Parallel finite-difference migration in practice

Stewart A. Levin

ABSTRACT

In preparation for my SEG talk on vector and parallel finite-difference migration I obtained some free time on the National Science Foundation's CRAY X-MP in San Diego. I also talked Rob Clayton into letting me use his recently acquired Ametek 64-node hypercube for some experiments. This article expands on the required annual NSF report I wrote.

INTRODUCTION

In Levin and Parks (1985) we described effective ways, in theory, to employ vector and parallel computers to image the earth's subsurface using seismic waves recorded at the surface and finite-differences to extrapolate then into the subsurface. The main improvement was a change of coordinates in conventional finite-difference migration that reduced a recursion along all three coordinate axes to a recursion only along two. Along the remaining axis computations decoupled. Later I had the opportunity to try out my ideas on our Convex C-1 vector computer. I reported in SEP-48 (Levin 1986) that I gained about a five-fold improvement on side-by-side benchmarks.

CYCLIC REDUCTION

A second method I suggested for improving parallelism in finite-difference migration was to use cyclic reduction (Levin 1984b). Recently I implemented this in a conventional 15 degree implicit migration program (appendix A.) I then reran the earlier comparison against my SEP-48 vectorized migration and found the performance ratio decrease from 5:1 to 3:1. So I retract my SEP-48 comment that cyclic reduction runs slower in practice that purely scalar solution. Whether similar improvements are available on other machines is architecture dependent. For example on the Convex the need to access vector elements stored in noncontiguous locations was not a drawback because the vectors fit easily into the 64K high-speed cache. Other machines that do not have a large high-speed cache, such as the Cray series, do typically slow down accessing these arrays. Especially when the element-to-element stride is a power of two.*

* A year ago I wrote a power-of-three variant of cyclic reduction for constant coefficient tridiagonal solution (in Cray-1 assembler) to get around this problem. This made all strides odd at the cost of shortening the average vector length by about a third.

A question of general interest I was recently asked is how to apply cyclic reduction to higher order finite-differencing which involves solving pentadiagonal or higher order banded systems. The scalar tridiagonal techniques apply directly to the the higher order systems, needing only obvious changes. But generalizing cyclic reduction is trickier. Here's how it's done. I'll use a pentadiagonal system for concreteness.

First factor the system into ordinary LDU form. Then L is a lower tridiagonal matrix and corresponds to the recursion

$$y_i \;=\; x_i \;-\; l_{i,i-1}\, y_{i-1} \;-\; l_{i,i-2}\, y_{i-2} \tag{1}$$

for computing the y's given the x's and some initial (zero) values. The trick is to recast this second order scalar recursion as a first order vector recursion. Defining

$$\begin{aligned} X_i &= (x_i\,,\; x_{i-1})^T \\ Y_i &= (y_i\,,\; y_{i-1})^T \end{aligned} \tag{2}$$

then relation (1) becomes

$$Y_i \;=\; X_i \;-\; L_i\, Y_{i-1} \tag{3}$$

where

$$L_i \;=\; \begin{pmatrix} l_{i,i-1} & l_{i,i-2} \\ 1 & 0 \end{pmatrix} \quad . \tag{4}$$

Now cyclic reduction can be applied as in the scalar bidiagonal case shown in the appendix. The only difference is that the new coefficients are now 2 by 2 matrices and you must be careful not to multiply them in the wrong order. The U factor can then be applied the same way. For a more detailed analysis as well as alternatives to cyclic reduction I recommend Axelsson and Eijkhout (1986) and Eijkhout (1985).

WHY THE HYPERCUBE?

In July I made a brief visit to Caltech's geophysics department to run a parallel finite-difference migration on a 64 node Ametek hypercube. The reason for the visit was this was the first truly parallel (as opposed to vector) computer I've been able to get my hands on. As the title of my talk includes the word "parallel" it was desirable to have some hands-on experience with finite-difference migration on such a machine. Also I'd been keeping up with the hypercube literature just for the fun of it.

The Ametek hypercube is not a number cruncher. Applying LDU factors of a tridiagonal matrix in one node to a 256 point vector took about a second. The only compiler available is a cross-compiler for C that generates extremely inefficient code. (Rob Clayton has observed a factor of ten when he compiled and ran one of his programs on an IBM personal computer that use the same CPU chip.) Also the Ametek system doesn't permit overlapped computation and node-to-node data transfer. But it still serves as a reasonable diagnostic for how well finite-difference migration will fit on more expensive,

turbocharged brands of hypercube.

The algorithm I used configured the hypercube as a linear list, initialized with zeros, and fed the input time slices into one end (node 0) with the computational grid (two successive subdiagonals) sliding sideways each time to make room. When the last time slice had been input, the completely migrated image was then sucked back out one time slice at a time from node 0 and written to output. Due to careful preparation (i.e. reading the manuals) and a bit of luck, I was able to install, debug, and run my program in one afternoon on the Ametek. On a 64 time slice test I did indeed get runtime improvements of more than a factor of 20 over single processor times.

The improvements were progressively less marked when the number of time slices was increased to the more realistic number 1024. This turns out to be due to elegant but inefficient coding on my part. As the number of time slices was greater than the number of nodes (64) in the hypercube, I divided the computational grid into equal sized chunks of 1024/64 or 16 time levels. Then as the time slices were fed into the cube, they shifted across the first node 16 steps and then shifted to the next node for another 16 steps and so forth. This was wasteful. For example, the first node kicked things off by doing 16*15/2 extrapolation steps while the others nodes sat idle. If they had been used to help, each node would have needed to do at most 16 extrapolation steps to reach the same partial results. Hindsight is wonderful. Nevertheless the hypercube tests did demonstrate the feasibility of parallel finite-difference migration and gave me a good introduction to hypercube programming styles and practicalities.

WHY THE CRAY?

The purpose of using the Cray X-MP/48 for finite-difference migration tests was to work in a combination vector and parallel environment. The X-MP/48 contains four central processing units, each one a vector computer similar to the older Cray-1 series. The four units share the same central memory and I/O channels and can communicate and synchronize with each other through shared registers and semaphores. My idea was to vectorize in essentially the same way I did on our Convex and use two levels of cyclic reduction on the resulting vector recursion to allow all four processors to work in parallel on the computations. I did not succeed in doing this for the simple reason that multitasking is not yet available on the San Diego Cray. So I did not get my "money's-worth" from my half hour Cray allotment.

Second best

What I did instead was benchmark on the Cray the same program I had run on the Convex. The result: Convex 180 seconds, Cray 22 seconds. In terms of megaflops these translate to: Convex 15 Mflops, Cray 120 Mflops. The cycle time of a Convex for 32 bit floating point operations is about 5 times the Cray's. The additional factor of about two in Cray performance is due to its ability to fetch two operands from storage per cycle

```
      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(in) y(im,in)
#define E(in) y(im,in)
#define F(in) y(im,in)
#define L(in) l(im,in)
#define D(in) d(im,in)
#define U(in) u(im,in)
C
      integer im,in
C
      if(n.le.0) return
C
      do im=1,m
           E(1)=Y(1)
      enddo
C
      do in=2,n,1
           do im=1,m
                E(in)=Y(in)-L(in)*E(in-1)
           enddo
      enddo
C
      do im=1,m
           F(n)=E(n)*D(n)
      enddo
C
      do in=n-1,1,-1
           do im=1,m
                F(in)=E(in)*D(in)-U(in)*F(in+1)
           enddo
      enddo
C
      return
      end
```

FIG. 1. Algorithm **triply** used for vectorizing compiler tests. For speed, I expanded the inner loops to work on groups of five consecutive values of the **do** variable **in** in the comparisons reported below.

where the Convex can only fetch one. In short the numbers are consistent.

I also attempted to rate the relative performance of the Convex and Cray vectorizing Fortran compilers. I did this by taking the most heavily used inner loop of the finite-difference migration (subroutine triply of Figure 1.) and recoding it in Convex assembly language. The improvement in run time was only ten percent indicating the Convex compiler was doing a commendable job. I translated the assembly code into Cray assembler and did the same comparison on the X-MP. Result: the run time with the assembler program was about five percent *slower* than the Fortran run. And I thought I knew how to write good vector assembly code!

After going over my assembler subroutine to make sure I didn't make any conversion bugs during the translation process, I read through the X-MP hardware manual to see what architectural differences between the Convex and the X-MP might account for the problem. I found the major difference was in how memory is accessed. The Convex's memory can be accessed at the rate of one word per cycle, the X-MP can read two words per cycle or write one word per cycle. It also has a bidirectional mode enabling the two reads and one write to proceed in parallel. A query to the San Diego consultants determined that the default for Fortran programs is to disable bidirectional mode and told me about a newer version of the Cray Fortran compiler that was able to properly handle bidirectional mode. So I recompiled the Fortran subroutine with bidirectional mode enabled. I also rearranged my assembly code to take advantage of the memory differences and reran my tests. Result: Fortran 20 seconds, assembly 21 seconds. A speed improvement but the difference still remained.

My next suspect was the difference in the order that memory was being accessed by the two versions of the subroutine. So I added a parameter to the program to stagger the arrays the subroutines were using among different combinations of the 64 memory banks of the Cray. Ten side by side runs showed some small timing differences attributable to memory access order but the Fortran subroutine remained consistently faster than the assembler version.

So memory alignment was not the problem. I turned on the assembler listing option of the Fortran compiler and had a look. I discovered two things. First the new Fortran compiler did not generate safe bidirectional code for my loops. But this would not affect timings, only computational results. Second I found that the Fortran compiler took full advantage of the partial loop unrolling I had employed (a trick I took from the BLAS codes distributed by Jack Dongarra) to move register loads to the front and registers stores to the back of the loop thereby reducing significantly the number of register conflicts that were slowing down my assembler code. Simplified a bit, the point is this: a crucial loop required 3 registers loads and one register store per element. The X-MP is capable of 2 loads and one store per cycle. The theoretical best one can hope to attain is then one result every one and a half cycles. My Fortran was nearly achieving this, the assembler code was only producing one result every two cycles. Moral: for parallel

programming you may have to think in terms of fractions.

The unsimplified story is this. The assembler code was slowing down due to register conflicts: a vector register computed during one iteration of a loop was being both used as an arithmetic operand and stored into memory in the next trip through the loop. The Cray doesn't allow both to happen in parallel and delays one until the other is done. The correct thing to do is delay the store, say, one more trip through the loop. By partially unrolling the loop I permitted the Fortran compiler to do just that, delay some stores for two or even three "trips".

Analyzing the subroutine for ultimate efficiency I counted three multiplies, two additions, five memory reads and two memory writes. So with the bidirectional mode enabled, I could hope to attain one output result every three cycles, i.e. it was multiplier limited. I recoded my assembly routine to achieve this goal and reran the test. The Fortran still ran faster!

So the puzzle remains. I still have some ideas but I need pencil and paper, not the Cray, to work them out.

CONCLUSIONS

Cyclic reduction can be used to significantly reduce finite-difference migration time on a vector machine such as the Convex. However parallel reorganization of finite-difference migration provides much superior performance and is still preferred whenever memory limitations aren't a problem.

Use of one of Caltech's hypercubes let me try finite-difference migration on a parallel machine. For a small test case I was able to realize much of the speed improvements predicted in my earlier theoretical musings. For a larger case I didn't realize them because of programming shortcomings, not intrinsic limitations.

The Cray X-MP/48 did not fulfill my intended use as four parallel processors working together on (vector) cyclic reduction. As a vector processor finite-difference migration benchmarked at approximately the performance I would anticipate from comparison with the Convex C-1. The Cray Fortran compiler did an excellent job at vectorizing my migration program and produced code that ran consistently faster than my (supposedly) optimized assembler equivalent.

REFERENCES

Axelsson, O. and Eijkhout, V., 1986, A note on the vectorization of scalar recursions: Parallel Computing 3, 73-83.

Eijkhout, V.L., 1985, Scalar recurrences on chainable pipeline architectures: Univ. Tx. Austin Ctr. Num. Anal. Tech. Rep. CNA-202.

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

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

Levin, S.A., 1986, A program for vectorized finite-difference migration: SEP-48, 161-165.

Levin, S.A. and Parks, T., 1985, Parallel finite-difference migration: Presented at the 2*nd* SIAM Conference on Parallel Processing, November 18, Norfolk.

APPENDIX A

A 15 degree implicit migration using cyclic reduction.

```
# Time domain 15 degree (t',x',tau') migration of zero-offset section
#
#Usage: <Headin Tl5mig nl=nt= dl=dt=  n2=nx=  d2=dx= vc= vt= vtx= vel=
#        trick=.14 stepdown=1 >Headout
#
#
#  the retarded coordinate transform:
#       t'      =       t + tau
#       x'      =       x
#       tau'    =       tau       ( tau = 2*z/v -- psudo-depth )
#
#nl=nt,n2=nx    trace length, number of traces
#dl=dt,d2=dx    sampling intervals
#vc=            constant velocity(v=vc); must specify either vc ,or vt, or vxt
#vt=yes         velocity as a function of tau only
#vtx=yes        velocity as a function of tau and x
#vel=           headfile specifying velocity profile(s) of length nt;
#                 when vt=yes, v=v(nt); vxt=yes, v=v(nt,nx)
#trick=.14      1/6. trick in more accurate finite difference approximation
#                 of second x derivative;  trick<0 when dipfilter needed
#stepdown=1     increment of downward continuing step
#taper=yes      tapering side traces to reduce boundary reflections
#
# author:       Zhiming Li, 8/17/84
#
#%
#       Modified:
#               stew        --- transfered to convex and took out ap code
#               li 12/6/85  --- transpose data and velocity array
#               stew        --- added prefactorization and cyclic reduction
#
define SOURCE '/sep/stew/reverse.dir/src/Tl5mig.r'
define NTMAX 2048
define NXMAX 1024
define MAXSIZ 10000000
real p(NXMAX,NTMAX),u(3*NXMAX),w(NXMAX),z(NXMAX),y(NXMAX),vin(NTMAX)
real e(NXMAX),apb(NXMAX),amb(NXMAX)
real vav(NXMAX),diag(NXMAX),offdi(NXMAX),v(NXMAX,NTMAX),dt,dx,vc,dd
real subdi(2*NXMAX),di(NXMAX),supdi(2*NXMAX)
integer ix,nx,itau,it,nt,nl,n2,infd,input,outfd,output
integer auxin,inf2
integer stepdown
integer ier
integer fetch,is
real transp(NXMAX),tmp,trick
character*10 vt, vtx, taper
real factor

call doc(SOURCE)
infd = input()
outfd=output()

if ( fetch('nl nt','i',nt) == 0 )
        call erexit('nt is required')
if( nt > NTMAX )
        call erexit('nt dimension statement in program too small')
```



```
call putch('nl','i',nt)
if ( fetch('n2 nx','i',nx) == 0 )
        call erexit('nx is required')
if( nx > NXMAX )
        call erexit('nx dimension statement in program too small')
if ( nx*nt > MAXSIZ )
        call erexit('nt*nx too large for the host')

call putch('n2','i',nx)
dt=.004;        ier =  fetch('dl dt','f',dt);       call putch('dl','f',dt)
dx=25.;         ier =  fetch('d2 dx','f',dx);       call putch('d2','f',dx)

# Read data and transpose it
do ix=1,nx
        {
        call reed(infd,vin,4*nt)
        do it=1,nt
                p(ix,it) = vin(it)
        }

ier =  fetch('vc','f',vc)
ier =  fetch('vt','s',vt)
ier =  fetch('vtx','s',vtx)

if( vc > 0.)            # v = constant
{
do itau=1,nt
        do ix=1,nx
                v(ix,itau) = vc
call putlin('constant velocity migration')
call putch('vc','f',vc)
}
else if( vt(:1) == 'y')         # v = v(tau)
{
inf2 = auxin('vel')
call auxpar('nl nt','i',nl,'vel')
call reed(inf2,vin,4*nt)
do it=1,nt
        do ix=1,nx
                v(ix,it)=vin(it)
call putlin('depth-variable velocity migration')
}
else if(vtx(:1) == 'y') # v = v(tau,x)
{
inf2 = auxin('vel')
call auxpar('nl nt','i',nl,'vel')
call auxpar('n2 nx','i',n2,'vel')
do ix=1,nx
        {
        call reed(inf2,vin,4*nt)
        do it=1,nt
                v(ix,it) = vin(it)

        }
call putlin('spatially variable velocity migration')
}
```

```
else
{
call erexit('must specify vc, or vt, or vtx')
}
trick = 0.14; ier =  fetch('trick','f',trick); call putch('trick','f',trick)
if (trick<0.)
        call putlin('migration with dipfilter')
stepdown = 1;
if ( fetch('stepdown','i',stepdown) > nt-1 )
        call erexit('stepdown too large')
if (stepdown<1)
        call erexit('stepdown must be greater than zero ')
call putch('stepdown','i',stepdown)

taper = 'yes'; ier = fetch('taper','s',taper)

if( taper(:1) == 'y' )
{
        do ix = 1,nx/32
                {
                factor = (ix-nx/32-1)/(nx/32)
                e(ix) = exp( - factor * factor )
                }
        do it=1,nt      # Tapering the side traces to avoid boundary
                do ix=1,nx/32
                        {
                        p(ix,it)=p(ix,it)*e(ix)
                        p(nx-ix+1,it)=p(nx-ix+1,it)*e(ix)
                        }
}
tmp = dt*dt*stepdown/(4.*8.*dx*dx)
do itau=stepdown+1,nt,stepdown
        {                                       # Go down in steps of stepdown
        do ix=1,nx
                {                                       # Zero bottom conditions
                u(ix+NXMAX) = 0.
                w(ix) = 0.
                }
       do ix=1,nx
                {
                vav(ix)=0.
                do is=1,stepdown
                        vav(ix) = vav(ix) + v(ix,itau-is)/stepdown
                }
        do ix=1,nx
                {
                # tridiagonal coefficients
                apb(ix) = vav(ix)*vav(ix)*tmp+trick
                amb(ix) = vav(ix)*vav(ix)*tmp-trick
                diag(ix) =1.+2.*amb(ix)
                offdi(ix) = -amb(ix)
                }
        diag(1)=diag(1)+offdi(1)
        diag(nx)=diag(nx)+offdi(nx)
        do ix=1,nx
                {
```



```
                subdi(ix) = offdi(ix)
                supdi(ix+NXMAX) = offdi(ix)
                di(ix) = diag(ix)
                }
        call trifac1(subdi,di,supdi(1+NXMAX),nx)

        do it=nt,itau,-1
                {
                # update the differencing star
                # Differencing star:    w=p(t,z)        u=p(t,z+1)
                #                       y=p(t+1,z)      z=p(t+1,z+1)
                do ix=1,nx {

                        z(ix) = u(ix+NXMAX)
                        y(ix) = w(ix)
                        w(ix) = p(ix,it)
                        }
                # compute right-hand-side column vector; zero-slope b.c.'s
                dd = (1.-apb(1))*(z(1)+w(1))+apb(2)*(z(2)+w(2))
                u(1+NXMAX) = dd-diag(1)*y(1)-offdi(1)*y(1)-offdi(2)*y(2)

                do ix=2,nx-1
                        {
                        dd    = (1.-2.*apb(ix))*(z(ix)+w(ix))
                        dd    = dd + apb(ix-1)*(z(ix-1)+w(ix-1))
                        dd    = dd + apb(ix+1)*(z(ix+1)+w(ix+1))
                        u(ix+NXMAX) = dd-diag(ix)*y(ix)-offdi(ix-1)*y(ix-1)
                        u(ix+NXMAX) = u(ix+NXMAX) -offdi(ix+1)*y(ix+1)
                        }
                dd = (1.-apb(nx))*(z(nx)+w(nx))+apb(nx-1)*(z(nx-1)+w(nx-1))
                u(nx+NXMAX) = dd-diag(nx)*y(nx)-offdi(nx)*y(nx)-offdi(nx-1)*y(nx-1)

                call triply1(subdi,di,supdi(1+NXMAX),u(1+NXMAX),nx)
                do ix=1,nx
                        p(ix,it) = u(ix+NXMAX)
                }
        }
do ix=1,nx
        {
        do it=1,nt
                vin(it) = p(ix,it)
        call rite(outfd,vin,4*nt)
        }
call hclose()
stop;   end
```

```
      subroutine trifacl(a,b,c,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) and lots else are clobbered in this cyclic reduction version.
C
C
      implicit none
      integer n
      real a(n),b(n),c(n)
C
#define L(im) a(im)
#define D(im) b(im)
#define U(im) c(im)
C
      integer im
C
      if(n.le.0) return
C
C First compute ordinary LDU factors
C
          D(1)=1.0/B(1)
          U(1)=-C(1)*D(1)
C
      do im=2,n-1,1
               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
C
      if(n.eq.1) return
C
          D(n)=1.0/(B(n)+A(n)*U(n-1))
          L(n)=-A(n)*D(n-1)
C
C Now modify for cyclic reduction
C
      call rbi(L(1),1,D(1),1,n,1)
      call rbi(U(n),-1,D(n),-1,n,1)
C
      return
      end
```



```
      subroutine triplyl(l,d,u,y,n)
C
C  apply LDU factors to solve tridiagonal system.  Answer overwrite y.
C
      implicit none
      integer n
      real l(n),d(n),u(n),y(n)
C
#define E(im) y(im)
#define F(im) y(im)
C
      integer im
C
      if(n.le.0) return
C
      if(n.gt.1) then
          call rbi(l(1),1,y(1),1,n,0)
      endif
C
      do im=1,n
              y(im)=y(im)*d(im)
      enddo
C
      if(n.gt.1) then
          call rbi(u(n),-1,y(n),-1,n,0)
      endif
C
      return
      end
```

```
C/****** RBI = real bidiagonal solver ****/
C/*
C       call rbi(a,i,d,l,n,fac)
C
C       solves Ax=d where bidiagonal matrix A has the form
C
C       |  1                                |
C       | -A1  1                            |
C       |     -A2  1                        |
C       |         -A3  1                    |
C       |                                   |
C       |                                   |
C       |                                   |
C       |                                   |
C       |                        -An-1  1   |
C
C
C       Answer overwrites "d".  All inputs will be overwritten
C       during calculation. A0 will be modified.
C
C Algorithm separates system to be solved into even and odd numbered  equations.
C The odd numbered equations are used to eliminate the odd index variables from
C the even numbered equations.  The result is a bidiagonal system of half
C the original size for the even index variables. This is recursively subdivided
C until the system reduces to a single equation.  This equation is solved
C and recursively back-substituted for the odd indexed variables.
C
C discontinues reduction when off-diagonals drop to 1e-5 or less.
C switches from vector to scalar mode when system size is 8 or less.
C
C Author: Stewart A. Levin    SEP    adapted SEP-41 tridiagonal solver
C
C INPUTS:
C
C a     coefficient vector of length 2*n.  when fac=1 elements
C       2 through n contain the negative of the original subdiagonal.
C
C i     element-to-element stride in array "a".  i=1 for compact vector.
C       may be negative.
C
C d     right hand side vector of length 2*n.  ignored when fac=1.
C       elements 1 to n contain the original rhs of Ax=d.
C
C l     element-to-element stride in array "d".  l=1 for compact vector.
C       may be negative.
C
C n     number of equations in input system.
C
C fac   1 to factor, 0 to apply cyclic reduction
C
C
C OUTPUTS:
C
C a     factored coefficient array of size 2*n if fac=1.  must be
C       saved for later application with fac=0.
C
```



```
C d     if fac=0 elements 1 to n will be replaced by the solution,
C       elements n+1 to 2*n will contain intermediate results.
C
C */
        subroutine rbi(abar,ibar,dbar,lbar,nbar,fac)
        implicit none
        integer ibar,lbar,nbar,fac
        real abar(1),dbar(1)
C
        integer a,d,anew,dnew
        integer api,dpl  ! second element of vector
        integer ami,dml  ! element preceding vector
        integer a2i,d2l  ! third element of vector
        integer nodd  ! number of odd/even indexed equations
        integer i,ii,j,l,n,i2,l2
        integer stack(20),istk,niter
        real abig,amax1,abs
        real eps/1.0e-5/
C
        istk=0
        a=1
        d=1
        i=ibar
        i2=i+i
        l=lbar
        l2=l+l
        n=nbar
        if(n.le.1) return
        if(fac.eq.0) niter = abar(1)
        if(fac.eq.1) niter = 20
        abar(1) = 0.0
C
 10     nodd = n/2
        anew = a+n*i
        dnew = d+n*l
        ami = a-i
        dml = d-l
        n = n - nodd
        istk = istk+1
C
        if(fac.eq.0) then
            if(n.gt.8) then
C$DIR NO_RECURRENCE
            do ii=1,n
                j=ii-1
                dbar(dnew+j*l) = dbar(d+j*l2) +
     1                           abar(a+j*i2)*dbar(dml+j*l2)
            enddo
            else
C$DIR SCALAR
            do ii=1,n
                j=ii-1
                dbar(dnew+j*l) = dbar(d+j*l2) +
     1                           abar(a+j*i2)*dbar(dml+j*l2)
            enddo
            endif
```

```
        endif
        if(fac.eq.1) then
            abig = 0.0
C$DIR NO_RECURRENCE
            do ii=1,n
                j=ii-1
                abar(anew+j*i) = abar(a+j*i2) * abar(ami+j*i2)
                abig = amax1(abig,abs(abar(anew+j*i)))
            enddo
            if(abig.le.eps) niter=istk
        endif
        a = anew
        d = dnew
        stack(istk) = nodd
C
        if(n.gt.1 .and. niter.gt.istk) goto 10   ! solve system half the size
C
        if(fac.eq.0) then
 20         nodd = stack(istk)  ! restore nodd from stack
            istk = istk-1
            n = n + nodd
            dnew = d-n*1
            anew = a-n*i
            dpl  = dnew+1
            api  = anew+i
            if(nodd.gt.8) then
C$DIR NO_RECURRENCE
            do ii=1,nodd
                j=ii-1
                dbar(dnew+j*12) = dbar(d+j*1)
                dbar(dpl+j*12) = dbar(dpl+j*12) +
     1                           abar(api+j*i2)*dbar(d+j*1)
            enddo
            else
C$DIR SCALAR
            do ii=1,nodd
                j=ii-1
                dbar(dnew+j*12) = dbar(d+j*1)
                dbar(dpl+j*12) = dbar(dpl+j*12) +
     1                           abar(api+j*i2)*dbar(d+j*1)
            enddo
            endif
            d = dnew
            a = anew
            if (istk .gt. 0) goto 20
        endif
C save recursion depth in otherwise unused first element of abar
        abar(1) = niter + 0.5
        return
        end
```