A program for vectorized finite-difference migration

Stewart A. Levin

ABSTRACT

In SEP-38 and SEP-41 I described a method for finite-difference migration on a parallel computer. Unfortunately, parallel computers do not abound yet. But pipelined vector computers do. Here I show by example that my parallel ideas also vectorize. I provide a Fortran program for 15 ° time-domain finite-difference migration that

1) vectorizes every inner loop,
2) handles both seismic data and migration velocities in natural trace ordering,
3) is short, simple and clear, and
4) easily converts to higher order finite-differences.

This program is the fastest and most sensible adaptation of 15 ° time-domain migration to a large, vector computer I have ever seen and I predict that it will soon be used routinely within the geophysical industry.

INTRODUCTION

In a short note on reverse-time migration (Levin, 1984a) I noted that on the conventional 15 ° migration grid reverse-time migration is most directly implemented by filling in successive planes parallel to the diagonal, migrated image. In follow-up SEP articles (Levin, 1984b,c) I showed that this reverse-time ordering is well suited to parallel computation and discussed practical details and limitations.

In 1985 the Stanford Exploration Project purchased a Convex C-1 vector computer (Claerbout, et al., 1985). A vector computer gains its speed through pipelining, a specialized form of parallelism. Because of this specialization, it is not necessary that a good procedure on a parallel computer is also good on a vector computer. Fortunately in finite-difference migration I find my parallel methods do work well as vector methods. In the appendix I demonstrate this with a listing of a short program I wrote for 15 ° finite-difference migration.

DISSECTING THE PROGRAM

The payoff of this exercise is shown in the comments inserted by the Convex Fortran compiler: every single inner loop in the program vectorized. On the Convex this $(t+\tau)$-outer program migrated a 256 trace, 1024 time point section in four and a quarter minutes of CPU time. A well-coded conventional τ-outer 15 ° migration took over 21 minutes to migrate the same section. This is a factor of five slower. The time lost was spent solving tridiagonal equations recursively by the method used in subroutine "triply" in the appendix. As shown in Levin, 1984c, there are less recursive alternatives to this method but in practice they run even slower than the efficient, though scalar, recursive method I've used.

A second distinction of this vectorized program is that it treats both the seismic data and the seismic velocities in natural trace ordering. Conventional finite-difference migration wants these grids transposed. Keeping them in trace order slows the conventional migration down because of noncontiguous memory access. Thus I also avoid the generally small overhead of transposing the grids. The drawback is that my program has a much higher storage requirement, requiring about $9nt \times nx$ locations compared to about $9nx$ for conventional migration. This is not a problem on our Convex nor on most other large vector computers.

The vectorized 15 ° code is just as short and simple as a conventional code. After all, they both do exactly the same computations, just in a different order. And with the aid of a few macro preprocessor statements to suppress "vectorized" subscripts, my program is just as clear.

Lastly, as discussed in Levin, 1984a, the parallel-vector approach extends easily to higher order finite differences. One needs to increase the amount of padding on the input data and add a few more diagonal planes to the storage requirements.

SUMMARY

Truly fast finite-difference migration is a reality on large vector computers. The program organization outlined in the references and illustrated in the appendix is an excellent way to achieve it.

REFERENCES

Claerbout, J., Dellinger, J., Harlan, B., Levin, S., Ottolini, R., and Sword, C., 1985, The Convex C-1 SEP's financial event of the decade: SEP-44, 175-182.

Levin, S.A., 1984a, Principle of reverse-time migration: Geophysics, 49, 581-583.

Levin, S.A., 1984b, Parallel space-time migration: SEP-38, 207-214.

Levin, S.A., 1984c, Footnote to parallel x-t migration: SEP-41, 207-216.

APPENDIX A. MIGRATION PROGRAM LISTING

The following pages list my vectorized 15 ° finite-difference migration program **parmig**. Notice that the input, output and velocity data are in processed in normal, i.e. untransposed, trace order. I've split the usual tridiagonal solver into two parts, **trifac** and **triply**, moving the former outside the main loop to avoid needless duplicate computations. The Fortran compiler would do the same code movement if I expanded these inside the main loop of **parmig**. Also, I've made liberal use of macro preprocessor statements to hide unimportant subscripts and improve readability.

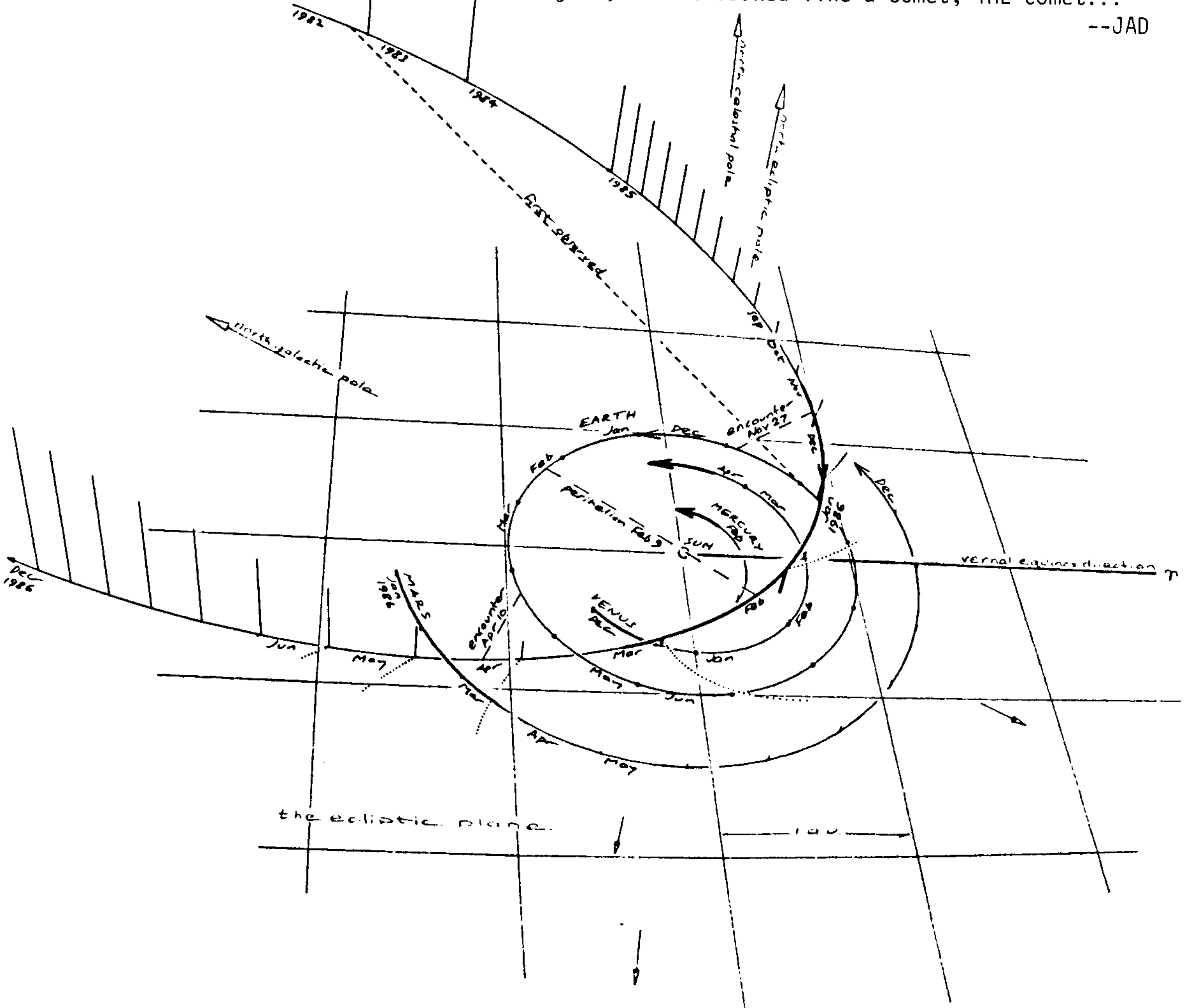
```
      subroutine parmig(nt,nx,infd,outfd,velfd,ntpl,p,alp,a,b,c,rtl,rt2,dt,dx,beta)
      integer nt,nx,ntpl,ix,it,itau,infd,outfd,velfd
      real p(ntpl,nx,3),alp(nt,nx),a(nt,nx),b(nt,nx),c(nt,nx)
      real rtl(nt,nx),rt2(nt,nx),sc,beta,dt,dx
      integer reed,rite,icnt,itemp,iprev,inow,inext
C
      do ix=1,nx
          icnt=reed(infd,p(1,ix,1),nt*4)
          p(ntpl,ix,1)=0.0
C###10 [fc] Loop on line 10 of parmig.f (DO IT) fully vectorized%%%
          do it=1,ntpl
              p(it,ix,2)=p(it,ix,1)
              p(it,ix,3)=p(it,ix,1)
          enddo
      enddo
C
      sc = - (1./32.)*(dt/dx)**2
      do ix=1,nx
          icnt=reed(velfd,alp(1,ix),nt*4)
C###19 [fc] Loop on line 19 of parmig.f (DO IT) fully vectorized%%%
          do it=1,nt
              alp(it,ix)=sc*alp(it,ix)**2
          enddo
      enddo
C
C set up and prefactor tridiagonal systems
C
      do ix=2,nx-1
C###27 [fc] Loop on line 27 of parmig.f (DO IT) fully vectorized%%%
          do it=1,nt
              a(it,ix)=beta+alp(it,ix)
              b(it,ix)=1.0-2.0*a(it,ix)
              c(it,ix)=a(it,ix)
          enddo
      enddo
C###34 [fc] Loop on line 34 of parmig.f (DO IT) fully vectorized%%%
      do it=1,nt
          a(it,nx)=-1.0
          b(it,nx)=1.0
      enddo
C###39 [fc] Loop on line 39 of parmig.f (DO IT) fully vectorized%%%
      do it=1,nt
          b(it,1)=1.0
          c(it,1)=-1.0
      enddo
C
      call trifac(a,nt,b,nt,c,nt,nt,nx)

      inow=1
      iprev=2
      inext=3
C
C now for the "parallel" migration
C
C     P1 P4
C     P2 P3
C
C     (1+(beta+alpha)T) P4 = (1+beta T) (P1+P3-P2) - alpha T (P1+P3+P2)
C
      do itau=nt,2,-1  ! work from bottom up
C
          call rhs(nt,nx,ntpl,p,alp,rtl,rt2,beta,iprev,inow,inext,itau)
          call triply(a,nt,b,nt,c,nt,p(itau,1,inext),ntpl,ntpl-itau,nx)
C
          itemp=iprev
          iprev=inow
          inow=inext
          inext=itemp
      enddo
C
      do ix=1,nx
          icnt=rite(outfd,p(1,ix,inow),nt*4)
      enddo
C
      return
      end

      subroutine rhs(nt,nx,ntpl,p,alp,rtl,rt2,beta,iprev,inow,inext,itau)
      integer nt,nx,ntpl,ix,it,itau
      real p(ntpl,nx,3),alp(nt,nx)
      real rtl(nt,nx),rt2(nt,nx),sc,beta,dt,dx
      integer iprev,inow,inext
C
#define P2(ix) p(it+1,ix,iprev)
#define P3(ix) p(it+1,ix,inow)
#define P1(ix) p(it,ix,inow)
#define P4(ix) p(it,ix,inext)
#define R1(ix) rtl(it,ix)
#define R2(ix) rt2(it,ix)
#define ALPHA(ix)  alp(it-itau+1,ix)
C
      do ix=1,nx
C###91 [fc] Loop on line 91 of parmig.f (DO IT) fully vectorized%%%
          do it=itau,nt
              R1(ix) = P1(ix)+P3(ix)-P2(ix)
              R2(ix) = P1(ix)+P3(ix)+P2(ix)
          enddo
      enddo
      do ix=2,nx-1
C###97 [fc] Loop on line 97 of parmig.f (DO IT) fully vectorized%%%
          do it=itau,nt
              P4(ix) = (1.0-2.0*BETA)*R1(ix) + BETA*(R1(ix-1)+R1(ix+1))
              P4(ix) = P4(ix)-ALPHA(ix)*(R2(ix-1)-2.0*R2(ix)+R2(ix+1))
          enddo
      enddo
C###103 [fc] Loop on line 103 of parmig.f (DO IT) fully vectorized%%%
      do it=itau,nt
          P4(1)=0.0
          P4(nx)=0.0
      enddo
      return
      end
```

```
      subroutine trifac(a,i,b,j,c,k,m,n)
C
C  factor tridiagonal system into LDU.  L replaces a, 1/D replaces b, and
C  U replaces c.  Diagonals of L and U are identically 1 and not stored.
C  a(1) and c(n) are unused placeholders
C
C
      implicit none
      integer i,j,k,m,n
      real a(i,n),b(j,n),c(k,n)
C
#define A(im) a(ii,im)
#define B(im) b(ii,im)
#define C(im) c(ii,im)
#define L(im) a(ii,im)
#define D(im) b(ii,im)
#define U(im) c(ii,im)
C
      integer ii,im
C
      if(n.le.0) return
C
C###23 [fc] Loop on line 23 of trifac.f (DO II) fully vectorized%%%
      do ii=1,m
          D(1)=1.0/B(1)
          U(1)=C(1)*D(1)
      enddo
C
      do im=2,n-1,1
C###29 [fc] Loop on line 29 of trifac.f (DO II) fully vectorized%%%
          do ii=1,m
               D(im)=1.0/(B(im)-A(im)*U(im-1))
               L(im)=A(im)*D(im-1)
               U(im)=C(im)*D(im)
          enddo
      enddo
C
      if(n.eq.1) return
C
C###38 [fc] Loop on line 38 of trifac.f (DO II) fully vectorized%%%
      do ii=1,m
          D(n)=1.0/(B(n)-A(n)*U(n-1))
          L(n)=A(n)*D(n-1)
      enddo
C
      return
      end
```

```
      subroutine triply(l,i,d,j,u,k,y,ly,m,n)
C
C  apply LDU factors to solve tridiagonal system.  Answer overwrite y.
C
      implicit none
      integer m,n,i,j,k,ly
      real l(i,n),d(j,n),u(k,n),y(ly,n)
C
#define Y(im) y(ii,im)
#define E(im) y(ii,im)
#define F(im) y(ii,im)
#define L(im) l(ii,im)
#define D(im) d(ii,im)
#define U(im) u(ii,im)
C
      integer ii,im
C
      if(n.le.0) return
C
C###20 [fc] Loop on line 20 of triply.f (DO II) fully vectorized%%%
      do ii=1,m
          E(1)=Y(1)
      enddo
C
      do im=2,n,1
C###25 [fc] Loop on line 25 of triply.f (DO II) fully vectorized%%%
          do ii=1,m
               E(im)=Y(im)-L(im)*E(im-1)
          enddo
      enddo
C
C###30 [fc] Loop on line 30 of triply.f (DO II) fully vectorized%%%
      do ii=1,m
          F(n)=E(n)*D(n)
      enddo
C
      do im=n-1,1,-1
C###35 [fc] Loop on line 35 of triply.f (DO II) fully vectorized%%%
          do ii=1,m
               F(im)=E(im)*D(im)-U(im)*F(im+1)
          enddo
      enddo
C
      return
      end
```

Halley's comet, 1985-1986:
This has been dubbed the "worst apparition in recorded history". The problem is that it is too symmetrical: instead of one close approach as in 1910, we get two mediocre ones. Also, when the comet is closest to the sun and brightest, the earth is almost exactly on the opposite side of the sun. The entire central portion of the comet's loop about the sun was lost in the glare. Of the two approaches, the April one was the brighter: the comet was both closer and also intrinsically brighter due to its recent baking after it rounded the sun in February. However, the comet faded unexpectedly quickly before the April encounter: by April 7, when it was predicted to be brightest, it had dimmed substantially and lost most of its tail. Those that travelled to Australia were disappointed. However, those that had clear skies in late March were rewarded with a very nice view of the comet standing on its head on the southern horizon. (The April encounter was very far south in the sky; hence the reason to travel to the southern hemisphere.) Huge crowds mobbed Mt. Hamilton at that time, due to an erroneous article in the San Jose paper which led most people who read it to believe the comet would disappear the next day. The view wasn't as good as in 1910, but the crowds were probably even bigger, the weather was good, and it looked like a comet, THE comet...

--JAD



Reprinted with Permission from Mankind's Comet, by Guy Ottewell and Fred Schaaf