Bitonic Sort on a Mesh-Connected Parallel Computer

DAVID NASSIMI AND SARTAJ SAHNI

Abstract—An \( O(n) \) algorithm to sort \( n^2 \) elements on an Illiac IV-like \( n \times n \) mesh-connected processor array is presented. This algorithm sorts the \( n^2 \) elements into row-major order and is an adaptation of Batcher's bitonic sort. A slight modification of our algorithm yields an \( O(n) \) algorithm to sort \( n^2 \) elements into snake-like row-major order. Extensions to the case of a \( 2 \times 2 \)-dimensional processor array are discussed.

Index Terms—Bitonic sort, complexity, mesh-connected parallel computer, parallel sorting, SIMD machine.

I. INTRODUCTION

Batcher's bitonic sort [2], [6, pp. 232–233, 237] is based upon his algorithm to sort a bitonic sequence into nondecreasing order. A sequence \( X = (x_1, x_2, \ldots, x_n) \) is said to be bitonic [2], [8] if either 1) there is an index \( i, 1 \leq i \leq N \), such that \( x_1 \leq x_2 \leq \cdots \leq x_i \geq x_{i+1} \geq \cdots \geq x_n \) or 2) the sequence can be shifted cyclically so that condition 1) is satisfied. Batcher's algorithm to sort a bitonic sequence \( X \) is to recursively sort the bitonic sequences \( X_{ODD} = (x_1, x_3, x_5, \ldots) \) and \( X_{EVEN} = (x_2, x_4, x_6, \ldots) \) and then perform the comparison-interchanges \( x_1 : x_2, x_3 : x_4, x_5 : x_6, \ldots \). During the comparison-interchange \( x_i : x_{i+1} \), \( x_j \) is replaced by the smaller of \( x_j \) and \( x_{j+1} \) and \( x_{j+1} \) becomes the larger of the two. Any sequence \( Y = (y_1, y_2, \ldots, y_n) \) may be sorted by recursively sorting \( (y_1, y_2, \ldots, y_{n/2}) \) into nonincreasing order, \( (y_{n/2+1}, \ldots, y_n) \) into nondecreasing order (or vice versa) and then sorting the bitonic sequence \( (y_1, y_2, \ldots, y_n) \) into nondecreasing order using Batcher's method.

Bitonic sort has been adapted by Orcutt [7] and Thompson and Kung [9] for an \( n \times n \) mesh-connected parallel computer. The computer consists of \( N = n^2 \) identical processors configured in a manner similar to the Illiac IV machine [1]. The assumptions we shall be making on the model are as follows.

1) It is a SIMD type [4] machine. The \( N = n \times n \) identical processors may be thought of as positioned according to an \( n \times n \) array \( P(0:n-1, 0:n-1) \). Each processor \( P(i, j) \) is connected to its neighbor processors \( P(i+1, j), P(i, j+1), P(i-1, j), P(j+1, i) \), and \( P(i, j-1) \) if they exist. The end-around connections of the Illiac IV are not assumed here.

2) Each processor has three registers: one routing register \( R_s \) and two storage registers \( R_a \) and \( R_r \).

3) A register interchange instruction with time \( = \tau_r \). Each selected processor unconditionally interchanges the contents of two of its registers. (The same registers are used for all processors.) In our algorithm, only column-selectability and row-selectability of processors is needed.

4) A route instruction with time \( = \tau_r \). All processors route the contents of their \( R_s \) to their immediate neighbor in the same direction. Thus, this instruction simply shifts the entire \( R_s \)-array (end-off, zero-filled) unit-distance in one of the four directions up, down, left, or right.

5) A compare-interchange instruction with time \( = \tau_c \).

All processors do the (hardware) equivalent of the following statement:

\[
\text{If } \text{sign} (I, S) \ast (R_s - R_a) < 0 \text{ then interchange } (R_s, R_a)
\]

where \( I = \text{processor index} \) and \( S = \text{"pass number" of the algorithm. The function sign will be specified later. After a compare-interchange instruction, we shall refer to the element in } R_s \text{ as the } \text{"accepted" element (to be kept by the processor and the one in } R_a \text{ as the } \text{"rejected" element (to be routed back). Note that even though all processors carry out this instruction, only } N/2 \text{ of the processors would be doing "useful work." The result of the other half is "don't care."}

The sorting problem studied in [3], [7], and [9] is that of routing the contents of the \( n \times n \) routing registers to destination processors. Each data item is to be routed to a distinct processor. The processors are assumed indexed in some manner and the routing is such that the \( i \)-th processor is to contain the \( i \)-th smallest element, \( 1 \leq i \leq N \). Three different indexing schemes have been considered by Thompson and Kung [9]: row-major, shuffled row-major, and snake-like row-major. In row-major order, the index \( i \) of processor \( P(i, j) \) is \( i \times n + j \) (i.e., processors are indexed left to right, top to bottom). In shuffled row-major, the index of a processor is obtained by shuffling its row-major index. For example, if the row-major index in binary is \( b_1 b_2 b_3 b_4 b_5 b_6 b_7 b_8 \) then its shuffled index is \( b_1 b_3 b_5 b_7 b_2 b_4 b_6 b_8 \). Snake-like row-major index is obtained by indexing the processors by row (as in row-major). Processors on even rows are indexed left to right while those on odd rows are indexed right to left (recall that rows are numbered 0 through \( n - 1 \)).

Thompson and Kung [9] present fast parallel algorithms for sorting into snake-like row-major and shuffled row-major order. For snake-like row-major order, they present an \( s^2 \)-way merge algorithm requiring \( 6n + O(n^{2.5} \log n) \) routing steps and \( n + O(n^{2.5} \log n) \) comparison-
interchanges. Thus, the time needed to sort \( n^2 \) elements into
snake-like row-major order is \((6n + 0(n^{2.3} \log n)r_r + (n + 0(n^{2.3} \log n)r_c)\) time. Following a sort into snake-like row-
major order the elements may be rearranged into row-major
order by reversing the order of elements in odd numbered
rows. The additional time needed for this is \((2(n - 1)r_r +
0(\log n)r_r)\), Thompson and Kung also analyze bitonic sort for
shuffled row-major order. Their algorithm takes \((14(n - 1) - \log n)r_c + (2 \log^2 n + \log n)r_r\) time. Their
algorithms require each processor to have only two reg-
isters. They also point out that if \( n \times n \) elements have
already been sorted by some index function, and if each
processor can store \( n \) elements, then the \( N = n^2 \) elements
can be sorted with respect to any other index function using
an additional \( 4(n - 1)r_c \) units of time. Orcutt [7] analyzes
bitonic sort for the case of row-major order. His algorithm
takes \( 0((n \log n)r_r + (\log^2 n)(r_c + r_r)) \) time to sort \( n^2 \)
elements.

In this paper we shall obtain a different adaptation of
bitonic sort for row-major ordering. Our bitonic sort algo-
rithm will require \((14(n - 1) - \log n)r_r + (2 \log^2 n +
\log n)r_c + (4.5 \log^2 n + 1.5 \log n)r_r\) time. If we include the
register interchanges needed by the algorithm of [9], the time
for that algorithm becomes \((14(n - 1) - \log n)r_r + (2 \log^2 n +
\log n)(r_c + 2r_r)\). Hence, our algorithm for row-
major order is almost as fast as that of Thompson and Kung
[9] for shuffled row-major order. Our adaptation is, of
course, faster than that of Orcutt [7] by a factor of \( 0(\log n) \).
However, the algorithm uses more routes and interchanges
than does the \( s^2 \)-way merge algorithm of [9] followed by an
odd-even transposition sort. The importance of the algo-
rithm developed here lies in the fact that bitonic sort is faster
than \( s^2 \)-way merge sort for \( n \leq 512 \) [9]. Hence, while \( s^2 \)-way
merge followed by an odd-even transposition sort will be
faster than our algorithm for large \( n \), it will not be so for
smaller (and perhaps more practical) values of \( n \). Secondly,
[9] states that the row-major indexing scheme is "decidedly"
nonoptimal for bitonic sort. Our adaptation shows that this
statement is inaccurate. Finally, it is worth noting that every
sorting algorithm for a mesh connected machine must result
in at least \( 4(n - 1) \) routes in the worst case [9]. Hence, our
algorithm (as well as those of [9]) is optimal to within a
matrix multiplication on a machine model similar to that
used here.

In Section II, we present our algorithm, and also specify
the sign function to be used in comparison-interchange. In
Section III, we extend our algorithm to the case of a
j-dimensional array processor.

II. ROW-MAJOR BITONIC SORT

Our row-major bitonic sort algorithm is specified as a
series of subalgorithms in algorithmic notation. In analyzing
the algorithms, we shall count only \( N_r \), \( N_h \), and \( N_w \) which
are respectively the number of routes, register interchanges,
and comparison-interchange steps. The analysis will assume
that the number of elements involved is a power of 2. All
logarithms throughout this paper are in base 2.

A. Row Merge

Our first subalgorithm, \( \text{row-merge}(K) \), sorts a bitonic
sequence of size \( K \). The \( K \) elements are in \( K \) adjacent
processors on one row of the \( n \times n \) array.

**procedure** \( \text{row-merge}(K) \)

1) Let \( P_1, \ldots, P_K \) be the processors corresponding to
   the elements
2) if \( K = 1 \) then return
3) shift elements from \( P_{K/2+1}, \ldots, P_K \) respectively to
   \( P_1, \ldots, P_{K/2} \)
4) perform a comparison-interchange on \( P_1, \ldots, P_{K/2} \)
5) shift rejected elements from \( P_1, \ldots, P_{K/2} \) respectively
to \( P_{K/2+1}, \ldots, P_K \)
6) invoke in parallel, \( \text{row-merge}(K/2) \) for \( P_1, \ldots, P_{K/2} \)
   and \( P_{K/2+1}, \ldots, P_K \) (note that this is not a
   recursive call but simply a go to step 1 with \( K \)
   updated).

**end** \( \text{row-merge} \)

The analysis for \( \text{row-merge} \) is

\[
N^r_r(K) = \begin{cases} 
K + N^r_r(K/2) & \text{if } K > 1 \\
0 & \text{if } K = 1 
\end{cases}
\]

\[
N^r_c(K) = \begin{cases} 
1 + N^r_c(K/2) & \text{if } K > 1 \\
0 & \text{if } K = 1 
\end{cases}
\]

If we assume all elements to initially and finally be in the
routing registers then, preceding Step 3, elements in \( P_1, \ldots,
P_{K/2} \) have to be transferred to register \( R_r \). Following Step 5,
elements from \( R_r \) in \( P_1, \ldots, P_{K/2} \) have to be transferred to \( R_c \).
Hence

\[
N^r_r(K) = 2N^r_r(K).
\]

Solving these recurrences, we get (recall \( K \) is a power of 2)

\[ N^r_r(K) = 2K - 2; \ N^r_c(K) = \log K \]

and

\[ N^r_c(K) = 2 \log K. \]

B. Column Merge

Procedure \( \text{column-merge}(K) \) is identical to \( \text{row-
merge}(K) \) except that it sorts a bitonic sequence of \( K \)
elements which are in \( K \) adjacent processors on one column
of the \( n \times n \) array. The analysis is identical to that for \( \text{row-
merge} \). We shall use \( N^c_r(K), N^c_h(K), \) and \( N^c_w(K) \) to denote
the counts.

C. Vertical Merge

Procedure \( \text{vertical-merge}(J, K) \) sorts into either nonin-
creasing or nondecreasing row-major order a \( J \times K \)
array which is made up of two vertically aligned \( J/2 \times K \)
arrays. One of these is in nondecreasing row-major order and
the other is in nonincreasing row-major order.

**procedure** \( \text{vertical-merge}(J, K) \)

1) for all columns in parallel do \( \text{column-merge}(J) \);
2) for all rows in parallel do
   ROW_MERGE(K);
end VERTICAL_MERGE

An example of vertical merge is illustrated in Fig. 1. An arrow indicates a compare-interchange. The head of an arrow points to the processor which retains the larger element.

The correctness of VERTICAL_MERGE may be established by considering the sequence of comparison-interchanges that take place during the bitonic sort of a bitonic sequence \( X = \langle x_1, x_2, \ldots, x_p \rangle \). Unfolding the recursion, we see that if \( p \) is a power of 2 then comparison-interchanges take place in the order

compare-interchange elements \( p/2 \) apart
compare-interchange elements \( p/4 \) apart
compare-interchange elements \( p/8 \) apart

compare-interchange elements 1 apart.

VERTICAL_MERGE begins with a bitonic sequence of \( J \times K \) elements in row-major order. If we look at Step 1 then the following sequence of comparison-interchanges takes place:

compare-interchange elements \( JK/2 \) apart
compare-interchange elements \( JK/4 \) apart

compare-interchange elements \( K \) apart.

Finally, in Step 2 the following sequence is performed:

compare-interchange elements \( K/2 \) apart
compare-interchange elements \( K/4 \) apart

compare-interchange elements 1 apart.

Hence, VERTICAL_MERGE is identical to bitonic sort and so must correctly sort the \( J \times K \) bitonic array. The analysis for VERTICAL_MERGE is

\[
N^{c}_1(J, K) = N^{c}_0(J) + N^{c}_0(K) = 2(J + K) - 4
\]
\[
N^{c}_1(J, K) = N^{c}_0(J) + N^{c}_1(K) = \log(JK)
\]
\[
N^{c}_1(J, K) = N^{c}_1(J) + N^{c}_1(K) = 2 \log(JK).
\]

D. Horizontal Merge

In this section we give an algorithm to sort a \( J \times K \) array which is made up of two horizontally aligned and adjacent \( J \times K/2 \) arrays. One of these is already sorted in nondecreasing row-major order while the other is in nonincreasing row-major order. But first we give an algorithm, TWO_COLUMN_MERGE, to sort a bitonic sequence \( < a_0, a_1, \ldots, a_{2^i-1} > \) initially loaded in a column of \( J \) processors \( < P_0, P_1, \ldots, P_{J-1} > \) such that \( P_i \) contains \( a_i \) and \( a_{i+J}, 0 \leq i < J \).

If the sorted sequence is \( < b_0, b_1, \ldots, b_{2^i-1} > \) then at termination, processor \( P_i \) contains elements \( b_{2i} \) and \( b_{2i+1} \).

procedure TWO_COLUMN_MERGE(J)
1) Let \( P_0, P_1, \ldots, P_{J-1} \) be the \( J \) processors
2) Compare-interchange the elements in each processor
end TWO_COLUMN_MERGE

3) if \( J > 1 \) then
   a) Exchange the rejected elements of \( P_0, \ldots, P_{J/2-1} \) with the accepted elements of \( P_{J/2}, \ldots, P_{J-1} \);
   b) In parallel perform TWO_COLUMN_MERGE(J/2) on the processors \( P_0, \ldots, P_{J/2-1} \) and \( P_{J/2}, \ldots, P_{J-1} \);
end TWO_COLUMN_MERGE

Fig. 2 illustrates the working of TWO_COLUMN_MERGE on an example.

The correctness of TWO_COLUMN_MERGE may be established using an argument similar to that used for VERTICAL_MERGE. Analyzing the number of steps, we obtain

\[
N^c_1(J) = \begin{cases} 
  J + N^c_1(J/2) & \text{if } J > 1 \\
  0 & \text{if } J = 1
\end{cases}
\]
\[
= 2J - 2
\]
\[
N^c_1(J) = \begin{cases} 
  1 + N^c_1(J/2) & \text{if } J > 1 \\
  1 & \text{if } J = 1
\end{cases}
\]
\[
= 1 + \log J.
\]

Register interchanges are needed in Step 3a) to exchange rejected and accepted elements. This can be done with three register interchanges: first move the rejected elements on \( P_{J/2}, \ldots, P_{J-1} \) to \( R \); next route the rejected elements from \( P_0, \ldots, P_{J/2-1} \) to \( P_{J/2}, \ldots, P_{J-1} \); now interchange between \( R \) and \( R \) on \( P_{J/2}, \ldots, P_{J-1} \); route from \( P_{J/2}, \ldots, P_{J-1} \) to \( P_0, \ldots, P_{J/2-1} \); finally move from \( R \) to \( R \) on \( P_{J/2}, \ldots, P_{J-1} \). Hence

\[
N^c_1(J) = \begin{cases} 
  3 + N^c_1(J/2) & \text{if } J > 1 \\
  0 & \text{if } J = 1
\end{cases}
\]
\[
= 3 \log J.
\]

We are now ready for the horizontal merge algorithm.

procedure HORIZONTAL_MERGE(J, K)
1) Let the \( K \) columns be \( C_1, C_2, \ldots, C_K \)
2) Move in parallel elements from the \( J \) processors in each of the columns \( C_{K/2+1}, \ldots, C_K \) to the corresponding processors in the columns \( C_1, C_2, \ldots, C_{K/2} \) respectively
3) For each of the columns \( C_1, C_2, \ldots, C_{K/2} \) perform in parallel, TWO_COLUMN_MERGE(J)
4) Move, in parallel, the rejected elements back to the processors in \( C_{K/2+1}, \ldots, C_K \)
5) if \( K > 2 \) then invoke in parallel ROW_MERGE(K/2) for each of the \( 2J \) rows of size \( K/2 \) // note: \( 2J \) rows, each containing \( K/2 \) adjacent processors, are obtained by splitting each of the original \( J \) rows into two. //
end HORIZONTAL_MERGE
Figs. 3 and 4 illustrate the working of HORIZONTAL_MERGE. The correctness of the algorithm follows from an argument similar to that used for VERTICAL_MERGE.

The number of routing steps is given by

\[ N^H_r(J, K) = K/2 + N^I_r(J) + K/2 + N^R_r(K/2) \]
\[ = 2(J + K) - 4 \]

For the number of comparison-interchanges, we get

\[ N^I_r(J, K) = N^I_r(J) + N^R_r(K/2) + 2 \]

Substituting, we get

\[ N^I_r(J, K) = 3 \log J + 2 \log K \]

The number of register interchanges is

\[ N^R_r(J, K) = N^I_r(J) + N^R_r(K/2) + 2 \]

where the 2 comes from Steps 2 and 4.

Having defined the subalgorithms, we now give the main procedure, sort, that will sort \( n^2 \) elements into nondecreasing row-major order. This algorithm also defines a pass number \( S \) which will be used (as explained in the next section) to determine how comparison-interchanges are to be performed.

**procedure** sort\((n, n)\)
1. \( K \leftarrow S \leftarrow 1 \)
2. **while** \( K < n \) **do**
   a) Consider the \( n \times n \) processor array as composed of many adjacent \( K \times 2K \) subarrays
   b) do in parallel for each \( K \times 2K \) array HORIZONTAL_MERGE\((K, 2K)\)
   c) \( S \leftarrow S + 1 \)
   d) Consider the \( n \times n \) processor array as composed of many adjacent \( 2K \times 2K \) subarrays
   e) do in parallel for each \( 2K \times 2K \) array VERTICAL_MERGE\((2K, 2K)\)
   f) \( S \leftarrow S + 1; K \leftarrow 2 \times K \)
**end**
**end sort**

The total number of routing steps is

\[ N^I(n, n) = N^I(n/2, n/2) + N^H_r(n/2, n) + N^R_r(n, n) \]
\[ = N^I(n/2, n/2) + 7n - 8, \quad n > 1 \]

and

\[ N^I(1, 1) = 0. \]

Hence, \( N^I(n, n) = 14(n - 1) - 8 \log n. \) The number of comparison-interchanges is

\[ N^c(n, n) = N^c(n/2, n/2) + N^I_r(n/2, n) + N^R_r(n, n) \]
\[ = N^c(n/2, n/2) + 4 \log n - 1 \]
\[ = 2 \log^2 n + \log n. \]

The number of register-interchanges is

\[ N^r(n, n) = N^r(n/2, n/2) + N^I_r(n/2, n) + N^R_r(n, n) \]
\[ = N^r(n/2, n/2) + 5 \log n - 3 + 4 \log n \]
\[ = N^r(n/2, n/2) + 9 \log n - 3 \]
\[ = 4.5 \log^2 n + 1.5 \log n < 2.25 N^I(n, n). \]

**F. The SIGN Function**

In order for procedure sort to work correctly, it is necessary that the \( K \times 2K \) and \( 2K \times 2K \) subarrays being sorted in Steps 2b) and 2c) satisfy the initial conditions of HORIZONTAL_MERGE and VERTICAL_MERGE, respectively. In order to meet these conditions, it is necessary to sort some of the subarrays into nonincreasing order and others into nondecreasing order. The order into which a subarray gets sorted is determined by the sign function used during a comparison-interchange. Recall that the comparison-interchange instruction was defined in Section I to be

\[ \text{IF} \ \text{SIGN}(I, S) \ast (R_i - R_j) < 0 \ \text{then} \ \text{interchange} \ (R_i, R_j). \]

If during the sort of a \( K \times 2K \) (or \( 2K \times 2K \)) subarray SIGN is +1 for all processors on which comparison-interchanges are made, then the \( K \times 2K \) (or \( 2K \times 2K \)) subarray will be sorted into nondecreasing order. If the SIGN is -1, then the subarray will be sorted into nonincreasing order. One may easily verify that the following SIGN function will serve our purpose:
procedure \textbf{SIGN}(SI, S)
\begin{verbatim}
//SI = shuffled row-major index of processor
(as explained in Section I)/
//S = pass number defined in SORT//
If \{\text{SI}/2^3\} is even then return (+1)
else return (−1)
\end{verbatim}
end \textbf{SIGN}

Thus, each processor can determine the \textbf{SIGN} for its comparison-interchange if "S" is broadcast to all processors.

Fig. 5 illustrates the working of procedure \textbf{SORT} on a 4 × 4 mesh-connected computer. The "T" operation, when \( S = 3 \), represents a two-column merge. Four pairs of 2 × 1 columns are merged in parallel.

III. Extensions

Procedure \textbf{SORT} is easily modified to sort into snake-like row-major order. Only the \textbf{SIGN} function needs to be changed for the last \textbf{VERTICAL-MERGE}, i.e., the call \textbf{VERTICAL-MERGE}(n, n). During this call, \textbf{SIGN} is to be altered only when \textbf{ROW-MERGE} is invoked from Step 2 of \textbf{VERTICAL-MERGE}. This alteration is such that the \textbf{SIGN} for odd rows becomes −1 and remains +1 for even rows (recall rows are indexed 0 through \( n − 1 \)).

Following along the lines of Thompson and Kung [9], we may extend our row-major bitonic sort to the case of a \( j \)-dimensional array processor. We now have \( N = n^j \) processors arranged as in a \( n \times n \times \cdots \times n \) \( j \)-dimensional array. Each processor is connected to all of its neighbors. As before, the number of elements to be sorted is \( N \) and each processor is assumed to have three registers. For this extension, we define \textbf{LINEAR-MERGE}(i, K) to be identical to \textbf{ROW-MERGE}(K) of \textbf{COLUMN-MERGE}(K) except that the \( K \) elements are on the \( i \)-th axis of the \( j \)-dimensional array, \( 1 \leq i \leq j \). We also define \textbf{TWO-COLUMN-MERGE}(i, K) to be the same as the corresponding algorithm for a two-dimensional array except the "column" is now the \( i \)-th axis. When \( K = 1 \), this algorithm is modified to do nothing. This will avoid redundant comparisons in the following algorithm. Procedure \textbf{MERGE} will merge along the \( i \)-th axis two subarrays of size \( K_j \times \cdots \times K_j/2 \times \cdots \times K_1 \) to result in an array of size \( K_j \times \cdots \times K_j \times \cdots \times K_1 \) sorted in row-major.

procedure \textbf{MERGE}(i, K_j, \ldots, K_j, \ldots, K_1)
\begin{verbatim}
1) Move elements from the second subarray to corresponding processors in the first subarray
2) for \( A = j, j − 1, \ldots, i + 1 \) do
   \textbf{TWO-COLUMN-MERGE}(A, K_A)
3) compare-interchange elements
4) move rejected elements back to corresponding processors in the second subarray
5) \textbf{LINEAR-MERGE}(i, K_i/2)
6) for \( A = i − 1, i − 2, \ldots, 1 \) do
    \textbf{LINEAR-MERGE}(A, K_A)
end \textbf{MERGE}
\end{verbatim}

Note that the "for" loops of Steps 2 and 6 are done sequentially for each value of \( A \). One may verify that for \( j = 2 \), procedure \textbf{MERGE} reduces to \textbf{HORIZONTAL-MERGE} when

\[ i = 1 \] and to \textbf{VERTICAL-MERGE} when \( i = 2 \). The number of routing steps is

\[ N_r^M(K_j, \ldots, K_1) = 2(K_1 + K_2 + \cdots + K_j) − 2j. \]

The sorting algorithm for \( N = n^j \), then, is recursively defined as

procedure \textbf{JSORT}(n^j)
\begin{verbatim}
1) \textbf{JSORT}(n^j/2);
2) \textbf{MERGE}(1, n/2, n/2, \ldots, n/2, n)
3) \textbf{MERGE}(2, n/2, \ldots, n/2, n, n)
\ldots
j + 1) \textbf{MERGE}(j, n, n, \ldots, n)
end \textbf{JSORT}
\end{verbatim}

The "sign" of comparison is determined by a simple extension of the method of Section II-F.

The total number of routing steps will be

\[ N_r^J(n^j) = N_r^M(n^j/2^j) + \sum_{i=1}^{j} (2ni + n(j − i)) − 2j^2 \]

which gives

\[ N_r^J(n^j) = (3j^2 + j)(n − 1) − 2j \log N. \]

(This is the same number of routing steps as in [9] for the shuffled indexing.)

The number of compare-interchange steps \( N_r^J \) is invariant to the interconnection scheme, is the same as for \textbf{SORT} and, the number of register-interchange steps \( N_r^J \) will still be less than \( 3N_r^J \).

It is interesting to consider the case of maximal connecti-
vity (with respect to the bitonic sort algorithm), where
N = 2^\log N \text{ processors are interconnected}
(lo=log N)-dimensionally. Then, each processor would be con-
ected to exactly log N other processors. Upon substituting
n = 2 and j = log N, the number of routing steps is reduced to
N' = \log^2 N + \log N = 2N'.

This is as expected as every pair of processors involved in a
compare-interchange will be adjacent. Stone's "perfect-
shuffle" network [7] also sorts N elements in O(\log^2 N) time.
His network uses a far smaller processor connectivity than
log N.

IV. Conclusions

We have shown that bitonic sort can be adapted to sort n^2
elements into row-major order on an n x n mesh-connected
computer in O(\log n) time. This is an improvement over Orcutt's
[7] adaptation which requires O(n \log n) time. Our al-
gorithm makes the same number of routes and comparison-
interchanges as does that of Thompson and Kung [9]. Their
algorithm, however, obtains a shuffled row-major order.
Thus, row-major order is not "decidedly" nonoptimal for
bitonic sort as claimed in [9]. Our algorithm needs about
12.5 percent more register-interchanges than does that of [9].
These are, however, much cheaper than "routes." Our
algorithm for row-major bitonic sort can be extended to
snake-like row-major ordering and also to sorting on a
j-dimensionally connected computer. In the latter case, the
algorithm requires as many routes and comparison-
interchanges as does that of [9].

REFERENCES

1978.
speed computations," Ph.D. dissertation, Stanford Univ., Stanford,
CA, Sept. 1974, ch. 2.
Apr. 1977.