Footnote to parallel x-t migration

Stewart A. Levin

INTRODUCTION

In SEP-38 I described a highly parallel method of performing finite-difference migration in the space-time domain. I mentioned that explicit migration further decoupled computations along the spatial axis which might potentially reduce migration times to acceptable interactive speeds. A recent discussion of parallel algorithms with Gene Golub reminded me of an algorithm that is used for solving tridiagonal equations, and which lends itself to spatial parallelism as well.

FGDP ALGORITHM

The usual way we solve a tridiagonal system in migration is discussed in Claerbout's FGDP. Describing our system by the four vectors A,B,C,D, where the system of coefficients is given by

$$\begin{matrix} b_1 & c_1 & & & & & d_1 \\ a_2 & b_2 & c_2 & & & & d_2 \\ & a_3 & b_3 & c_3 & & & d_3 \end{matrix}$$

$$\begin{matrix} & & & & a_{n-1} & b_{n-1} & c_{n-1} & & d_{n-1} \\ & & & & & a_n & b_n & & d_n \end{matrix}$$

we use the first row to eliminate a_2 from the second row. We then use the new second row to eliminate a_3 from the third row, and so on. This leaves us with an upper bidiagonal system of the form

$$\begin{matrix} 1 & e_1 & & & & & f_1 \\ & 1 & e_2 & & & & f_2 \\ & & 1 & e_3 & & & f_3 \end{matrix}$$

$$\begin{matrix} & & & & 1 & e_{n-1} & f_{n-1} \\ & & & & & 1 & f_n \end{matrix}$$

which can be solved by back substitution in the opposite direction.

This is a serial algorithm: the results from the previous row are needed to reduce the next row. It is also an efficient algorithm: the number of computations grows as $O(n)$ (1 division and 3 multiply-adds per element).

A MORE PARALLEL ALGORITHM

An alternative algorithm is based on nested bisection. It is described in Hockney and Jesshope (1981) as (one form of) cyclic reduction. Hockney developed it in collaboration with Golub. In outline: divide the equations (rows) and the unknowns (columns) of a tridiagonal system into even and odd indices. Use the even-numbered equations to eliminate the even-indexed unknowns from the odd-numbered equations. The result, I derive below, is a tridiagonal subsystem of half the size of the original system. Apply this procedure to that new system to produce another subsystem one quarter the size of the original system. Repeat until a one by one system is reached. Solve this trivial system and back substitute until the original system is solved.

Derivation

Write out the tridiagonal equations algebraically

$$a_i x_{i-1} + b_i x_i + c_i x_{i+1} = d_i \quad ,$$

and introduce the new variables $w_i = x_{2i}$ and $u_i = x_{2i+1}$. Take three successive equations

$$\begin{array}{llllllll}
a_{2i} & u_{i-1} & + & b_{2i} & w_i & + & c_{2i} & u_i & = & d_{2i} \\
a_{2i+1} & w_i & + & b_{2i+1} & u_i & + & c_{2i+1} & w_{i+1} & = & d_{2i+1} \\
a_{2i+2} & u_i & + & b_{2i+2} & w_{i+1} & + & c_{2i+2} & u_{i+1} & = & d_{2i+2}
\end{array}$$

and eliminate the w's from the middle equation to get

$$\left(-\frac{a_{2i+1}a_{2i}}{b_{2i}}\right)u_{i-1} + \left(b_{2i+1} - \frac{a_{2i+1}c_{2i}}{b_{2i}} - \frac{c_{2i+1}a_{2i+2}}{b_{2i+2}}\right)u_i + \left(-\frac{c_{2i+1}c_{2i+2}}{b_{2i+2}}\right)u_{i+1}$$

$$= d_{2i+1} - \frac{a_{2i+1}d_{2i}}{b_{2i}} - \frac{c_{2i+1}d_{2i+2}}{b_{2i+2}}$$

This is a tridiagonal system in the u's. After solving it, rearrange the third equation to get the formula

$$w_i = \frac{d_{2i+2}}{b_{2i+2}} - \frac{a_{2i+2}}{b_{2i+2}} u_i - \frac{c_{2i+2}}{b_{2i+2}} u_{i+1}$$

for the remaining unknowns.

Example

Let's work a numerical example. Suppose the original system is

2.0	-1.0								1.0
-1.0	2.0	-1.0							-1.0
	-1.0	2.0	-1.0						1.0
		-1.0	2.0	-1.0					2.0
			-1.0	2.0	-1.0				-1.0
				-1.0	2.0	-1.0			0.0
					-1.0	2.0	-1.0		1.0
						-1.0	2.0	-1.0	3.0
							-1.0	2.0	-1.0

First we reduce this to

1.5		-0.5							0.5
0.5	-1.0	0.5							0.5
-0.5		1.0		-0.5					1.5
		0.5	-1.0	0.5					-1.0
		-0.5		1.0		-0.5			0.0
				0.5	-1.0	0.5			0.0
				-0.5		1.0		-0.5	2.5
						0.5	-1.0	0.5	-1.5
						-0.5		1.0	4.0
								0.5	-1.0	0.5

From this we extract the new tridiagonal subsystem

1.5	-0.5				0.5
-0.5	1.0	-0.5			1.5
	-0.5	1.0	-0.5		0.0
		-0.5	1.0	-0.5	2.5
			-0.5	1.0	4.0

Again eliminate even-indexed variables from odd-numbered equations to get

1.25		-0.25			1.25
0.5	-1.0	0.5			-1.5
-0.25		0.5		-0.25	2.0
		0.5	-1.0	0.5	-2.5
		-0.25		0.75	5.25

and extract the subsystem

1.25	-0.25		1.25
-0.25	0.5	-0.25	2.0
	-0.25	0.75	5.25

Reduce this in turn to

1.125		-0.125	2.25
0.5	-1.0	0.5	-4.0
-0.125		0.625	6.25

and extract the subsystem

1.125	-0.125	2.25
-0.125	0.625	6.25

This we reduce to

$$\begin{array}{rrr} 1.1 & & 3.5 \\ 0.2 & -1.0 & -10.0 \end{array}$$

and we finally extract

$$\begin{array}{rr} 1.1 & 3.5 \end{array}$$

with solution 35/11'ths.

Appendix A lists a computer program that uses this method to solve tridiagonal equations on an FPS array processor. Just as the FGDP serial method described earlier, this alternate tridiagonal algorithm require $O(n)$ arithmetic operations. Its drawback is that it requires more arithmetic (1 division, 11 multiplies and 6 additions) per point than the FGDP solution. Its advantage is that the vector calls are the operations that could be done in parallel. Thus, using the bisection algorithm on a parallel machine, the tridiagonal equations can be solved in $2\log_2 n$ serial steps* as opposed to the $2n$ serial steps needed for the FGDP algorithm. The speedup factor is $n/\log_2 n$. For a 1024 trace section this factor is just over 100, about one-fifth of the parallelism for explicit migration of these data but still well within the range of interactive migration (perhaps 1.5 sec/mig; see SEP-38).

AN EVEN MORE PARALLEL ALGORITHM

This variant of cyclic reduction, termed PARACR in Hockney and Jesshope, applies the above reduction to all indices, not only the even indices, and defines $(a,b,c,d) = (0,1,0,0)$ for subscripts less than 1 or greater than n. Reduction results in another set of n three-term equations half of which are the $n/2$ equations from the cyclic reduction algorithm I described earlier. Continuing the reduction for skips of 4, 8, 16, etc. we arrive after $\log_2 n$ steps at a set of n

* and possibly fewer. If the system that we are solving is diagonally dominant, we can normalize the equations so that b is identically 1 and we see that the normalized off-diagonal elements of the halved system are approximately equal to the squares of the normalized off-diagonal elements of the original system implying their magnitudes decrease very rapidly (geometrically) towards zero.

equations involving x_i, x_{i-n}, and x_{i+n}. Because the latter two values (being outside the subscript range 1,...,n) are zero the resulting system is diagonal. At this point simply divide by the diagonal elements to produce all the x_i's. The earlier cyclic reduction algorithm only solves for one unknown at the corresponding point. In other words, the tridiagonal system has been solved in half the number of serial steps of the previous method.

Interestingly, this algorithm though fastest on a parallel machine, requires $O(n \log n)$ floating point computations, so it is less efficient in a serial computer than either of the previous $O(n)$ methods. It attains its power on parallel architectures by follows all branches of the reduction simultaneously.

CAVEAT

Parallelism is never infinite in actual architectures. Hockney and Jesshope give first order descriptions of a machine's actual parallelism in terms of $n_{1/2}$, the vector length for which the algorithm attains half its asymptotic efficiency. When vector lengths are $>> n_{1/2}$ the machine's resources are saturated and the execution times function like those of a serial machine. When vector lengths are $<< n_{1/2}$ the machine mimics an infinitely parallel device. Typical $n_{1/2}$'s are around 2 for the FPS-120B, 10 for the CRAY-1, 100 for the CYBER-205, and 1000 for the ICL DAP (a distributed processor array). Therefore with the parallel machines currently in use it is impossible to take advantage of all the possible parallelism I have proposed for finite difference migration.

CONCLUSION

In SEP-38 I showed how to introduce a high degree of parallelism into conventional time-domain finite-difference migration algorithms. I have briefly shown in this note how parallelism may be even further advanced for implicit schemes as well as explicit schemes.

REFERENCES

Claerbout, J.F., 1976, Fundamentals of geophysical data processing: McGraw Hill, 188-189.

Hockney, R.W. and Jesshope, C.R., 1981, Parallel Computers: Adam Hilger Ltd, 286-289.

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

APPENDIX A. TRI PROGRAM LISTING

This subroutine solves tridiagonal equations using the algorithm outlined in the text. Note that the signs of a and c are reversed from those in the text.

```
/****** TRI = (real) tridiagonal solver (VFC version) *****/
/*
                call tri(a,i,b,j,c,k,d,l,n)
```

solves Tx=d where tridiagonal matrix has the form

```
| B0 -C0                                  |
|-A1  B1 -C1                              |
|    -A2  B2 -C2                          |
|        -A3  B3 -C3                      |
|                                         |
|                                         |
|                                         |
|                                         |
|                              -An-1 Bn-1 |
```

Answer overwrites ``d''. All inputs will be overwritten during calculation. A0 and Cn–1 will be modified.
A and C may not be equivalenced.

Algorithm separates system to be solved into even and odd numbered equations.
The odd numbered equations are used to eliminate the odd index variables from
the even numbered equations. The result is a tridiagonal system of half
the original size for the even index variables. This is recursively subdivided
until the system reduces to a single equation. This equation is solved
and recursively back–substituted for the odd indexed variables.
*/

```
tri(a,i,b,j,c,k,d,l,n)                                          tri
int a,i,b,j,c,k,d,l,n;
{
      int api,bpj,cpk,dpl; /* second element of vector   */
      int a2i,b2j,c2k,d2l; /* third element of vector    */
      int nodd,nm1; /* number of odd/even indexed equations */
      int n0;

      n0 = n; /* save so we know when to exit */
      if(n>1)
            goto reduce;
      svdiv(b,j,d,d,l,n);                    /* else x = d/b */
      return;

reduce:    nodd = n/2;
      n = n - nodd;
      nm1 = n-1;
      api = a+i;
      bpj = b+j;
      cpk = c+k;
      dpl = d+l;
      a2i = api+i;
      b2j = bpj+j;
      c2k = cpk+k;
      d2l = dpl+l;
```

```
    i=i<<1;                          /* double vector increments */
    j=j<<1;
    k=k<<1;
    l=l<<1;

#define TMONE 04001
    tmmov(TMONE,a,1);                /* fetch 1.0 from table memory */
    svdiv(bpj,j,a,bpj,j,nodd);   /* 1/b(2j+1) */
    vmul(bpj,j,api,i,api,i,nodd); /* a(2i+1)/b(2j+1) */
    vmul(bpj,j,cpk,k,cpk,k,nm1); /* c(2k+1)/b(2j+1) */
    vmul(bpj,j,dpl,l,dpl,l,nodd); /* d(2l+1)/b(2j+1) */
    vneg(b,j,b,j,n);
    vma(a2i,i,cpk,k,b2j,j,b2j,j,nm1);
    vma(c,k,api,i,b,j,b,j,nodd);
    vneg(b,j,b,j,n);                 /* new b diagonal */
    vma(a2i,i,dpl,l,d2l,l,d2l,l,nm1);
    vma(c,k,dpl,l,d,l,d,l,nodd); /* new d vector */
    vmul(a2i,i,api,i,a2i,i,nm1); /* new a vector */
    vmul(c,k,cpk,k,c,k,nm1);         /* new c vector */

    apmdout(bpj,nodd);               /* save nodd on ``stack'' */
    if(n>1)
        /* tri(a,i,b,j,c,k,d,l,n); */   /* solve system half the size */
        goto reduce;

    svdiv(b,j,d,d,l,n);              /* solve n=1 system: b x = d */

subst: nodd = apmdin(bpj);           /* restore nodd from stack */
    vma(api,i,d,l,dpl,l,dpl,l,nodd);   /* back substitute solution */
    vma(cpk,k,d2l,l,dpl,l,dpl,l,nm1); /* of half sized system      */
    i = i>>2;
    j = j>>2;
    k = k>>2;
    l = l>>2;
    n = n+nodd;
    nm1 = nm1+nodd;
```

```
        api = api-i;
        bpj = bpj-j;
        cpk = cpk-k;
        d2l = dpl;
        dpl = dpl-l;
        i = i<<1;
        j = j<<1;
        k = k<<1;
        l = l<<1;
        if (n < n0)
            goto subst;
}
```