SEL Report #130

Application of Sorting Networks to Sparse Matrix Problems.

T. N. Mudge

K. Hadavi

May 25, 1979

This work was supported in part by the National Science Foundation under grant NSF-ENG-78-5779.

DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING
SYSTEMS ENGINEERING LABORATORY
THE UNIVERSITY OF MICHIGAN, ANN ARBOR

APPLICATIONS OF SORTING NETWORKS TO SPARSE MATRIX PROBLEMS

by

T. N. Mudge and K. Hadavi

Technical Report

June 10 1979

The Computer Information and Control Engineering Program
and
The Electrical and Computer Engineering Department
The University of Michigan

ABSTRACT

This report outlines how sorting networks can be used to efficiently perform the following operations frequently found in problems involving sparse matrices.

1. Row operations between sparse vectors.
2. Dot products between sparse vectors.

Sorting networks -- in particular Batcher sorting networks [B] -- are introduced. The architecture of an attached support type processor for sparse matrix problems is proposed, in which sorting networks are used to gather data for processing.

TABLE OF CONTENTS

1. INTRODUCTION

Large sparse matrices occur in a wide range of applications that include the following: network analysis problems where large numbers of circuits are cascaded together, finite element structural analysis, linear programming problems, electric power system analysis, as well as the more recently developed field of image reconstruction. Sparse matrices are so called because they have only a small percentage of non-zero elements. For example, it is not unusual to find matrices of rank O(1000) having no more than 10 non-zero elements per row, in electric power flow and transient stability problems.

In this report m will be used to denote the number of elements in a sparse vector, or the rank of a sparse matrix. Its meaning will be clear from the context. Similarly k will be used to denote the upper bound on the number of non-zero elements in a sparse vector, or in any row or column of a sparse matrix. Sparsity implies that $k<<m$. Furthermore, we shall assume that the distribution of non-zero elements is random, although this is not the case for many important classes of sparse matrices, e.g. tridiagonal matrices. Discussion of techniques for handling more structured sparse matrices, such as tridiagonal matrices, is outside the scope of this report. The term array will be used to denote vectors of unspecified dimension.

Taking advantage of sparsity to speed up array operations poses a number of problems for computer architects. High speed vector processor architectures typified by supercomputers such as the CRAY I, the CDC STAR, and the TI ASC appear unable to take advantage of sparsity. This is largely due to the pipeline organization of their processing units. To keep these units busy it is necessary to restructure a problem into a chain of vector operations between full vectors. For sparse arrays having a random structure this "vectorization" appears to be beyond current techniques [C]. Hence, these machines are very much under utilized when running sparse array problems. Recent research, using electric power flow and stability problems as benchmarks, indicates a more cost effective approach is to run these problems on a conventional machine, such as a PDP/11, and off-load the numerical computations to a high speed attached processor, such as the Floating Point Systems' AP120B [BPW]. The AP120B is a 38 bit floating point unit with a three segment pipe. Having such a short pipe gives the unit the appearance of a scalar processor, this relaxes the need to vectorize problems that run on it. From this it can be concluded that computer architectures that are organized around long pipelines appear unsuitable for sparse array problems, unless some suitable "gathering " mechanism can be devised that vectorizes the sparse arrays. In this report presents some techniques that use sorting networks as a gathering mechanism for the specific application of large sparse array problems. However, some problems still

remain regarding the accessing of data from memory and its subsequent alignment after it has been processed and is ready to be returned to memory.

In the next section, section 2, sorting networks -- in particular Batcher's bitonic sorter [B] -- are introduced.

Section 3 outlines how the bitonic sorter can be used to efficiently perform the following two types of operations frequently found in problems involving sparse matrices.

1. Row operations between sparse vectors.
2. Dot products between sparse vectors.

Based on this a processor incorporating a bitonic sorter is proposed for the specific application of sparse array handling.

Section 4 concludes with a discussion of extensions to the ideas presented in this report, and a discussion of the unsolved problems.

2. SORTING NETWORKS

A sorting network is a device with n input ports and n output ports that accepts a list of up to n data items on its inputs and sorts them based on some key specified in each data item. The sorted list is then passed to its output ports.

There is a well developed theory available to the designer of sorting networks (see [Kn] pp. 220-246). Most of this is devoted to the construction of sorters from a basic building block called a comparison-exchange module. As the name suggests this module accepts two items of data on its input ports, compares them using a key in each datum, then based on the outcome of this comparison, conditionally exchanges them before passing them to its output ports. Figure 1 illustrates a comparison-exchange module. The data is input through ports A and B. If A's key (denoted by key(A)) is less than or equal to key(B), then A is output through the port marked L and B is output through the port marked H. Otherwise, if key(a)>key(B) then an exchange is performed and A goes out on H and B on L. A simplified diagram is shown alongside. The position of the arrowhead indicates the H output and the tail indicates the L output.

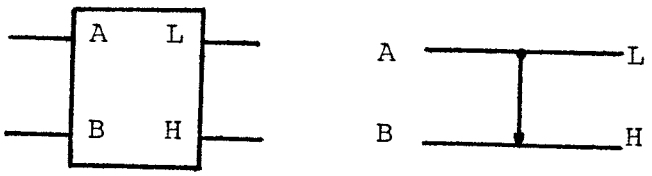


Figure 1. Comparison-Exchange Module.

Using comparison-exchange modules Batcher has demonstrated two methods for constructing sorting networks. The first is based on a network that merges two lists by separately merging the odd positioned items in each list and the evenly positioned items in each list, and then combining these two sublists using a row of comparison-exchange modules. The other is based on a bitonic sorting network which we will discuss in more detail shortly. Both require $O(n\log^2 n)$[1] comparison-exchange modules to sort a list of n items, and both take $O(\log^2 n)$ time units to do the sort. Other networks have been demonstrated that are marginally better [V], however, so far no improvements on the asymptotic behavior of the above bounds have been shown for

---

[1] All logarithms are to base 2, unless otherwise noted.

networks in which fanout is not allowed (i.e. an output port of a comparison-exchange module can feed at most one input port of any other module). Figure 2 illustrates a sorting network based on Batcher's bitonic sorter. Note the absence of fanout. If fanout is allowed sorters have been demonstrated that sort in

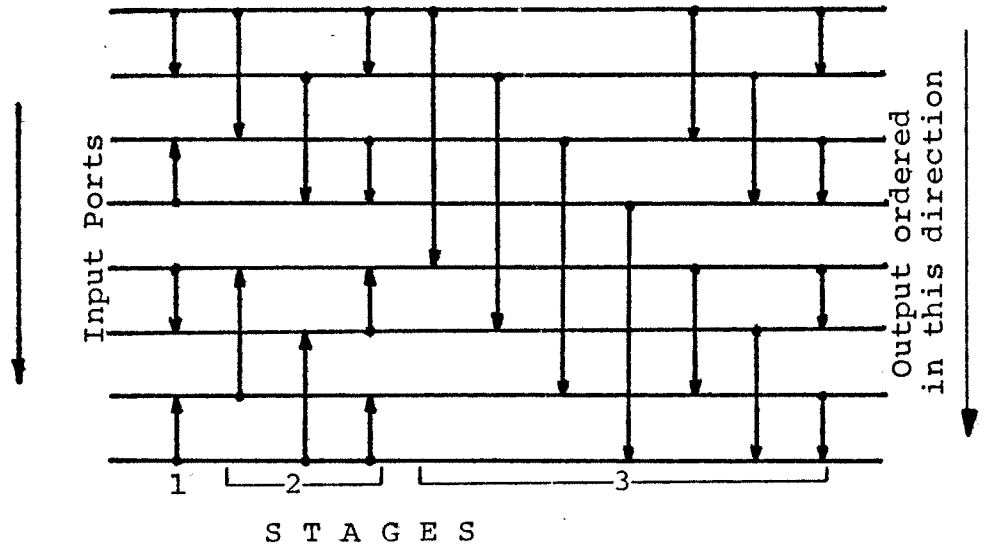


Figure 2. An 8 Input Sorter.

O(logn) time [MP]. These are based on simpler elements than comparison-exchange modules, but they require a discouraging O(n) such elements.

For the purposes of this report only bitonic sorters will be considered in any detail. To explain how they work we first need to define the term "bitonic".

Definition: A bitonic sequence is one of the following:

1. A monotonic sequence.
2. A sequence that partitions into two monotonic sequences: one increasing, the other decreasing.
3. A sequence that after cyclic permutation (rotation) falls into one of the above categories.

It is understood that the term sequence refers to an ordered list of items from some base set that can be put in one-to-one correspondence with a subset of the natural numbers.

From the above definition it can be readily seen that the following sequences defined on the natural numbers are bitonic:

A. 2 6 7 8 17 16 3 1
B. 34 31 18 2 64 72
C. 13 16 25 22 16 15 6 3 4 6 10

Figure 3 shows an iterative rule for constructing a bitonic sorter to sort bitonic sequences of n items into a totally order list. From this it can be seen that when n is a power of two,

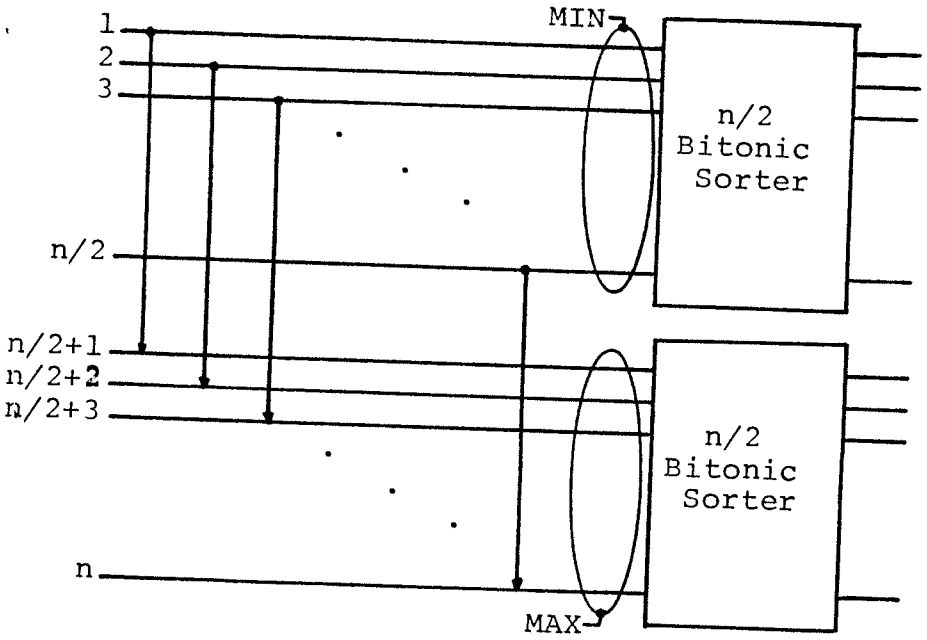


Figure 3. Iterative Rule to Construct A Bitonic Sorter

the number of comparison-exchange modules is given by (nlogn)/2. The key to the iterative rule of Figure 3 can be stated as follows: If the input to the n/2 comparison-exchange modules is a bitonic sequence, then the output is two bitonic sequences, MIN and MAX (see Figure 3), such that every item in MIN is less than or equal to every item in MAX. This is stated more formally in the following theorem due to Batcher.

Theorem 1. Let A = A[1],...,A[n] be a bitonic sequence (assume n is even for convenience) Let MIN[i] = min(A[i],A[i+n/2]) and MAX[i] = max(A[i],A[i+n/2]) for $1 \leq i \leq n/2$. Then the two sequences MIN and MAX are both bitonic, and MIN[i] $\leq$ MAX[j] for all i and j.

Proof: See [B] Appendix B.

In Figure 4 a geometrical illustration is presented to give some insight into Theorem 1 (this is similar to the approach

taken by Stone in [S]). The line ABC represents a typical bitonic sequence. It can be viewed as a plot of the values of the items in the sequences versus their position in the sequence. Point M is the midpoint in the sequence, i.e. there are an equal number of items on either side of M. Items at

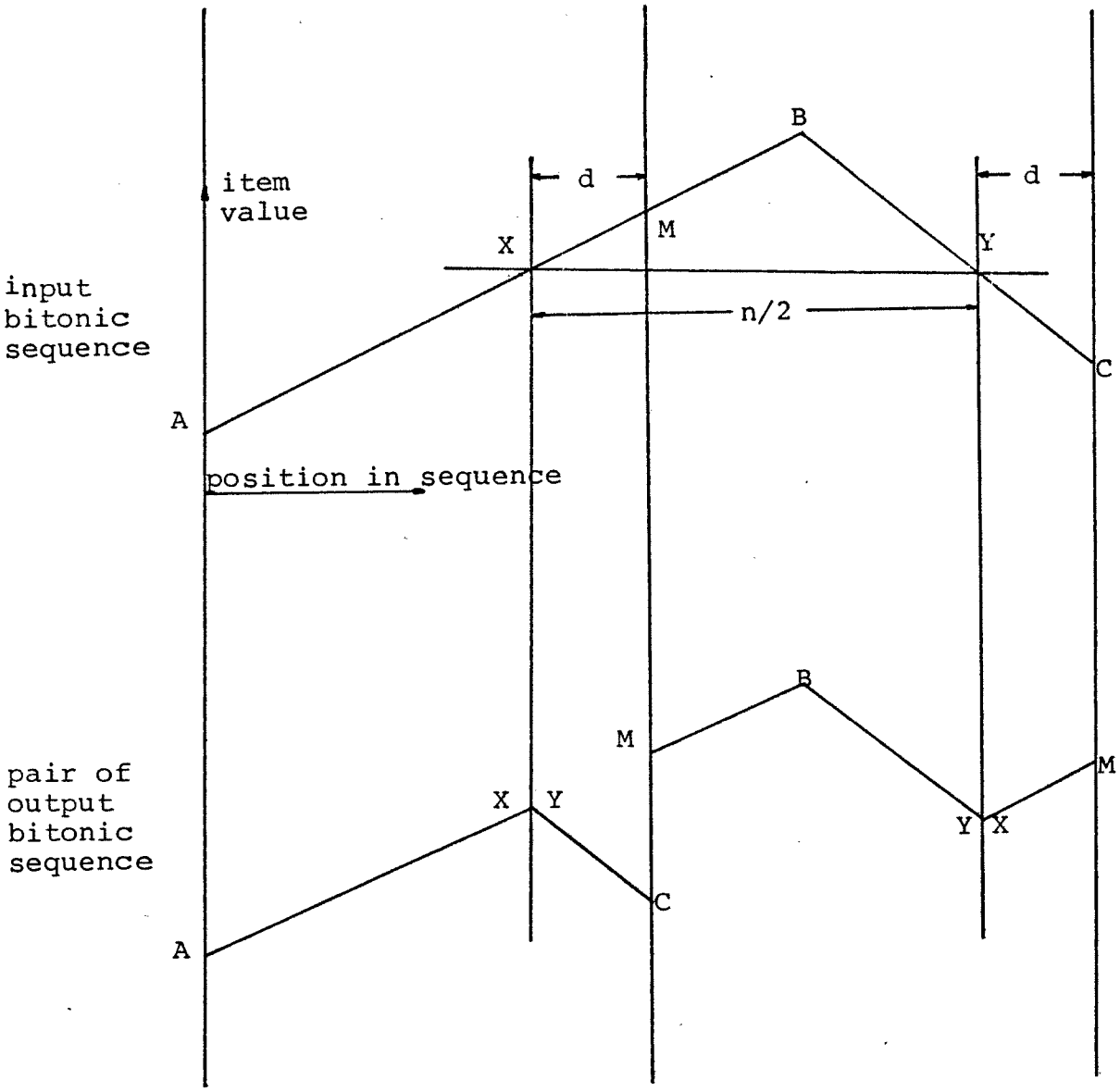


Figure 4. Geometrical Aid to Theorem 1.

points X and Y are separated by half the length of the input sequence (hence they will be compared in the same comparison-exchange module). Furthermore, all items to the left of X (those on AX) are less than or equal to those between M and Y, and all items between X and M are greater than or equal to the items to the right of Y (those on YC). Since items on AX are compared with those on MBY and items on XM with those on YC the output shown at the bottom of Figure 4 results. Note AXYC and MBYXM are

both bitonic and every term in the first is less than or equal to those in the second. Rotating ABC in the horizontal direction causes both AXYC and MBYXM to rotate also, but at twice the frequency.

From the above discussion it should be clear that bitonic sorters can be used to merge two ordered lists by forming a bitonic sequence from the lists then inputting them into the sorter. One way the bitonic sequence can be formed is by concatenating one list with the reverse of the other. This technique was used in Figure 2 to construct a full sorter. Referring to that figure it can be seen that Stage 1 consists of 4 bitonic sorters that sort pairs of items into ordered lists of length 2 (these sorters are simply comparison-exchange modules). These lists are arranged to form two 4 item bitonic sequences. These 2 sequences are sorted using the two 4 input bitonic sorters of Stage 2. The output of Stage 2 is arranged as an 8 item bitonic sequence which is then sorted by the 8 input bitonic sorter of Stage 3.

In some cases it is necessary to merge two sequences whose combined length is greater than n, the number of inputs to the sorter. The next theorem shows how to do this and gives an upper bound on the number of time steps required.

Theorem 2. Given 2 ordered lists A and B of length s and t respectively, and a bitonic sorter with n inputs such that $s>>n$ and $t>>n$, the upper bound on the time to merge A and B is given by $O((s+t)\log n/n)$.

Proof: Assume, for simplicity, that s=pn, t=qn and n=2l where p, q and l are integer. Then the following procedure merges A[1:s] with B[1:t] and places the result in C[1:s+t].

```
Procedure MERGELONGLISTS
     i<-j<-k<-1;
     T[1,..,n]<-merge A[1,..,l] and B[1,..,l];
      comment T is a temporary vector. Use of the n input
      bitonic sorter is indicated by underlining "merge"
      endcomment;
     C[1,..,l]<-T[1,..,l];
while i< 2*p and j< 2*q do
begin if A[i*l]>B[j*l] then
     begin T[1,..,n]<-merge T[l+1,..,n] and
                                   B[j*l+1,..,(j+l)*l];
          j<-j+l;
          C[k*l+1,..,(k+l)*l]<-T[1,..,l];
          k<-k+l
     end
else begin T[1,..,n]<-merge T[l+1,..,n] and
                                   A[i*l+1,..,(i+l)*l];
          i<-i+l;
          C[k*l+1,..,(k+l)*l]<-T[1,..,l];
          k<-k+l
     end
endif
endwhile
if j2*q then C[k*l+1,..s+t]<-A[i*l+1,..s] else
                    C[k*l+1,..,s+t]<-B[j*l+1,...,t] endif
endMERGELONGLISTS
```

This algorithm partitions A and B into 2p and 2q blocks of n/2 items respectively. It then merges (using the sorter) block-wise. Consider the situation after the i-th block of A and the j-th block of B have been merged into the partial result. The last item in the partial result is either A[il] or B[jl]. If it is the former this implies there may be some items B[u] (u>jl) that are less than A[il], hence the next block to be merged into the partial result must be the j+l-st block of B. However, this block need only be merged with the last n/2 items in the partial result. A symmetrical argument arises if B[jl] is the last item in the partial result -- the i+l-st block from A must be merged into the partial result. This merging is repeated until either A or B is exhausted. In the case where 2p>2q a concluding vector transfer of the remaining 2(p-q) blocks in A to the result list c is performed (the final if statement). A symmetrical case arises for B if 2p<2q.

In the worst case the while loop will be traversed 2p+2q times (i.e. k will step from 1 to 2p+2q). Whatever the result of the test A[il]>B[jl] a merge using the sorter and taking O(logn)

time units takes place, followed by a vector transfer which we assume takes O(1) time units. Thus the worst case time is given by:

$$O((2p+2q)\log n) = O((s+t)\log n/n)$$

3. APPLICATION OF SORTERS TO SPARSE ARRAYS

In this section we shall show how bitonic sorters can be use to facilitate two of the most common operations found in manipulating sparse arrays and sparse matrices in particular. These are row operations between sparse vectors, and dot products between sparse vectors. To do this we shall assume that non-zero array elements are available together with their indices in the form of data items with several fields. Any one of these fields may be used as the key for the sorter. Figure 5 illustrates an element from a matrix together with its index fields. In general the number of index fields will equal the dimension of the array from which the element has come. In figure 5 they are shown in APL order (row followed by column, see [P]), although other orderings may be more convenient depending on the application.

A[7,3]=12 stored as:

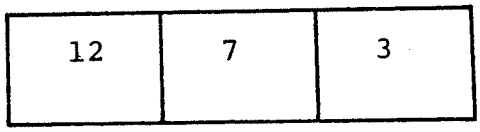


Figure 5. Storage of a non-zero element.

After developing methods for row operations and dot products based on bitonic sorters the design of a special purpose processor for sparse matrix handling will be outlined.

3.1 Row Operations

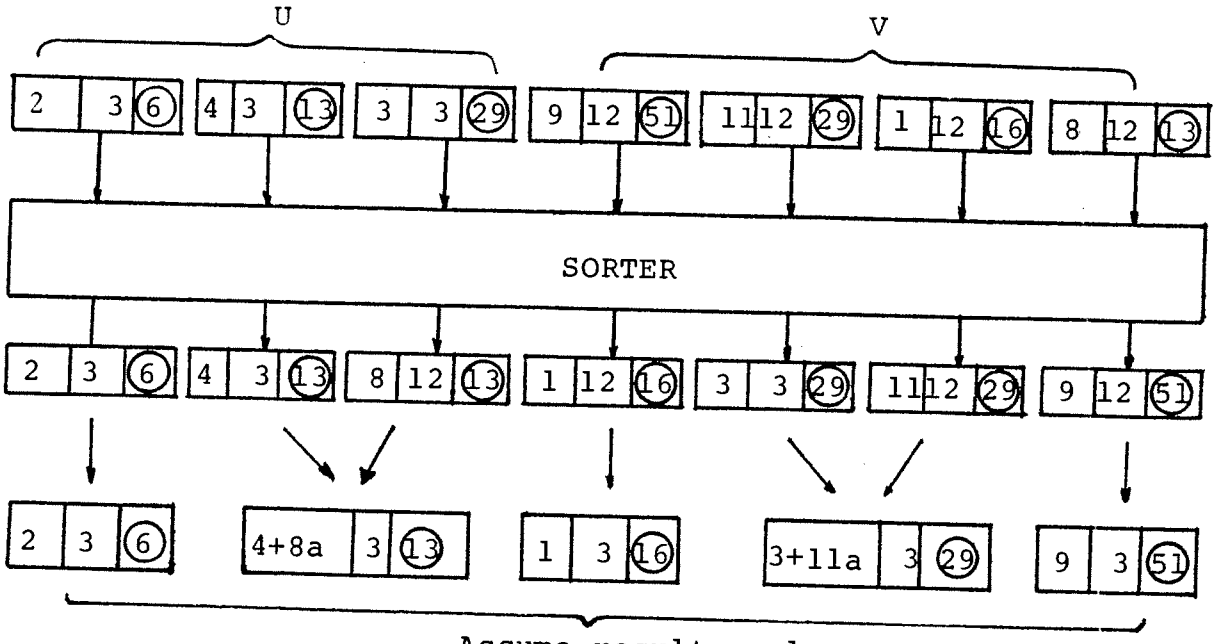
A row operation is an element-wise operation between two vectors to form a third, that is defined as follows:

for i <- 1 until m do U[i] <- V[i] + a*W[i];
where U, V and W are vectors, and a is a scalar.

In the case of sparse vectors a time consuming activity associated with this type of operation is pairing off corresponding non-zero elements in the two vectors, in preparation for scaling and adding. This preprocessing can be done efficiently using a bitonic sorter. Figure 6 shows how.

In Figure 6 two sparse vectors from a matrix (rows 3 and 12) are preprocessed ready for the element-wise row operation. The sorter insures that elements to be combined appear at

adjacent output ports of the sorter. This happens because items with equal keys (i.e. identical indices) appear next to one another after sorting. They can then be fed directly into processing elements for scaling and adding. Since figure 6 shows two row vectors from the same matrix, the elements from both are



**keys used in sorting are circled.

Figure 6. Preprocessing for a Row operation

sorted using their second index field as a key. Furthermore, since the sorter requires a bitonic sequence to operate on, the non-zero elements of the first vector are input in row major order, and those of the second are input in reverse row major order. The result is in row major order. Notice that only the non-zero elements are ever used in this method, so they are the only ones that need to be stored, and further that no testing for zero elements is needed. Hence, the efficiency of this method is related only to the number of non-zero elements, not to the overall number of elements.

If the width of the sorter is greater than the combined width of the two vectors, i.e. $n>2k$, a pair of vectors can be preprocessed in $\log n$ time steps. This is an improvement over the list searching and matching required in a conventional environment that is by nature a serial operation (if we assume that in most cases $O(2k)=n$, then preprocessing two sparse vectors in a serial fashion would take $O(n)$ time steps). An even more important source of improvement arises by noticing that the sorter lends itself naturally to pipelining -- registers need only be inserted between each one of the $\log n$ levels. This gives

a further increase in performance, provided the processing elements have sufficient bandwidth to keep pace with the sorter. If this is the case and the pipe can be kept full, row operations can now be performed in the time it takes to traverse just a single stage of the sorter. In section 3.3 we will work out an estimate for this. In the case of row operations between 2 vectors where k>n. The result of Theorem 2 now reduces to O((s+t)/n), i.e. an apparent speed up by a factor of n over serial merging -- the best technique for combining two sparse vectors in a more conventional processor.

3.2 Dot Products

A dot product is an operation between two vectors to form a scalar, that is defined as follows:

```
a <- 0;
for i <- 1 until m do a <- a + U[i]*V[i];
```
where U and V are vectors and the result, a, is scalar.

The same approach to preprocessing that was used for row operations can be used. However, only those elements that appear on the output ports adjacent to other elements with identical index fields need to be considered by the processing elements. These pairs must be multiplied, and then all the products must be accumulated. Notice that in the case of one vector being a row vector from a matrix and the other a column vector (as frequently occurs when matrices, or vectors and matrices are multiplied), the first one is sorted using its second index field as a key and the second is sorted using its first index field as a key.

3.3 A Sparse Matrix Processor

In this section we shall outline a design for a sparse matrix processor based on a bitonic sorter. It could function as the main arithmetic processing unit in a machine designed specifically with sparse matrix problems as an application, or it could operate as an attached processor in the style of the AP120B. Figure 7 shows the block diagram.

The processing unit has 16 input ports that are each 64 bits wide -- 32 bits for a floating point number, and 32 bits for two 16 bit index fields. S is a bitonic sorter comprising 32 comparison-exchange modules arranged into 4 levels of 8 modules, with each level separated by D flip-flops to allow pipelining. The 16 outputs of S are feed into 16 matching elements (the Ms in Figure 7) that can be programmed to detect whether their input ports are carrying data with matching keys. The operation of an M unit depends on whether a row operation or dot product

is to be performed, which of the index fields are being used as keys, and whether the left adjacent M unit detects a match.

If a row operation is to be performed an M element responds in one of three ways.

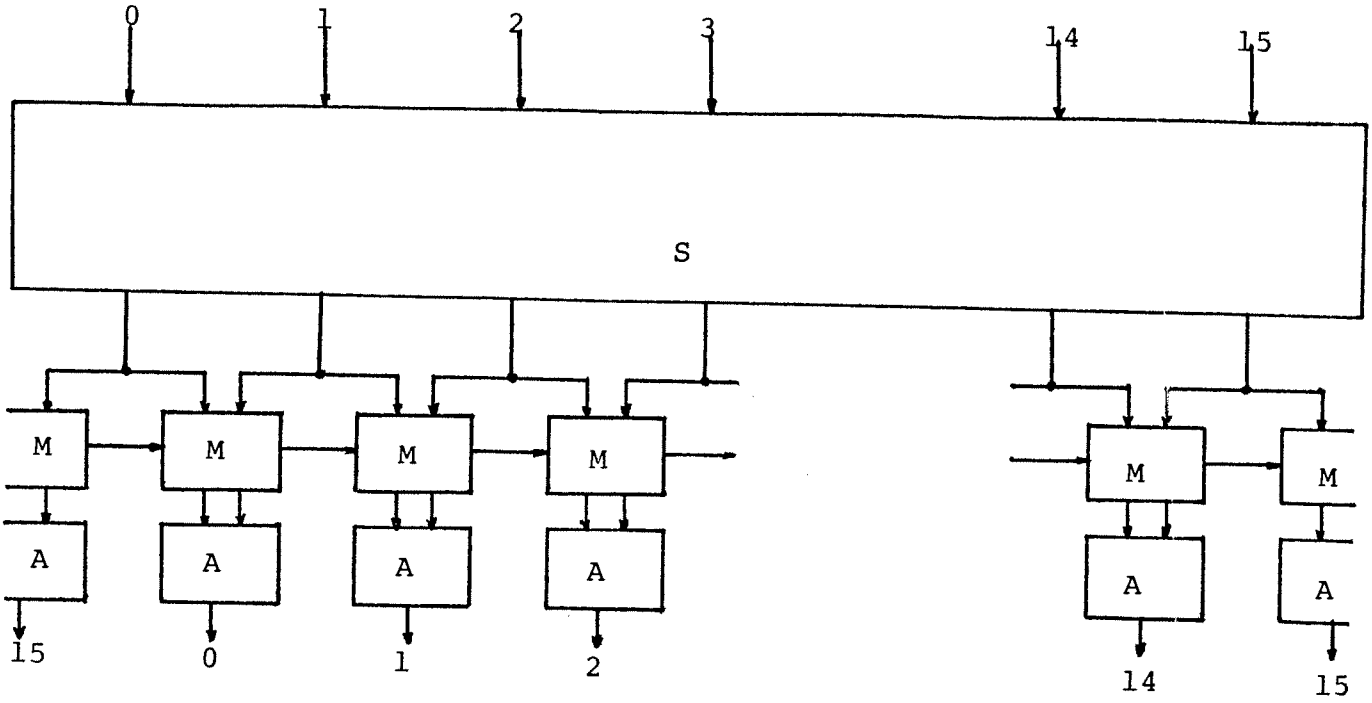


Figure 7. Sparse Matrix Processing Unit.

1. If the appropriate index fields match, the two 32 bit floating point numbers input to the element are sent to an arithmetic processing element, A. In A one is scaled and then they are added, and the result is output on the A element's output.

2. If the appropriate index fields do not match, and if the left adjacent M element does not find a match, then the M element outputs on its left output the data item on its left input. This is transmitted through the corresponding A element. The right input is ignored.

3. If the appropriate index fields do not match, and if the left adjacent M element does find a match, then the M element ignores both of its inputs.

Note that no more than 2 consecutive outputs of S match at any

time. Hence, pair-wise comparisons are sufficient. Furthermore, this means that the unit shown in Figure 7 can be simplified because no more than one of any consecutive pair of A elements will be used at any one time. Therefore, only 8 A elements are needed plus some multiplexing logic to enable sharing.

If a dot product is to be performed an M element responds in one of two ways.

1. If the appropriate index fields match the two 32 bit floating point numbers input to the element are sent to an A element where they are multiplied.

2. If they do not match both inputs are ignored.

To complete the dot product operation the output from the A elements must be accumulated. This can be done most efficiently using a multi-operand (8 in this case) carry-save adder tree (see [AO] for design details).

In general, the index fields of the results of row operations or dot products are selected from the index fields of the inputs to the M elements. This task is handled by the M elements.

To get a rough idea of the gate complexity of this processor, as well as its operating speed, consider the realization of one of the 32 comparison-exchange modules. Figure 8 shows a possible design. The width in bits of data paths is indicated by drawing a slash through the path and writing the number of bits next to it. For simplicity it is assumed that the same pair of index fields are always used for comparison. This requires additional programmable logic at the input to the sorter to make sure this is true. This extra logic is more than offset by the saving in the sorter design.

Let the ordered tuple <g,i> denote the number of gates (g) and the number of inputs to those gates (i) used in the realization of a logic function. Also, let T be the delay of a single gate.

The complexity of realization of a 1 bit latch	= <5,9>
Time to operate	= 4T
Hence, complexity of the two 64 bit pipeline latches	= <640,1152>

The comparator that matches indices is a 16 bit fixed point adder. Its time to operate will be a bottleneck in the operation of the comparison-exchange module. Therefore, we shall consider it to be implemented using a tree of lookahead units that lookahead across 4 bits (thus the tree has 5 lookahead units in it).

Complexity of the 16 bit subtractor
(including a 16 input NOR gate to detect a match) = <217,552>
Time to operate = 10T
Complexity of

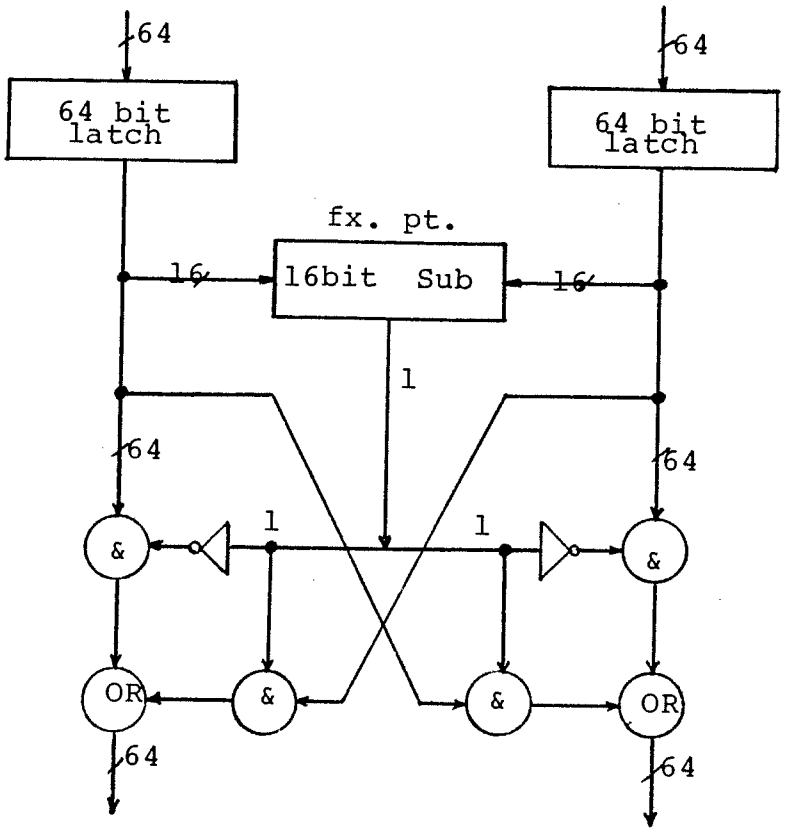


Figure 8. Details of a Comparison-Exchange Module.

the two 2-to-1 64 bit wide multiplexers = <386,770>
Time to operate = 3T
Hence, complexity of comparison-exchange module = <1243,2474>
And time to operate = 17T

Assume state-of-the-art ECL logic (e.g. Fairchild's F100K series). Then T = 1nS, i.e. time/pipeline stage = 17nS. Thus if the pipeline can be kept busy it can multiply two rank 1000 matrices in 17mS, provided $k \leq 8$ for both of the matrices. This bound is in keeping with the types of matrices that occur in power systems problems (see [BPW]).

The total complexity of the sorter is about 40,000 gates

with 80,000 inputs. This compares favorably with present levels of integration for HMOS technology, but is beyond present ECL technology. The major problem with realizing a sorter is not gate complexity but pin count limitations: if a comparison-exchange module were in a single package it would require >256 pins. Hence the problem of partitioning the design of a sorter still needs attention. In the above discussion the M elements and the A elements were assumed to be able to match the bandwidth of the sorter. Furthermore, their complexity was considered to be much less than that of the sorter, i.e. not more than a few thousand gates.

APPLICATION OF SORTERS TO SPARSE ARRAYS

4. CONCLUSION

The ideas presented in this report offer solutions to some of the problems that face the designer of a high speed computer for handling large sparse matrix problems. However, several problems still remain. Firstly, a minor point, and that is that the performance of the processor described in section 3.3 deteriorates if the arrays are not sparse. Clearly, if the arrays were full it would not be necessary to use the sorter, and so using it with full arrays just causes it to be an unnecessary bottleneck. Secondly, and much more important, data still must be accessed before it is input to the sorter so that its indices, that are to be used as sorting keys, form a bitonic sequence. How this is to be achieved depends heavily on the organization of the machine's memory system, and the data structure used to store the arrays in that memory.

There are a number of methods for storing the non-zero elements of a sparse matrix. Most are based on storing the non-zero elements, often with their indices, as linked lists (see [G] for further details). Unfortunately these methods do not fit in with the current techniques for data access and alignment that have been developed for vector processors (see [Ku] and [L]), as these techniques were developed for the more usual case of full arrays. At present it is still not feasible to store many of the sparse arrays that occur in real situations as if they were full ones because of their large rank. Furthermore, anticipated decreases in hardware cost will encourage attempts at the solution of larger sparse matrix problems, even if they encourage the solution of present sparse problems by using the techniques developed for full ones. Hence, the problems of data storage, access and alignment for sparse matrices will not be solved by cheap hardware, at least not in an obvious way.

Several lines of research suggest themselves. Firstly, in order to match the bandwidth of the memory system with processors such as that proposed in section 3.3 it is necessary to use multiple banks of memory, so that several words (data items) can be accessed simultaneously. This is incompatible with the linked list storage schemes referenced above, because of their sequential nature. In a sequential environment linked list storage performs better than hashing for the storage of sparse matrices. In a parallel processor this may no longer be the case, because each one of the independently addressable memory banks can have its own private hash table. Thus hashing will not inhibit parallelism, and deserves further consideration. Secondly, if elements of arrays are stored as depicted in Figure 5, and the bitonic sorter is expanded to a full sorter, then data access and alignment becomes much more simple. Data can be scattered much more randomly throughout memory and still be aligned. Of course the price for this is a factor of logn increase in the size of the sorter.

5. REFERENCES

[AO] Atkins, D. E., and S. C. Ong, "Time-Component Complexity of Two Approaches to Multi-Operand Binary addition", IEEE TC, to appear.

[B] Batcher, K. E., "Sorting Networks and Their Applications", 1968 Spring Joint Computer Conf., AFIPS Proc., Vol. 32, pp. 307-314, 1968.

[BPW] Barry, D. E., C. Pottle, and K. A. Wirgau, Technology Assessment Study of Near Term Computer Capabilities and Their Impact on Power Flow and Stability Simulation Programs, EPRI Final Rept. EL-946, Dec. 1978.

[C] Calahan, D. A., "Vectorized Solution of Load Flow Problems", Exploring Applications of Parallel Processing to Power Systems Analysis Problems, EPRI Special Rept. EL-566-SR, Oct. 1977.

[G] Gustavson, F. G., "Some Basic Techniques for Solving Sparse Systems of Linear Equations", Sparse Matrices and Their Applications, pp. 41-52, Ed. D. J. Rose and R. A. Willoughby, Plenum Press, 1972.

[Kn] Knuth, D. E., "Sorting and Searching", The Art of Computer Programming, Vol. 3, Addison-Wesley, 1973.

[Ku] Kuck, D. J., "A Survey of Parallel Machine Organization and Programming", ACM Computing Surveys, Vol. 9, No. 1, pp. 29-59, Mar. 1977.

[L] Lawrie, D., "Access and Alignment of Data in an Array Processor," IEEE TC, Vol. C-24, No. 12, pp. 1145-1155, Dec. 1975.

[MP] Muller, D. E. And F. P. Prepareta, "Bounds to Complexities of Networks for Sorting and for Switching", JACM, Vol. 22, No. 2, pp. 195-201, Apr. 1975.

[P] Pakin, S., APL\360, 2nd Ed. Science Research Associates, 1972.

[S] Stone, H. S., "Parallel Processing with the Perfect Shuffle", IEEE TC, Vol. C-20, No. 2, pp. 153-161, Feb. 1971.

[V] Van Voorhis, D., "An Improved Lower Bound for Sorting Networks", IEEE TC, Vol. C-27, No. 6, pp. 612-613, Jun. 1972.