Computational Techniques for Downward Continuation

Because of our basic belief that accurate velocity estimation and multiple reflection removal ultimately hinge on techniques for downward continuation (migration) we continued our efforts to improve techniques. The continuing hassle has been the organization of computer memory. Because we found the mid-point variable always on our inner loops we began by looking at multiplexing. Then we discovered explicit techniques which are stable, twice as fast, and which enable us to reorder calculations in a wide variety of ways. One attractive reordering is to get the midpoint coordinate on the outer DO loop. This is like building a machine which accepts seismic traces as inputs. After some appropriate (space) delay migrated traces emerge as outputs. Internal memory is independent of the length of the survey line.

J.F.C. 24 September 1974

THE BALANCED TWO-WAY MERGE ALGORITHM

by W. Scott Dunbar

The amount of seismic data from a particular survey is often far too large to be stored in a computer at one time. Consequently, if any rearranging (i.e., multiplexing, transposing, etc.) of this data is to be done, special external sorting algorithms for use with tapes or disks are necessary.

Entire books have been written on sorting (see reference). Fortunately, most multiplexing or transposition problems can be reduced to a "merging" problem. Merging means the combination of two or more files (records) into a single ordered file (record).

The simplest algorithm used for tape merging is the balanced two-way merge. It proceeds by sorting two files (records) into one and starts by dividing the original set of data into two parts.

The two-way algorithm is best explained by an example. Suppose that one wanted to transpose a 3 row by 4 column array, $A(i,j)$ $i=1,3;\ j=1,4$. The array may be stored on a tape as follows (by columns):

$$[\ a_{11}\ a_{21}\ a_{31}\ ][\ a_{12}\ a_{22}\ a_{32}\ ][\ a_{13}\ a_{23}\ a_{33}\ ][\ a_{14}\ a_{24}\ a_{34}\ ]$$

The numbers within the brackets constitute one column of the array. The transpose of the array would be a 4 row by 3 column array as shown:

$$[\ a_{11}\ a_{12}\ a_{13}\ a_{14}\ ]\ [\ a_{21}\ a_{22}\ a_{23}\ a_{24}\ ]\ [\ a_{31}\ a_{32}\ a_{33}\ a_{34}\ ]$$

where the brackets now denote a column of the transposed array.

By inspecting these two arrays, it may be seen that the numbers of the original array are in the correct sequential order, but not in the correct consecutive order.

The merging algorithm uses four "working" tapes and begins by dividing the original data into two parts. It is convenient to put odd columns on Tape 1 and the even columns on Tape 2 as follows:

Tape 1: [a_{11} a_{21} a_{31}] [a_{13} a_{23} a_{33}]

I III

Tape 2: [a_{12} a_{22} a_{32}] [a_{14} a_{24} a_{34}]

II IV

Tapes 3 and 4 are left blank. The two columns on each tape are called "runs" or sorted portions of the array. Thus, there are 4 runs of length 3. Tapes 1 and 2 are now rewound.

Looking at the transposed array, we see that a_{11} must be followed by a_{12} and that a_{21} must be followed by a_{22} etc. Therefore, alternately picking one number from runs I and II, the following is written on Tape 3:

Tape 3: [a_{11} a_{12} a_{21} a_{22} a_{31} a_{32}]

I

Alternately picking one number from runs III and IV, the following is written on Tape 4:

Tape 4: [a_{13} a_{14} a_{23} a_{24} a_{33} a_{34}]

II

Thus we have 2 runs of length 6; i.e., the run length has doubled and the number of runs has been halved. All tapes are now rewound.

The next step is to alternately pick two numbers from runs I and II and write on Tape 1 as follows:

Tape 1: $[\ a_{11}\ a_{12}\ a_{13}\ a_{14}\ a_{21}\ a_{22}\ a_{23}\ a_{24}\ a_{31}\ a_{32}\ a_{33}\ a_{34}\ ]$

which is the required transpose. Note that we now have one run of length 12; i.e., again the run length has doubled and the number of runs has been halved.

Two passes (plus the original pass) were made of the data to do the transpose. This is $\log_2 4$, where 4 is the number of columns. A logarithmic number of passes is typical of this type of algorithm.

Note that the number of columns is a power of two, but that the number of rows is arbitrary. In this sense, the algorithm is something like a Fast Fourier Transform. For an arbitrary number of columns, the logic would have to be altered, probably in a similar manner to that of a generalized Fourier transform (which is really a multi-way merge, a generalization of the present two-way algorithm). To obtain a number of columns which is a power of two, the array could be padded with zeros. These could then be retrieved after transposition.

Most seismic data can be stored in the form of a three dimensional array $A(i, j, k)$ where i denotes time, j is the receiver-shot offset and k is the number of records. The array would be stored such that the innermost subscript, i , varies the fastest while the third subscript, k , varies the slowest. That is, there are k two dimensional arrays. To transpose time and receiver-shot offset

(i and j) , the balanced two-way merge algorithm is used k times.

Transposition of the last two indices (j and k) is slightly more difficult to visualize than the last example. However, if we realize that an element of an array can be a vector (or anything, for that matter), the method required is the balanced two-way merge itself. Consider the following example of an I by 3 by 2 array:

$$[A_{11}][A_{21}][A_{31}][A_{12}][A_{22}][A_{32}]$$

where A_{jk} represents a vector of length I. This can be transposed to an I by 3 by 2 array by treating each vector just like the elements of a 3 row by 2 column array. Note that the third subscript, k , must be a power of two.

Shown overleaf is a subroutine that will transpose a M row by 2^N column array stored columnwise, where M and N are integers. It is a rather inefficient routine, since it assumes one word records. This, however, can be easily modified to suit a particular application, such as the ones above. One change that could certainly be made is to incorporate the initial read (off the original tape) and the initial pass of the data into one operation. Thia would require internal storage.

Also shown is a test program that transposes a 9 row by 8 column array.

Reference

Knuth, D. E., 1973, Sorting and Searching;The Art of Computer Programming, v. 3. Addison-Wesley.

```
      SUBROUTINE TRANSP (NC,NR,NT,LU,NFT)
C
C     A MATRIX TRANSPOSE ROUTINE WHICH USES NO CORE, BUT TAKES
C     LOG2 (NC) PASSES AT 4 TAPES USING A BALANCED TWO-WAY MERGE
C     ALGORITHM.
C
C     INPUT:
C
C     NC- NUMBER OF COLUMNS. MUST BE A POWER OF TWO.
C     NR- NUMBER OF ROWS.
C     NT- LOGICAL UNIT ON WHICH ORIGINAL ARRAY RESIDES.
C     LU- AN ARRAY CONTAINING THE 4 LOGICAL UNIT NUMBERS.
C
C     OUTPUT:
C
C     NFT-LOGICAL UNIT ON WHICH TRANSPOSE RESIDES.
C
C     COUNTERS:
C
C     NTR,NTW- SUBSCRIPTS FOR LU ARRAY DENOTING A READ,WRITE
C              LOGICAL UNIT. THESE MAY BE INITIALIZED TO ANY
C              EVEN NUMBER.
C     NRUN- NUMBER OF RUNS (SORTED PORTIONS) OF ARRAY DURING
C           ONE PASS.
C     IP- NUMBER OF ELEMENTS TO BE READ DURING PASS AT ARRAY.
C
C     THIS PROGRAM ASSUMES ONE REAL NUMBER PER RECORD. TO BE
C     EFFICIENT IT SHOULD BE MODIFIED FOR MORE NUMBERS PER
C     RECORD. TRANSPOSING THE LAST TWO INDICES OF A THREE
C     DIMENSIONAL ARRAY COULD BE DONE BY IMPLIED DO LOOPS
C     ON THE READ/WRITE STATEMENTS. THIS ALSO INCREASES
C     RECORD LENGTH. TRANSPOSING THE FIRST TWO INDICES OF A
C     THREE DIMENSIONAL ARRAY COULD BE DONE BY A SERIES OF TWO
C     DIMENSIONAL TRANSPOSES. BUFFERING COULD ALSO BE USED
C     IN THE PROGRAM.
C
C     THE PROGRAM IS SET UP TO TRANSPOSE AN TWO DIMENSIONAL
C     ARRAY STORED COLUMNWISE.
C
C     REFERENCE:
C
C     D.E. KNUTH, 1973. SORTING AND SEARCHING; THE ART OF COMPUTER
C                      PROGRAMMING, V.3, (P.7).
C
C     SCOTT DUNBAR, MAY 1974.
C
      INTEGER LU(4)
C
C REWIND ALL TAPES
C
      REWIND NT
      DO 1 I=1,4
      J=LU(I)
    1 REWIND J
```

```
C
C INITIALIZE COUNTERS
C
      NRT2=NR*2
      NTW=2
      NTR=2
      NRUN=NC
      NP=0
      IP=1
      NADDW=3
      NADDR=1
C
C INITIAL READ OF COLUMNS ONTO LU(1) AND LU(2)
C
      DO 3 I=1,NC
      NTW=MOD(NTW,2)+1
      LUW=LU(NTW)
      DO 2 J=1,NR
      READ (NT) X
    2 WRITE (LUW) X
    3 CONTINUE
C
C DO THE TRANSPOSE
C
    4 DO 5 I=1,4
      J=LU(I)
    5 REWIND J
      NRUN=NRUN/2
C
      DO 7 I=1,NRUN
      NTW=MOD(NTW,2)+NADDW
      LUW=LU(NTW)
      DO 7 J=1,NRT2
      NTR=MOD(NTR,2)+NADDR
      LUR=LU(NTR)
      DO 6 K=1,IP
      READ (LUR) X
    6 WRITE (LUW) X
    7 CONTINUE
C
      NP=NP+1
      IF (NRUN.EQ.1) GO TO 8
      IP=IP*2
      NSAVE=NADDW
      NADDW=NADDR
      NADDR=NSAVE
      GO TO 4
C
    8 NFT=LUW
      WRITE (6,100) NP
      RETURN
C
  100 FORMAT (//' THE TRANSPOSE WAS COMPLETE AFTER',I4,' PASSES'//)
C
      END
```

MAIN PROGRAM

```
      DIMENSION A(9,8),B(8,9)
      INTEGER LU(4)
      NR=9
      NC=8
      NT=20
      DO 1 I=1,4
    1 LU(I)=NT+I
      DO 2 J=1,NC
      DO 2 I=1,NR
      A(I,J)=J+0.1*I
    2 WRITE (NT) A(I,J)
      WRITE (6,100) ((A(I,J),J=1,NC),I=1,NR)
      CALL TRANSP (NC,NR,NT,LU,NFT)
      REWIND NFT
      DO 3 J=1,NR
      DO 3 I=1,NC
    3 READ (NFT) B(I,J)
      WRITE (6,101) ((B(I,J),J=1,NR),I=1,NC)
      STOP
C
  100 FORMAT (1X,8F5.2)
  101 FORMAT (1X,9F5.2)
C
      END
```

JOB CONTROL LANGUAGE

```
//GO.FT20F001 DD DSNAME=ORIG,UNIT=2314,
//      SPACE=(TRK,1),DCB=(RECFM=VBS,BLKSIZE=7294)
//GO.FT21F001 DD DSNAME=NO1,UNIT=2314,
//      SPACE=(TRK,1),DCB=(RECFM=VBS,BLKSIZE=7294)
//GO.FT22F001 DD DSNAME=NO2,UNIT=2314,
//      SPACE=(TRK,1),DCB=(RECFM=VBS,BLKSIZE=7294)
//GO.FT23F001 DD DSNAME=NO3,UNIT=2314,
//      SPACE=(TRK,1),DCB=(RECFM=VBS,BLKSIZE=7294)
//GO.FT24F001 DD DSNAME=NO4,UNIT=2314,
//      SPACE=(TRK,1),DCB=(RECFM=VBS,BLKSIZE=7294)
```

```
              ORIGINAL ARRAY

 1.10 2.10 3.10 4.10 5.10 6.10 7.10 8.10
 1.20 2.20 3.20 4.20 5.20 6.20 7.20 8.20
 1.30 2.30 3.30 4.30 5.30 6.30 7.30 8.30
 1.40 2.40 3.40 4.40 5.40 6.40 7.40 8.40
 1.50 2.50 3.50 4.50 5.50 6.50 7.50 8.50
 1.60 2.60 3.60 4.60 5.60 6.60 7.60 8.60
 1.70 2.70 3.70 4.70 5.70 6.70 7.70 8.70
 1.80 2.80 3.80 4.80 5.80 6.80 7.80 8.80
 1.90 2.90 3.90 4.90 5.90 6.90 7.90 8.90

THE TRANSPOSE WAS COMPLETE AFTER   3 PASSES

              TRANSPOSED ARRAY

1.10 1.20 1.30 1.40 1.50 1.60 1.70 1.80 1.90
2.10 2.20 2.30 2.40 2.50 2.60 2.70 2.80 2.90
3.10 3.20 3.30 3.40 3.50 3.60 3.70 3.80 3.90
4.10 4.20 4.30 4.40 4.50 4.60 4.70 4.80 4.90
5.10 5.20 5.30 5.40 5.50 5.60 5.70 5.80 5.90
6.10 6.20 6.30 6.40 6.50 6.60 6.70 6.80 6.90
7.10 7.20 7.30 7.40 7.50 7.60 7.70 7.80 7.90
8.10 8.20 8.30 8.40 8.50 8.60 8.70 8.80 8.90
```