A New Parallel Sorting Algorithm and its Efficient VLSI Implementation

S. DEY AND P. K. SRIMANI*
Department of Computer Science, Southern Illinois University, Carbondale, IL 62901-4511, USA

In this paper we develop a new parallel algorithm for sorting which has a time complexity of O(log n) and requires $n^2/\log n$ processors. The algorithm can be readily mapped on an SIMD mesh connected array of processors which has all the features of efficient VLSI implementation. The corresponding hardware algorithm maintains the O(Log n) execution time and has a low O(n) interprocessor communication time.

Received November 1987, *revised February* 1990

1. INTRODUCTION

The advent of parallel architectures has prompted the design of numerous parallel algorithms using various models of computation. Most of these algorithms strike a tradeoff between the reduction of time complexity and the number of processors used. However, with the recent advances in VLSI technology, it has become technically possible and economically feasible to build parallel computers with thousands of processors. This hardware availability has made it possible to design parallel algorithms cutting down the execution time as the primary objective and then optimizing the number of processors as a secondary objective. Also, more emphasis is recently being placed on designing VLSI implementable parallel algorithms rather than algorithms using more theoretical models which are not readily VLSI implementable. And when an algorithm is to run on a chosen architecture, the total data communication time needed by the algorithm becomes as equally important a criterion to judge the complexity of the algorithm as the inherent execution time of the algorithm.

Because of its practical importance in the field of computer science, there has been a flurry of research efforts towards developing parallel algorithms for sorting [7]. The serial tree selection sort algorithm can be parallelized and executed on a $(2n-1)$ processor tree machine in $O(n)$ time.[6] Various sorting networks have been proposed which implement Batcher's odd-even and bitonic sort algorithms,[8] illustrative of which is Stone's bitonic sorting network with $n/2$ processors which requires $O(\log^2 n)$ time to sort n numbers.[5] These network sorting algorithms have also been adapted to the SIMD mesh connected model of computation, requiring n processors and $O(n)$ comparison and move steps.[9] Kumar *et al.*[10] proposed an improved version of parallel merging by which n^2 numbers, initially stored in the local memories of n^2 processors organized as an SIMD mesh connected machine, can be sorted in time $O(n)$ requiring a smaller proportionality constant than.[9] More recently, VLSI implementable sorting networks have been proposed where sorting is done in $O(\log n)$ time with $O(n^2)$ chip area, illustrative of which is the VLSI network proposed by Bilardi *et al.*[12] However, the above network[12] assumes that the words are restricted to a size of $(1+\varepsilon)\log n$ $(\varepsilon > 0)$. It is also to be noted that another $O(\log n)$ execution time algorithm was reported in Ref. 14 which uses $O(n \log n)$ processors. But author in Ref. 14 didn't consider any implementation aspects and the algorithm has not been shown to be VLSI implementable. In fact the implementations described in Ref. 12 use more or less the same concept as introduced in Ref. 14.

Sorting algorithms have also been proposed for the SIMD shared memory model. By using the fast merging technique that he developed, Valiant[2] showed that the merge sort algorithm can be parallelized to execute in $O(\log n \cdot \text{loglog } n)$ time using n processors; however, his comparison model is essentially theoretical and is not VLSI implementable. Moreover, his model considers only the time taken to perform comparisons, and all other computational overheads including data communication time are completely ignored. Hirschberg's bucket sort algorithm[11] sorts n numbers in $O(\log n)$ time using n processors. This algorithm also suffers from drawbacks. The numbers to be sorted have to be in the range $\{0..m-1\}$, and to avoid memory contention problems, m arrays, each of size n, are required. Then, to accommodate duplicate numbers in the given sequence, the time and processor requirements of the algorithm increase to $O(k \log n)$ and $n^{1+1/k}$ respectively, where k is an arbitrary integer. Moreover, the SIMD shared memory architecture is not a good candidate for VLSI implementation, because of unbounded nature of the memory connections needed.

In this paper, we develop a VLSI implementable parallel algorithm to sort a set of n given numbers. Our algorithm has a comparison time complexity of $O(\log n)$ and requires $n^* \lceil n/\log n \rceil$ processors. It can be readily mapped on an SIMD mesh connected array of processing elements[4] which has all the features of efficient VLSI implementation. The hardware algorithm maintains the $O(\log n)$ execution time and has $O(n)$ interprocessor communication time.

In section 2, we present our $O(\log n)$ algorithm along with an analysis of its time and processor requirements. In section 3, we map our algorithm on a SIMD mesh connected array of n by $\lceil n/\log n \rceil$ homogenous pro-

* Address for Correspondence: Pradip K. Srimani, Department of Computer Science, Colorado State University, Ft. Collins, CO 80523, USA.

cessors. The architecture resembles the Illiac-IV architecture[3] minus its end-around connections. It is also to be noted that the proposed hardware algorithm can be easily mapped on presently available real-life SIMD array machines like MPP.[13] We analyze the performance of our hardware algorithm in terms of its execution time complexity and its interprocessor communication time complexity. Section 4 concludes the paper.

2. THE PARALLEL ALGORITHM SORT

We assume that the given set of n elements is stored in an array A. We sort the elements in non-increasing order and store them in array B. It should be noted that the proposed algorithm works correctly even in presence of duplicate elements. We present below the parallel algorithm SORT which sorts the elements of given array A in a non-increasing order, where array A may contain duplicate elements. Throughout this paper, m is used as a global constant set to the value $\lceil n/\log n \rceil$.

2.1 Algorithm SORT

Input: Array A of n elements;
Output: Array B, which contains the elements of A sorted in a non-increasing order.
Data Structures: A, $B = array\ [1..n]$;

$COUNT = array\ [1..n, 1..m]$;

```
1.  begin
2.    for i = 1 to n do in parallel
3.      begin
4.        for k = 1 to ⌈log n⌉ do
5.          begin
6.            x = (k − 1).m;
7.            for j = (x + 1) to min (n, x + m)
                do in parallel
8.              if A[i] < A[j] then COUNT[i,
                j − x]:=
                  COUNT[i, j − x] + 1;
9.          end;
10.       for p = 0 to (⌈log m⌉ − 1) do
11.         for k = 2^p + 1 in steps of 2^(p+1) to m do in
            parallel
12.           COUNT[i, k − 2^p]:= COUNT[i, k] +
              COUNT[i, k − 2^p];
13.       B[i] := 0;
14.       B[COUNT[i, 1] + 1] := A[i];
15.     end;
16.   SCAN;
17. end.
```

Procedure SCAN;

```
p1.  begin
p2.    for p = 0 to (⌈log n⌉ − 1) do
p3.      for i = 2^p in steps of 2^(p+1)
         to (n − 1) do in parallel
p4.        for k = (i + 1) to min {n, 2^p}
           do in parallel
p5.          if B[k] = 0 then B[k] := B[i];
p6.  end;
```

We assume the existence of a function $min(x, y)$ which returns the minimum of two elements x and y. We give below a brief description of the algorithm SORT. Lines 7–8 compare element $A[i]$ with m ($\leq \lceil n/\log n \rceil$) elements of array A. Executing the loop 5–9 $\lceil \log n \rceil$ times enables the comparison of $A[i]$ with all the n elements of array A and, at the end, $COUNT[i, q]$ gives the number of elements which are greater than $A[i]$ among the elements $(A[q], A[q + m], A[q + 2m], \ldots)$. Thus, $COUNT[i, q]$ gives the total number, t, of elements of array A which are greater than $A[i]$. This is done in lines 10–12, and then $COUNT[i, 1]$ contains t. Line 14 puts $A[i]$ in $B(t + 1]$ which is its proper position in a non-increasing order of the elements of A. This entire process is carried out in parallel for all the elements of A (lines 2–15). However, since there can be more than one element having the same value, more than one element may be mapped to the same position of array B. In general, if there are y elements with value v and there are t elements which have value greater than v, then $B[t + 1]$ is set to v, and $B[t + 2]$, $B[t + 3], \ldots,$ $B[t + y]$ remain unchanged (i.e. contain 0). For instance, all the three elements with value 5 in array A in Fig. 1 are mapped to $B[4]$ (Fig. 2(a)) since each of them has 3 elements greater than itself, and $B[5]$ and $B[6]$ are set to 0. In this example, v is 5, y is 3 and t is 3.

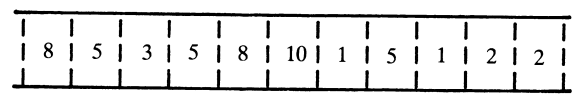


Figure 1a. The array A ($n = 11$).

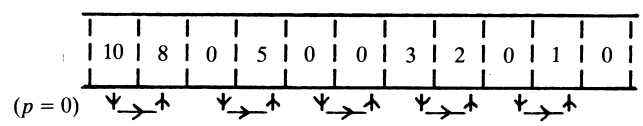


Figure 2(a)

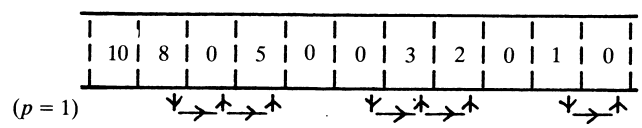
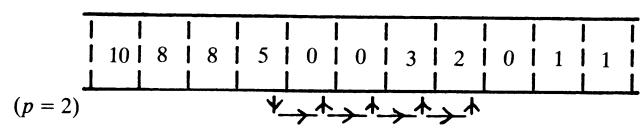
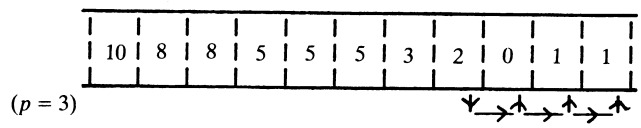


Figure 2(b)





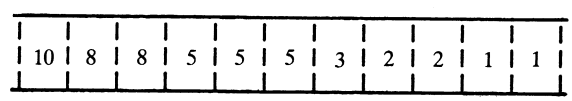


Figure 2(c)

Figure 2. This shows the steps of procedure Scan on the array B. (a) The array B at the start of procedure Scan. (b) The array B after the first step (after $p = 0$). (c) The final array B at the end of procedure Scan. '→—' shows transfer of data and its direction.

At this point, the distinct elements of array A have been arranged in array B in a non-increasing order. However, the locations $B[t+2]$ to $B[t+y]$ need to be filled up with the value v from $B[t+1]$, for any group of y elements with the same value v. This is exactly what procedure SCAN does. The operation done on any chosen pair of elements, $B[i]$ and $B[k]$, is replacing $B[k]$ by $B[i]$ if $B[k] = 0$. The pair of elements is chosen as follows. At the p-th step, $B[i]$ is operated in parallel with $B[i+1]$, $B[i+2]$,..., $B[z]$, where $z = \min(n, i+2^p)$, and this is done in parallel for all values of $i = 2^p, (2^p + 2^{p+1}),\ldots$, (largest value of i does not exceed $n-1$). After procedure SCAN has been executed, B contains the elements of A sorted in a non-increasing order.

Example: Fig. 2(a)-(c) trace the steps of procedure SCAN. Figs. 2(a) and 2(c) show the array B at the beginning and the end of procedure SCAN respectively.

2.2 Time and Processor requirements of algorithm SORT

We first analyze the time and number of processors required by the procedure SCAN. At each step p of the outer loop, number of possible values of i is $\lceil (n-1)/2^{p+1} \rceil$, and for each value of i, maximum number of possible values of k is 2^p. Thus at each step p of the outer loop, the maximum number of distinct computations that need be done is $((n-1)/2^{p+1})*2^p = \lceil (n-1)/2 \rceil$. Since all these computations are done in parallel, $\lceil n/2 \rceil$ processors suffice to execute the loop p3 – p5 in unit time. This loop is executed sequentially $\lceil \log n \rceil$ times, and hence procedure SCAN requires $\lceil \log n \rceil$ time units and $\lceil n/2 \rceil$ processors.

Now we can analyze the time and processor requirements of algorithm SORT. In lines 7–8, an element $A[i]$ is compared with m elements of array A in unit time using m processors. To complete comparing $A[i]$ with all the n elements of A, the above step is repeated $\lceil \log n \rceil$ times. Consequently, the loop 4–9 takes $2\lceil \log 2n \rceil$ time and $m/2$ processors. Elements of the i-th row of matrix $COUNT$, $COUNT[i, 1]$ to $COUNT[i, m]$, can be added up in $\lceil \log m \rceil$ time, using m processors (lines 10–12). Thus, an element of array A is processed and placed in its proper position in array B in $(3\lceil \log n \rceil - \lceil \log\log n \rceil + 2)$ time units using m processors. Since each of the elements of array A can be processed simultaneously in this way, lines 2–15 also take $(3\lceil \log n \rceil - \lceil \log\log n \rceil + 2)$ time and require n^*m processors. Consequently, considering the time required by procedure SCAN, algorithm SORT takes $(4\lceil \log n \rceil - \lceil \log\log n \rceil + 2)$, or, $O(\log n)$ time, and requires n^*m or $O(n^2/\log n)$ processors.

3. HARDWARE IMPLEMENTATION OF ALGORITHM SORT

In this section, we map algorithm SORT on a suitable VLSI architecture, maintaining the $O(\log n)$ execution time and minimizing the interprocessor communication time. We use an SIMD array of $n^* \lceil n/\log n \rceil$ homogeneous processors with bidirectional interprocessor communication links forming a mesh, as shown in Fig. 3. The array contains n rows and m $(= \lceil n/\log n \rceil)$ columns of processors. For notational convenience, we

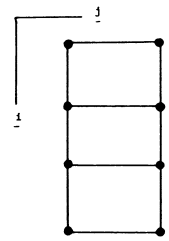


Figure 3. Processor Organization for $n = 4$.

will refer to the j-th processor in the i-th row as P_{ij}. Each processor P_{ij} has the following internal registers: A_{ij}, X_{ij}, Y_{ij} and F_{ij}.

The processors are required to have the following instruction set. As in SIMD array processors, each single instruction is executed in parallel on a set of multiple data as is mentioned below with each instruction specification. In the instruction specifications given below, R,SR and DR are register parameters to specify a Register, Source Register, and Destination Register respectively, and these can be any one of the A, X, Y or F registers of the processors. Similarly, *NAME* represents an array parameter.

1. Load-horiz *NAME* (s,t):
   for $i = 1$ **to** n **do in parallel**
   for $j = 0$ **to** $(m-1)$ **do in parallel**
   if $j \leq (t-2)$ **then** $\{Y_{ij} \leftarrow NAME[s+j]$
   $F_{ij} \leftarrow 1\}$
   else $F_{ij} \leftarrow 0$;

This instruction assumes that $(t-s+1)$, the number of elements of array *NAME* to be loaded to the Y registers, is not greater than m, the number of processors available in each row. For a given value of s and t, this instruction loads $NAME[s]$, $NAME[s+1]$,..., $NAME[t]$ to the Y registers of processors $P_{i1}, P_{i2},\ldots, P_{i,t-s+1}$, respectively and sets the F registers of these processors to 1, for all i, $1 \leq i \leq n$. If $m > (t-s+1)$, the Y registers of processors $P_{i,t-s+2}$, $P_{i,t-s+3},\ldots, P_{i,m}$ are unaffected, and their F registers are set to 0, for $1 \leq i \leq n$.

2. Load-vert *NAME*: $X_{ij} \leftarrow NAME[i]$; $1 \leq i \leq n$, $1 \leq j \leq m$;

This instruction loads $NAME[i]$ to the X register of processor P_{ij}, $1 \leq i \leq n$, $1 \leq j \leq m$.

3. Clear R: $R_{ij} \leftarrow 0$, $1 \leq i \leq n$, $1 \leq j \leq m$;

This instruction clears the contents of the specified register R of all processors P_{ij}, $1 \leq i \leq n$, $1 \leq j \leq m$.

4. Compare: **if** $X_{ij} < Y_{ij}$ **then** $Y_{ij} \leftarrow 1$ **else** $Y_{ij} \leftarrow 0$, $1 \leq i \leq n$, $1 \leq j \leq m$;

The contents of X and Y registers are compared. If the content of X is less than Y, then the Y register is set to 1, otherwise it is set to 0. This is done for all $1 \leq i \leq n$, $1 \leq j \leq m$.

5. Equal: **if** $F_{ij} = 1$ **then** {**if** $X_{ij} <> Y_{ij}$ **then** $F_{ij} \leftarrow 0$}, $1 \leq i \leq n$, $1 \leq j \leq m$;

If the register F is 1, then it is made 0 only if the contents of X and Y registers are not equal, for all i, j.

6. Transfer (DR, SR): $DR_{ij} \leftarrow SR_{ij}$, $1 \leq i \leq n$, $1 \leq j \leq m$;

This instruction transfers the contents of the specified source register SR to the specified destination register DR, in all the processors P_{ij}, for $1 \leq i \leq n$, $1 \leq j \leq m$;

7. Cond-Transfer (DR, SR): **if** $F_{ij} = 1$ **then** $DR_{ij} \leftarrow SR_{ij}$, $1 \le i \le n$, $1 \le j \le m$;
If register F is 1, then the contents of the specified source register SR is transferred to the specified destination register DR, in all the processors P_{ij}, for $1 \le i \le n$, $1 \le j \le m$.
8. Move-horiz (R, q):$\{R_{ij} \leftarrow R_{i,j+q}, F_{ij} \leftarrow 1\}$; $1 \le i \le n$, $1 \le j \le q$;
For all i, j such that $1 \le i \le n$ and $1 \le j \le q$, this instruction moves the contents of the specified register R of processor $P_{i,j+q}$ to the register R of P_{ij} and then sets F_{ij} to 1. Note that whenever this instruction is used, we will have $2q \le m$. It is also to be noted that the execution time of this instruction is q time units since any value has to move via q processors to reach the destination and then all the moves for different values of i and j are done in parallel.
9. Move (R, q, j): **for** $i = q$ **in steps of** $2q$ **to** $(n-1)$ **do in parallel**

$$R_{x,j} \leftarrow \ldots \leftarrow R_{i+2,j} \leftarrow R_{i+1,j} \leftarrow R_{i,j}$$
$$F_{x,j} \leftarrow \ldots \leftarrow F_{i+2,j} \leftarrow F_{i+1,j} \leftarrow F_{i,j} \leftarrow 1,$$
where $x = \min(i+q, n)$;

For a given value of j, the content of the R register of P_{ij} is stored in the R registers of each of the processors $P_{i+1,j}, P_{i+2,j}, \ldots, P_{x,j}$, where $x = \min(i+q, n)$, and the F registers of these processors are also set to 1. This is done in parallel for $i = q, 3q, 5q, \ldots$ etc. (largest value of i should not exceed $n-1$).
10. Add R: **if** $F_{ij} = 1$ **then** $A_{ij} \leftarrow A_{ij} + R_{ij}$, $1 \le i \le n$, $1 \le j \le m$;
If register F is 1, then Register A is set to the sum of the A register and the specified register R, for all i, j, $1 \le i \le n$, $1 \le j \le m$.
11. Inc: $A_{ij} \leftarrow A_{ij} + 1$, $1 \le i \le n$, $1 \le j \le m$;
The contents of the A registers of all the processors P_{ij} are incremented by 1, for $1 \le i \le n$, $1 \le j \le m$.
12. Store1 ($NAME$, q):$NAME[i] \leftarrow X_{iq}$, $1 \le i \le n$;
This instruction stores the contents of the X registers of the q-th column of processors in the corresponding locations of the array $NAME$.
13. Store2 ($NAME$, q):$NAME[A_{iq}] \leftarrow X_{iq}$, $1 \le i \le n$;
Here, the contents of the X register of processor P_{iq} is stored in $NAME[A_{iq}]$. This is done in parallel for all the processors P_{iq}, $1 \le i \le n$, for a given value of q.

Note that more than one processor may want to store data simultaneously in the same location, say when $A_{i1,q} = A_{i2,q}$ and $i1 = i2$. However, this instruction is used in the following algorithm in such a way that whenever more than one processor want to store in the same location, all of them want to store the same value. This is easily resolved by using a memory arbiter, allowing arbitrarily one of these processors to write and aborting the requests of the rest. Consequently, the time required by this instruction is constant.

Having described the instruction set required by the processing elements of the architecture, we now present a hardware algorithm, which is, indeed, a mapping of algorithm SORT on the SIMD mesh connected array architecture. The set of n elements is initially stored in the array $AR1$, and at the end of the algorithm, $AR2$ contains the elements sorted in a non-increasing order. That is, arrays $AR1$ and $AR2$ correspond to arrays A and B of algorithm SORT.

3.1 The Hardware Algorithm H-SORT:

```
0.  begin
1.    Clear X;
2.    Store1 (AR2, 1);
3.    Load-vert AR1;
4.    Clear A;
5.    for k = 1, m + 1, 2m + 1, 3m + 1,...,
        (⌈log n⌉ −1)*m + 1 do
6.      begin
7.        Load-horiz AR1[k, min(m, k + m − 1)]
8.        Compare;
9.        Add Y;
10.     end;
11.   FIND-COUNT;
12.   Inc;
13.   Store2 (AR2, 1);
14.   Load-vert AR2;
15.   H-SCAN;
16.   Store1 (AR2, 1)
17.   end.
```

Procedure FIND-COUNT;

```
1a. for p = ⌈m/2⌉, ⌈m/4⌉, ⌈m/8⌉,..., 1 do
1b.   begin
1c.     Clear F;
1d.     Transfer (Y, A);
1e.     Move-horiz (Y, p);
1f.     Add Y
1g.   end;
```

Procedure H-SCAN;

```
2a. for p = 2^0, 2^1, 2^2, 2^3,.... 2^(⌈log n⌉ −1) do
2b.   begin
2c.     Clear F;
2d.     Clear Y;
2e.     Transfer (A, X);
2f.     Move (A, p, 1);
2g.     Equal;
2h.     Cond-Transfer (X, A)
2i.   end;
```

3.2 Description of algorithm H-SORT

We give below an informal description of algorithm H-SORT. At first, the array $AR2$ is initialized to 0 (lines 1–2). The elements of given array $AR1$ are loaded in such a way that the X registers of all the m processors of the i-th row of the processor array contain $AR1[i]$, $1 \le i \le n$ (line 3). At the p-th iteration of the loop 5–10, m elements of array $AR2$, $AR2[(p-1)*m+1]$ thru $AR2[p*m]$, are loaded onto the m Y registers of the i-th row, compared with the corresponding X registers (i.e. $AR1[i]$), and register A_{ij} is incremented if $X_{ij} > Y_{ij}$ (i.e. $AR1[i] > AR1[(p-1)*m+j]$), for $1 \le i \le n$. However, at the last step ($\lceil \log n \rceil$-th step), less than m elements may be left to be compared with $AR1[i]$, and consequently, the *Add* operation at line 9 is not executed for all the processors. After execution of the loop 5–10 $\lceil \log n \rceil$ times, A_{ij} stores the same value as $COUNT[i, j]$ does after execution of the loop 4–9 of algorithm SORT. Thus, A_{iq} gives the total number, t, of elements of array $AR1$ which are greater than $AR1[i]$, for $1 \le i \le n$. This is done by procedure *FIND-COUNT*, and after its execution A_{i1} has this value t, for $1 \le i \le n$. Lines 12–13 perform the action of line 14 of algorithm

SORT, storing the distinct elements of $AR1$ in a non-increasing order in the array $AR2$.

However, at this point, the duplicate elements should also be placed in their proper places in the array $AR2$. This is done by procedure *H-SCAN* which exactly simulates the actions of procedure *SCAN*. Lines 2g–2h, together with line 2d, simulate the action of line p5 of procedure *SCAN*. It should be noted that at each step of the **for** loop of procedure *H-SCAN*, the F registers are cleared to prevent certain processors from executing the *Equal* instruction in line 2f. Also, the propagation of the data elements is done in such a way that at any instant of time any communication link between any two procesors need not pass more than one data element. Hence, data from all the registers A_{i1} for specified values of i, can be routed to their destinations in parallel without any data conflict and ensuring minimal communication time. After $\lceil \log n \rceil$ sequential executions of the loop 2b–2i of procedure *H-SCAN*, X_{11} through X_{n1} contain the elements of the given array $AR1$ such that $X_{11} \geq X_{21} \geq \ldots \geq X_{n1}$. Finally, line 16 stores the elements X_{11} through X_{n1} to the locations $AR2[1]$ thru $Ar2[n]$. Consequently, at the end of the algorithm H-SORT, $AR2$ contains the elements of the given array $AR1$ sorted in a non-increasing order.

3.3 Time Analaysis of Algorithm H-SORT

We analyze below the execution time as well as the communication time required by the hardware algorithm HKL. Let t_e be the execution time of the Clear, Compare, Equal, Add, Inc and Transfer instructions, and let t_s be the memory access time for the Load/Store type instructions. Also, let t_r be the data communication time between any two adjacent processors. Then the time required to execute the *Move* (R, q, j) and *Move-horiz* (R, q) instructions is $q.t_r$ each.

We first compute the time required to execute procedure *FIND-COUNT*. Lines 1c, 1d and 1f require t_e time each, while line 1e requires $p.t_r$ time. Thus a single execution of the loop 1b–1g requires $(3t_e + p.t_r)$ time, and this loop is executed $\lceil \log m \rceil$ times. Consequently, the time taken by procedure *FIND-COUNT* is $3\lceil \log m \rceil .t_e + (\lceil m/2 \rceil + \lceil m/4 \rceil + \ldots + 1).t_r$, or, $3(\lceil \log m \rceil .t_e) + (m + \log m).t_r$ at maximum, where $m = \lceil n/\log n \rceil$.

Next, we analyze the time required by the procedure *H-SCAN*. Since lines 2c, 2d, 2e, 2g and 2h take t_e time each, and line 2f takes p-t_r time, the total time taken by the procedure is $5\lceil \log n \rceil .t_e + 2(2^0 + 2^1 + 2^2 + \ldots + 2^{(\lceil \log n \rceil - 1)}).t_r$, or, $5\lceil \log n \rceil .t_e + 2^{\lceil \log n \rceil}.tr$.

We are now ready to analyze the time complexity of algorithm *H-SORT*. Line 7 takes t_s time and lines 8 and 9 take t_e time each. The loop 5–10, which is executed $\lceil \log n \rceil$ times, requires $2\lceil \log n \rceil .t_e + \lceil \log n \rceil .t_s$ time. Lines 2, 3, 13, 14 and 16 takes t_s time each, and lines 1, 4 and 12 take t_e time each. Considering the times required by the procedures *FIND-COUNT* and *SCAN*, the total time required by algorithm H-SORT is $(5 + \lceil \log n \rceil .t_s + (3 + 8\lceil \log n \rceil - 3\lceil \log\log n \rceil).t_e + (\lceil n/\log n \rceil + 2^{\lceil \log n \rceil} + \lceil \log n \rceil - \lceil \log\log n \rceil).t_r$. Consequently, the execution time of the algorithm, as given by the coefficients of t_e, is $O(\log n)$, and the interprocessor communication time, as given by the coefficient of t_r, is $O(n)$.

It is to be noted that the this timing analysis is done without assuming any particular ordering between comparison-exchange time (so called execution time of the algorithm) and the data communication time between processors. Since the former has an $O(\log n)$ complexity and the latter has an $O(n)$ complexity, in any VLSI implementation the communication time will eventually dominate over the comparison-exchange time as the number of elements, n, grows. Hence from user point of view for large n the limiting factor is the propagation delay down the wires and not the latency of the circuit elements.

4. CONCLUSION

We have described an $O(\log n)$ parallel algorithm to sort a given set of n elements. We have also shown how our algorithm can be easily mapped on an SIMD mesh connected array of $n^* \lceil n/\log n \rceil$ homogenous processors. The execution time complexity of the equivalent hardware algorithm is $O(\log n)$ and the interprocessor communication time complexity is $O(n)$. The execution time complexity of $O(\log n)$ of our algorithm is better than the time complexities of parallel sorting algorithms proposed by Valiant,[2] Hirschberg,[11] Stone[5] and Bentley and Kung.[6] In terms of execution time complexity, our algorithm compares favorably even with sorting algorithms which have used the SIMD-mesh-connected model of parallel computation, like Refs. 9 and 10, though they have a better communication time complexity. Also, when compared to algorithms which achieve the same $O(\log n)$ comparison time complexity, our algorithm has the advantage of having a better AT^2 (Ref. 1) value – $O(n^2 \log n)$ as compared to $O(n^2 \log^2 n)$ of Ref. 12 and our algorithm does not require the keys to be restricted to a size of $O(\log n)$ as in Ref. 12. Moreover, the simplicity of the underlying network topology required by our algorithm makes mapping of the corresponding hardware algorithm to existing array processors, like Illiac-IV, and reconfigurable architectures, very easy.

We also want to make two observations before we conclude. By VLSI implementation of an algorithm we have meant in this paper implementation of the algorithm on an architecture that consists of regular interconnection of identical processing elements (PE's). The processing elements are simple devices capable of doing a few simple operations and the emphasis is on the fact that all processing elements are identical. And since the interconnection pattern among these processors is regular, it will be possible to design chips or wafers for the entire architecture at least for moderate values of n with present day state of the art of technology. On the other hand it is also possible to view the architecture as interconnection of processors at the board level constructed from chip components. Secondly we have used the term 'SIMD machine' in the normally used sense of the term, i.e., a single instruction is broadcast to all the PE's simultaneously and the design is orthogonally connected bidirectional with point-to-point connections. This implies a central program store and removes the need for a stored program at each PE (Fig. 3 does not show the central program store and the associated connections). The central program broadcasts the same instruction to all the PE's but depending

on different local data in the PE's the results are different; this is particularly true for the summation and scan operations. The instruction set has been so designed to depend on the actual values of the operands at execution time and to do different things on different data. It is also to be noted that the same central program may be made responsible for initial loading of data to all the processors and hence there is no need of global memory connections for the load-store instructions. As usual, we have not considered the time needed for loading and storing in our analysis of the time complexity of the algorithm (since they are not part of the algorithm). Also if that is done by the central program, that will mean only a constant time difference.

Our approach in developing the parallel algorithm illustrates the difference between philosophies of designing efficient sequential and parallel algorithms. It may be noted here that sequential version of the proposed algorithm will be very time-inefficient due to the presence of many redundant operations. However, redundant operations are sometimes welcome in designing parallel algorithms, especially when the aim is to reduce execution time at the cost of more processing elements.

ACKNOWLEDGEMENT

The authors are grateful to the anonymous referees for their detailed comments which greatly helped to improve the clarity of presentations.

REFERENCES

1. C. D. Thompson, A complexity theory for VLSI, *Ph.D. Dissertation, Dept. of Comp. Sc., CMU* (1980).
2. L. Valiant, Parallelism in comparison problems, *SIAM J. Comput.*, **4** (3), 348–355 (1975).
3. G. H. Barnes, The Illiac-IV computer, *IEEE Trans. on Comp.*, **C-17**, 746–757 (1977).
4. N. J. Flynn, Very high speed computing systems, *Proc. IEEE*, **54** (12), 1901–1909 (1966).
5. H. S. Stone, Parallel processing with the perfect shuffle, *IEEE Trans. on Comp.*, **C-20** (2), 153–161 (1971).
6. J. L. Bentley and H. T. Kung, A tree machine for searching problems, *Proc. of the 1979 Int. Conf. on Parallel Processing, New York*, pp. 257–268.
7. D. Bitton *et al.*, A taxonomy on parallel sorting, *ACM Comp. Surveys*, **16** (3), 287–318 (1984).
8. K. E. Batcher, Sorting networks and their applications, *Proc. of the 1968 Spring Joint Comput. Conf., Atlantic City, N.J.*, **32**, AFIPS Press, Reston, Va., pp. 307–314.
9. C. D. Thompson and H. T. Kung, Sorting on a mesh connected parallel computer, *Comm. ACM*, **20** (4), 263–271 (1979).
10. M. Kumar and D. S. Hirschberg, An efficient implementation of Batcher's odd-even merge algorithm and its application in parallel sorting schemes, *IEEE Trans. on Comp.*, **C-32** (3), 254–264 (1983).
11. D. S. Hirschberg, Fast parallel sorting algorithms, *Comm. ACM*, **21** (8), 657–666 (1978).
12. G. Bilardi and F. P. Preparata, A minimum area VLSI network for O(log n) time sorting, *IEEE Trans. on Comp.*, **C-34** (4), 336–343 (1985).
13. K. E. Batcher, Design of a massively parallel processor, *IEEE Trans. Comput.*, **C-29**, 836–840 (1980).
14. F. P. Preparata, New parallel sorting schemes, *IEEE Trans. Comput.*, **C-27**, 669–673 (1978).

Announcements

22—25 OCTOBER 1990

JERUSALEM, ISRAEL

The 5th Jerusalem Conference on Information Technology (JCIT)
The Technologies of the 90's

The 5th Jerusalem Conference on Information Technology (JCIT) will take place on 22–25 October 1990. Like its four predecessors in 1971, 1974, 1978 and 1984, the conference will cover a broad range of topics on computer technology and applications, and will also explore the economics and management of the information industry. The emphasis will be on the technologies of the 90's.

JCIT is an international conference whose goals are to provide a broad based forum for the presentation of achievements and innovative ideas in the various areas of information technology. More than 40 countries were represented at the previous JCITs and an even larger participation is expected at JCIT-5. An attendance of over 3,000 from Israel and 600 from abroad, is anticipated.

The conference will provide an interdisciplinary environment for computer scientists, engineers, users and managers to exchange views and ideas, and discuss their likely impact on the information systems of the next decade.

The technical program will consist of papers by prominent invited speakers, submitted papers, panel discussions and exhibitions covering the state of the art.

Opportunities for the participants to become acquainted with the achievements of Israel as a fast developing country in the computer and related fields will be made available by means of an extensive international exhibit of computer hardware and software products displaying new trends in information technology. To this end, JCIT will be held in conjunction with the 25th National Conference of the Information Processing Association of Israel, and the 21st National Conference of the Israeli System Analyst Association.

Conference Topics

Technology: Software and Hardware
Foundation of Computer Science
Computer Architecture
Hardware Design
Distributed Systems
Networks and Communications
Data Bases
Software Engineering
Operating Systems
Logic Programming
Logic of Programs
Reliability and Performance

Applications
Artificial Intelligence
Natural Language Processing
Computer Assisted Instruction
Graphics
Vision
Computers in Medicine
Computer Aided Design and Manufacturing
Office Automation

1. *Conference chairpersons*:
Y. Maor, IBM, Tel-Aviv, Israel.
A. Peled, IBM T.J. Watson Research Center, Yorktown Hts., N.Y. USA.

2. *Organizing committee chairpersons*:
D.Z. Mittwoch, NCR, Tel-Aviv, Israel.
M. Gottlib, Bar-Ilan University, Ramat Gan, Israel.

3. *Program committee chairpersons*:
I. Borovits, Tel-Aviv University, Tel-Aviv, Israel.
Z. Manna, Stanford University, Palo Alto, Calif. USA.
A. Pnueli, Weizmann Institute of Science, Rehovot, Israel.

Secretariat:
Information Processing Association of Israel, Kfar Hamacabia Ramat-Gan 52109, Israel.