along the frequency axis, since it entails less inter-processor communication. I believe that, given enough memory, speeds such as what I reported should be obtainable for 1024 frequencies and x axes in excess of 1024 points. The key is just the available memory. So I chose a somewhat skewed example in order to report what I think are speeds that can realistically be obtained on more reasonably sized problems, if only the memory is there.

CONCLUSIONS

It was relatively easy to get a 2-D finite difference migration program running on the Connection Machine. A speed significantly greater than that of a well-vectorized Convex algorithm was obtained immediately. Going from this initial success to the "expected" level of performance was more difficult and time-consuming. Careful consideration of how a problem maps onto the Connection Machine hardware is required in order to get the best performance from the machine.

Imaging is the slow step in the migration because of the extensive inter-processor communication required. Perhaps more work will discover a way to speed it up.

I hope that the lessons I learned in this exercise will help others to understand better the process of porting a program to the Connection Machine, and give a rough idea of the work needed to bring a particular algorithm "up to speed". Hopefully some of these lessons apply directly to finite-difference migration problems that others might attempt to port to the CM.

ACKNOWLEDGMENTS

Thanks to Biondo Biondi for help with the CM, especially for the suggestions on how to speed up the algorithm.

REFERENCES

Claerbout, J.F., 1985, Imaging the Earth's Interior: Blackwell Scientific Publications.