MIT/LCS/TM-272

TIGHT BOUNDS ON THE COMPLEXITY OF PARALLEL SORTING

Tom Leighton

April 1985

Tight Bounds on the Complexity of Parallel Sorting

Tom Leighton

Mathematics Department and
Laboratory for Computer Science
Massachusetts Institute of Technology
Cambridge, Massachusetts 02139

Abstract: In this paper, we prove tight upper and lower bounds on the number of processors, information transfer, wire area and time needed to sort N numbers in a bounded-degree fixed-connection network. Our most important new results are:

1) the construction of an N-node degree-3 network capable of sorting N numbers in $O(\log N)$ word steps,

2) a proof that any network capable of sorting N $(7 \log N)$-bit numbers in T bit-steps requires area A where $AT^2 \geq \Omega(N^2 \log^2 N)$, and

3) the construction of a "small-constant-factor" bounded-degree network that sorts N $\Theta(\log N)$-bit numbers in $T = \Theta(\log N)$ bit steps with $A = \Theta(N^2)$ area.

Key Words: area-time tradeoffs, circuit complexity, communication complexity, fixed-connection network, information transfer, packet routing, parallel computation, sorting, universal computer, very large scale integration.

This research was supported by Air Force contract OSR-82-0326 and DARPA contract N00014-80-C-0622. A preliminary version of this paper was presented at the 16th Annual ACM Symposium on Theory of Computing [17].

1. Introduction

The problem of sorting N numbers with a fixed-connection network has a long and rich history [1-30]. For the most part, the complexity of parallel sorting has been measured in terms of the *number of processors* used and the *number of parallel operations* performed. In conjunction with the development of distributed computing and very large scale integration (VLSI), the complexity of parallel sorting has also been measured in terms of required *information transfer* and *chip area*. Determining the complexity of sorting in these four measures has remained a difficult problem for some time. Recently, however, several significant advances have been made. In some cases (particularly the breakthrough work of Ajtai, Komlos and Szemeredi [2]), tight bounds have been proved. In other cases (most notably [24] and [28]), methods have been developed that almost lead to tight bounds and that substantially increase our knowledge of the problem.

In this paper, we combine the work of [2] and [28] with new methods to precisely determine the complexity of sorting with fixed-connection networks. Our results and their relevance to previous work are described in the remainder of the introduction. The proofs are contained in Sections 2 through 5. In particular: Section 2 contains a description of a simple sorting algorithm that we call *columnsort*, Section 3 contains proofs of the bounds for the number of nodes needed to sort, Section 4 contains proofs of the bounds for information transfer and area, and Section 5 describes area-optimal, small-constant-factor networks. We conclude with some remarks and directions for future research in Section 6.

1.1 Two Fundamental Sorting Problems

Much of the work on parallel sorting has been directed towards solving the following two problems.

Problem 1: *Construct an $O(\log N)$-level circuit that sorts N numbers.*

An example of a 3-level circuit that sorts 4 numbers is shown in Figure 1. In general, each level consists of $N/2$ disjoint *comparators*. Each comparator can be viewed as an edge that possibly exchanges the numbers at its endpoints so that the bigger number exits at the endpoint marked B and so that the smaller number emerges at the other endpoint (marked S). After passing through all the levels (from left to right), the numbers emerge from the circuit in sorted order. As the comparisons and exchanges in each level are performed simultaneously, the total time required to sort the N numbers is equal to the number of levels in the circuit. Figure 2 illustrates the sorting process for the list 4, 7, 2, 9. In this example, 2 and 4 are exchanged in the first level, and 4 and 7 are exchanged in the third level.

Problem 2: *Construct a bounded-degree, $O(N)$-node network that sorts N numbers in $O(\log N)$ steps.*

An example of a 4-node, degree-3 network that sorts 4 numbers in 3 steps is shown in Figure 3. As in Problem 1, the edges serve to transmit numbers between processors. In Problem 2, however, the nodes are active throughout the algorithm and must be equipped with a local control telling them what to do at each step. In general, the control might be quite complex (depending on the

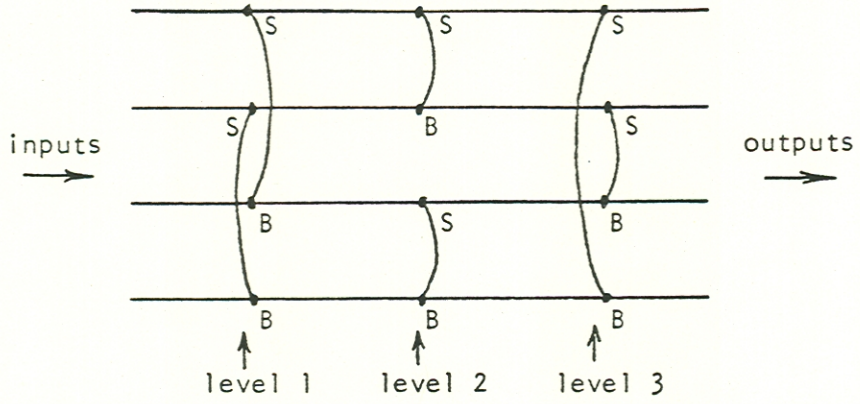

Figure 1: *A 3-level circuit for sorting 4 numbers.*

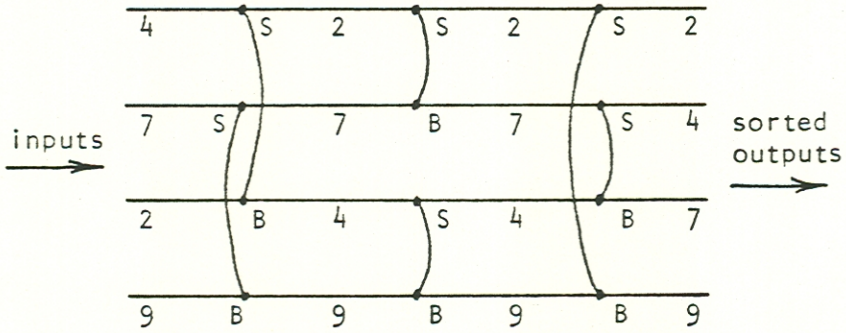

Figure 2: *Sorting the list 4, 7, 2, 9 with the 3-level circuit shown in Figure 1.*

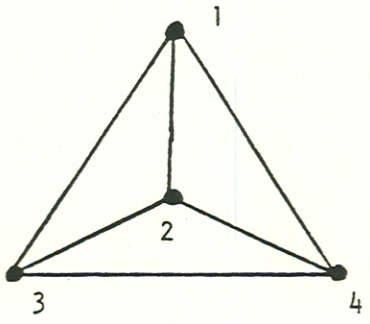

Figure 3: *A 4-node, degree-3 network that sorts 4 numbers in 3 steps. Initially each node contains one number. After 3 steps, the ith smallest number is in the ith node.*

time, the position of the node in the network, the value of N, as well as the history of everything the node has seen and done). For our purposes, however, it will be sufficient to consider local controls that can be described with a constant number of bits. In the example shown in Figure 3, each node corresponds to a row of Figure 1 and each edge corresponds to a comparator in Figure 1. In this case, each local control tells its node which edge should be used as a comparator at each step. Level 1 edges are used at the first step, level 2 edges at the second step, and level 3 edges at the third step. After 3 steps, the numbers are sorted.

Early work on Problems 1 and 2 led to the construction of a $\Theta(\log^2 N)$-level sorting circuit [4] and to the construction of an N-node, degree-3 network that sorts N numbers in $\Theta(\log^2 N)$ steps [26]. Both constructions are based on the butterfly implementation of odd-even merge sort (e.g., see [18, 29]). More recently, Ajtai, Komlos and Szemeredi [2] solved Problem 1 by constructing an $O(\log N)$-level sorting circuit. (Henceforth, we will refer to this circuit as the *AKS* sorting circuit.) This result also provided partial solutions to Problem 2; namely an N-node, $\Theta(\log N)$-degree network that sorts in $O(\log N)$ steps, and a $\Theta(N \log N)$-node bounded-degree network that sorts in $O(\log N)$ steps. In both cases, the resulting network for Problem 2 has $\Theta(N \log N)$ edges.

The only other improvement of the initial $\Theta(\log^2 N)$-step bound for Problem 2 is due to Reif and Valiant [24], who constructed an N-node, bounded-degree network that can sort in $O(\log N)$ steps with high probability provided that each node is allowed to maintain an $O(\log N)$-number queue. If (as is common) unbounded-size queues are not allowed, then the number of nodes in the Reif-Valiant construction may have to be increased by a factor of $\Theta(\log N)$ to simulate the queues. Hence the construction may really require $\Theta(N \log N)$ nodes. In addition, the time requirement might have to be increased by a factor of $\Theta(\log \log N)$ to manage the queues. (Whether or not the factor of $\Theta(\log N)$ blowup in the number of nodes and of $\Theta(\log \log N)$ in time is really needed to simulate the queues is not known. Very recent work by Pippenger, however, suggests that the blowups may not be needed, at least for a modified version of the algorithm [23].) Moreover, it *is* known that the randomness assumption cannot be removed from the Reif-Valiant algorithm since Borodin and Hopcroft [9] showed that any such algorithm requires $\Omega(\sqrt{N})$ steps in the worst case.

In Theorem 1 of this paper, we show that any solution for Problem 1 can be simply transformed into a solution for Problem 2, thereby extending the work of [2] to solve Problem 2. In fact, we prove that any depth-T sorting circuit can be transformed into an N-node, degree-3 network that sorts N numbers in $O(T)$ steps. The proof of this simple yet unexpected fact combines a standard pipelining argument with a generalization of odd-even merge sort that we call *columnsort*. The details are provided in Sections 2 and 3.

1.2 The Bit Model of Computation

Problem 2 can be reformulated in a variety of ways. One natural formulation restricts each node to have finite memory and control. Such is the case for the *bit model* of parallel computation, where each node has just c bits of state for some constant c that is independent of N. In the bit model, each bit of each input number and of each sorted number is treated individually. During a single *bit step*, each node can perform a constant number of bit-size operations (such as a compare). As a consequence of these restrictions, it is not possible to store a $\Theta(\log N)$-bit

number in a single node, nor is it possible to compare two $\Theta(\log N)$-bit numbers in a single bit step. Hence the bit model is far more restrictive than the corresponding *word model*, for which Problems 1 and 2 were originally defined. (Two numbers *can* be compared in a single *word step* in the word model, for example.)

It is well known [12], that any $O(\log N)$-level sorting circuit in the word model can be transformed into an $O(N \log N)$-node, bounded-degree network that sorts N $\Theta(\log N)$-bit numbers in $O(\log N)$ bit steps in the bit model. In fact, the two networks are the same. Instead of passing whole numbers at a time from left to right, the bit model version of the network passes the numbers bit-by-bit from left to right, most significant bits first. Each comparator sees two numbers bit-by-bit, leading bits first. As long as the leading bits of the two numbers are identical, the comparator simply passes the bits through. (Of course, the output would be the same if the comparator were exchanging the bits as long as the leading bits are identical.) Once bits are found that are different, the comparator knows instantly which number is bigger and henceforth sends all remaining bits of the bigger number to the node marked B and all remaining bits of the smaller number to the node marked S. The total time taken is the sum of the number of levels in the circuit and the length of the bit strings for each number. For $O(\log N)$-bit numbers and an $O(\log N)$-level circuit, at most $O(\log N)$ bit steps are used.

It is natural to ask whether or not $\Omega(N \log N)$ nodes are really needed to sort N $\Theta(\log N)$-bit numbers. For example, it might be possible to sort with $O(N)$ nodes, particularly if more time is allowed. In Theorem 2, we show that this is not the case. In fact, we show that no matter how much time is allowed, $\Omega(N \log N)$ nodes are required to sort N $(2 \log N)$-bit numbers. The result can be extended to sorting k-bit numbers where $k \geq (1 + \epsilon) \log N$ for any $\epsilon > 0$, but cannot be extended for values of $k \leq \log N$ since N $(\log N)$-bit numbers can be sorted (given enough time) using $O(N)$ nodes [25]. The proof of the lower bound requires that each input is provided just once and that the input/output schedule is where and when oblivious. The arguments used are similar to those used by Ullman [29] to prove an $\Omega(N)$ lower bound on the number of nodes needed to sort N $(\log N + 1)$-bit numbers.

In case it wasn't already obvious, the proof of the $\Omega(N \log N)$ lower bound on the number of processors necessary to sort N numbers in the bit model also provides an $\Omega(N)$ lower bound on the number of processors necessary to sort in the word model. Taken together, the results of Section 3 provide tight bounds on the number of nodes needed to sort in both the bit and word models. In addition, both lower bounds are achieved with $O(\log N)$-time algorithms (the best possible).

1.3 Information Transfer and Area Bounds

With the development of VLSI technology, *wire area* has become an important measure of a problem's difficulty. Whereas there is no relationship between the number of nodes and the number of steps necessary to sort N numbers, there is a relationship between the wire area and the number of steps. In the word model, it is well known that $AT^2 \geq \Omega(N^2)$ where A is the minimum area of any network that sorts in T steps [27, 28]. For $T = O(\log N)$, this means that $A \geq \Omega(N^2/\log^2 N)$. In Theorem 3, we show that the network constructed in Section 3 achieves this bound. Moreover, the construction can be modified (without increasing the number

of nodes, $O(N)$) to achieve the AT^2 lower bound for any T in the range $\Omega(\log N) \leq T \leq O(\sqrt{N})$. Formerly, such results were known only for $T \geq \Omega(\log^3 N)$ [6].

Of more practical interest is the AT^2 tradeoff for sorting networks in the bit model. Using crossing sequence techniques, several researchers have shown that the information transfer necessary to sort N $(\log N+1)$-bit numbers is at least $\Omega(N)$. The *information transfer* or, equivalently, *communication complexity* of a problem is the minimum number of bits that must pass between the left and right halves of the chip during a worst-case computation (provided that each "half" of the chip outputs half of the bits of the sorted numbers). Since the square of the information transfer is a lower bound for AT^2 [27], we know that $AT^2 \geq \Omega(N^2)$. Angluin and Thompson [3] (and later El Gamal [10]) improved this bound to $AT^2 \geq \Omega(N^2 \log N)$ and Thompson conjectured that the true bound is $AT^2 \geq \Omega(N^2 \log^2 N)$. Although the intuition for the stronger bound is clear, it can be misleading. For example, the same intuition also leads to the conjecture that $AT^2 \geq \Omega(N^2 k^2)$ for sorting N k-bit numbers where $k >> \Omega(\log N)$. The latter conjecture is false, however. In fact, the results in this paper can be used to construct networks for sorting N k-bit numbers that achieve an AT^2 bound of $O(N^2 k \log N \log^2(\frac{k}{\log N}))$ which is significantly less than $\Omega(N^2 k^2)$ for large k [18].

In Theorem 4, we verify that $AT^2 \geq \Omega(N^2 \log^2 N)$ by showing that the information transfer necessary to sort N $(7 \log N)$-bit numbers is $\Omega(N \log N)$. As before, this bound can be achieved for all T in the range $\Omega(\log N) \leq T \leq O(\sqrt{N \log N})$. For each T, the network contains just $O(N \log N)$ nodes, the fewest possible. Previous constructions for achieving these bounds were limited to the case when $T \geq \Omega(\log^3 N)$ [6].

1.4 Applications

With the advent of VLSI technology, it has become possible to fabricate large numbers of simple processors and to integrate them into large-scale, highly-parallel computers. In some circles, such machines are called *supercomputers*. Examples include the M.I.T. Connection Machine and the N.Y.U. Ultracomputer. For the most part, the architectures of these machines are based on well-known fixed-connection networks such as the hypercube, shuffle-exchange graph, cube-connected cycles and the FFT butterfly. The reason that these networks are used is that they can support fast algorithms for interprocessor communication and routing of data. In fact, most of these machines will spend most of their time trying to get the right data to the right processor at the right time. Hence a good solution to the data routing problem is critical to the successful construction of supercomputers.

In a fundamental paper [30], Valiant and Brebner formalized the importance of data routing by showing that if an N-processor fixed-connection network could solve an M-packet routing problem in $T(M)$ steps, then it could simulate any M-processor parallel machine with only a $T(M)$-factor time delay. If $M = N$ and $T(N) = O(\log N)$, such a network could be reasonably called *universal* since it could simulate any other parallel machine with the same number of processors (regardless of the interconnection architecture) with only an $O(\log N)$-factor time delay, the least possible.

Although good algorithms are known for fixed-permutation [21] and random-permutation [23, 30] data routing problems, the general many-one data routing problem is still best solved as a

special case of sorting [4, 18, 29], which is why parallel sorting is so important. In fact, were a good solution to Problem 2 found for $N = 10^6$ (the number of processors in currently planned machines), it would probably provide an excellent basis for the architecture of a supercomputer. Unfortunately, all of the constructions described thus far in the paper utilize the AKS sorting circuit which behaves terribly for "small" values of N (e.g., $N < 10^{100}$). Although variations of the AKS construction have been proposed [22], they are still a long way from being considered practical.

In Section 5 of this paper, we use columnsort to construct $O(\log N)$-time "small-constant-factor" sorting networks that do not depend on the AKS construction. These networks are area-optimal but they require more than the optimal number of nodes. Unfortunately, researchers who are building supercomputers appear to be constrained by processing time and the number of nodes as well as by wire area. Hence the networks in Section 5 are probably still not practical.

Recently, however, we have discovered probabilistic versions of columnsort for which it appears possible to sort $(1 - \epsilon)N$ numbers in nearly $2 \log N \log\log N$ word steps on an N-node shuffle-exchange graph. Although we are still working out the details, it seems quite possible that the algorithm will improve the traditional $\log^2 N$ time bound by an order of magnitude for $N = 10^6$. The details of this work will be reported when completed [19]. Similar observations have been made by Ajtai [1].

1.5 Summary

The results described in this paper complete the asymptotic characterization of the number of nodes, number of steps, amount of information transfer and amount of area needed to sort N $\Theta(\log N)$-bit numbers in a bounded-degree, fixed-connection network in both the bit and word models. Up to the AT^2 constraint, all of the lower bounds can be achieved simultaneously by a single construction for each model. In addition, the constructions are of the small-constant-factor variety when optimality in the number of nodes is sacrificed. For easy reference, we have summarized the bounds in the Table 1. (It should be noted that the lower bound on the number of

Table 1
Bounds on the Complexity of Parallel Sorting

	Word Model	Bit Model
Number of Nodes	$\Theta(N)$	$\Theta(N \log N)$
Information Transfer	$\Theta(N)$	$\Theta(N \log N)$
AT^2 (for $\Omega(\log N) \leq T \leq O(\sqrt{N})$)	$\Theta(N^2)$	$\Theta(N^2 \log^2 N)$

nodes needed in the bit model holds only when the number of bits per number exceeds $(1+\epsilon)\log N$ for some constant $\epsilon > 0$, and that the lower bound on the amount of information transfer needed in the bit model is only known to hold when the number of bits per number exceeds $7 \log N$.)

1.6 Acknowledgements

I would like to thank Alok Aggarwal, Miklos Ajtai, Noga Alon, Gianfranco Bilardi, Abbas El Gamal, Jon Greene, Tom Knight, Clyde Kruskal, Charles Leiserson, Gary Miller, Franco Preparata, John Reif, Alan Siegel, Clark Thompson and my *Theory of Parallel Computation and VLSI* students for their helpful discussions and comments.

I am particularly indebted to Gianfranco Bilardi and Franco Preparata for their contributions to this paper. Originally, I had considered the area upper bounds in Theorems 3, 5 and 6 only for the special case when $T = O(\log N)$. Independently, Bilardi and Preparata proved (using a very different construction) the same bounds when $T = O(\log N)$ in Theorems 5 and 6, as well as the general spectrum of bounds for arbitrary T in Theorem 6. Using the recent Bilardi-Preparata result that bounds the area of the AKS sorting circuit, I was later able to simplify the proofs of Theorems 3 and 5. All three area upper bounds were extended to their current general form (i.e., for all T) during discussions with Bilardi.

2. Columnsort

In this section, we describe a simple sorting algorithm that we call *columnsort*. The algorithm is a generalization of odd-even merge, but for simplicity, we first describe the algorithm as a series of elementary matrix operations. The relationship with odd-even merge will be made clear later.

Let Q be an $r \times s$ matrix of numbers where $rs = N$, $s \mid r$ and $r \geq 2(s-1)^2$. Initially, each entry of the matrix is one of the N numbers to be sorted. After completion of the algorithm, the i,j entry $(0 \leq i < r, 0 \leq j < s)$ of Q will contain the pth sorted number $(0 \leq p < N)$ where $p = i + jr$. For example, Figure 4 illustrates a typical matrix before and after sorting. (For simplicity, we have chosen a 6×3 matrix to illustrate the algorithm even though it does not satisfy the constraint that $r \geq 2(s-1)^2$. We will discuss the relevance of this constraint and the degree to which it can be relaxed later.)

Columnsort has eight steps. In Steps 1, 3, 5 and 7, the numbers within each column are sorted. (Just how we accomplish this will depend on the application and does not matter for the analysis in this section.) In Steps 2, 4, 6 and 8, the entries of the matrix are permuted. The permutation in Step 2 (shown for a 6×3 matrix in Figure 5) corresponds to a "transpose" of the matrix. The entries are picked up column by column and then deposited row by row (always going from top to bottom in a column and from left to right in a row). The permutation in Step 4 is the inverse of that in Step 2. The permutation in Step 6 corresponds to an $\lfloor r/2 \rfloor$-shift of the entries and is shown for a 6×3 matrix in Figure 6. The permutation in Step 8 is the reverse of that in Step 6. The step-by-step application of columnsort to the matrix in Figure 4 is shown in Figure 7.

$$\begin{bmatrix} 6 & 15 & 12 \\ 14 & 4 & 7 \\ 10 & 1 & 13 \\ 3 & 16 & 9 \\ 17 & 8 & 2 \\ 5 & 11 & 0 \end{bmatrix} \qquad \begin{bmatrix} 0 & 6 & 12 \\ 1 & 7 & 13 \\ 2 & 8 & 14 \\ 3 & 9 & 15 \\ 4 & 10 & 16 \\ 5 & 11 & 17 \end{bmatrix}$$

(a) (b)

Figure 4: *A 6×3 matrix before (a) and after (b) sorting.*

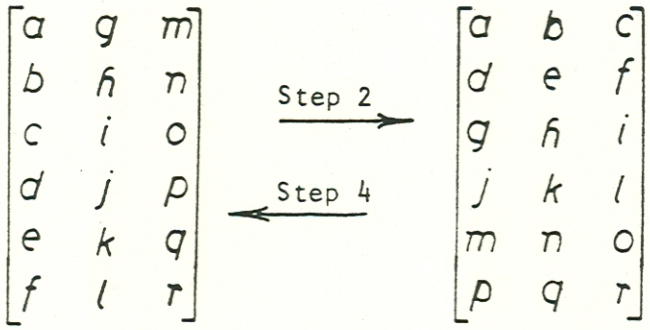

Figure 5: *The transpose and untranspose permutations in Steps 2 and 4, respectively.*

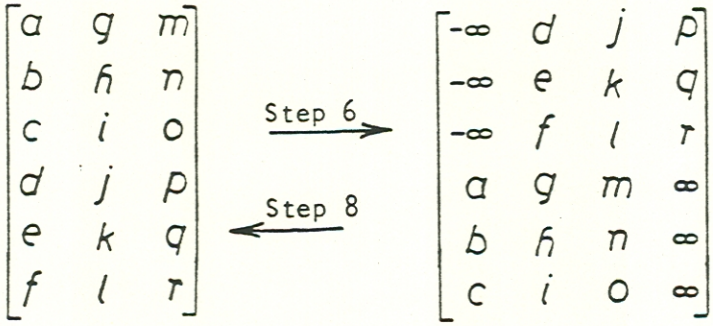

Figure 6: *The shift and unshift permutations in Steps 6 and 8, respectively. The $-\infty$'s denote arbitrarily small dummy elements, and the ∞'s denote arbitrarily large dummy elements.*

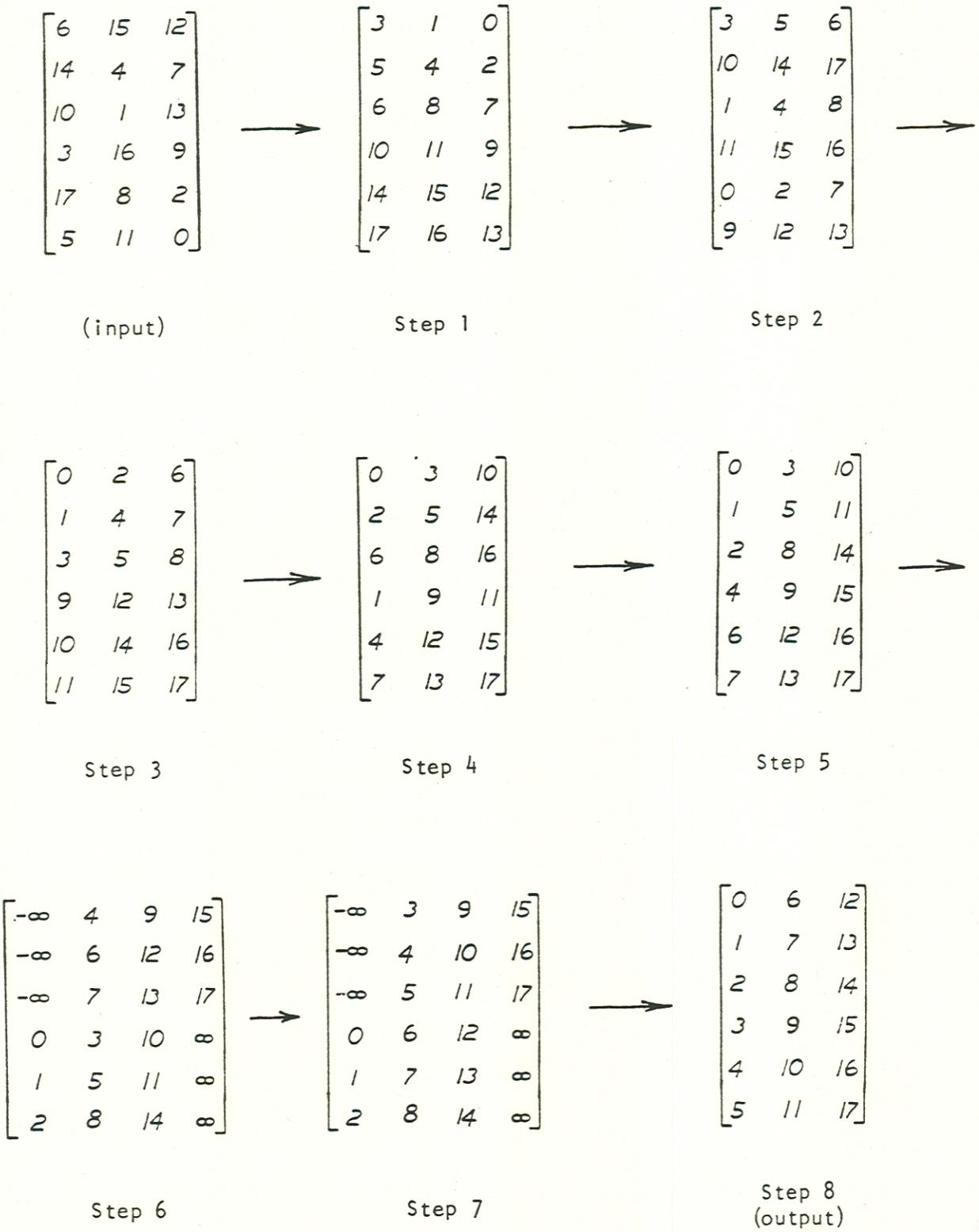

Figure 7: *The step by step application of columnsort to the matrix in Figure 4.*

At first glance, it seems impossible that such an algorithm works. For example, if the matrix were square (i.e., if $r = s$, which is not allowed), then we would essentially just be sorting rows and columns which is well-known to leave entries arbitrarily far away from their correct sorted position.

A far better intuition comes from the case when $r = N/2$ and $s = 2$. In this special case, we have precisely odd-even merge. In odd-even merge, a list of N numbers is divided into 2 sublists, each with $N/2$ numbers. This division corresponds to entering the numbers in an $N/2 \times 2$ matrix: each column of the matrix contains a sublist. The two sublists are then sorted, as is done in Step 1 of columnsort. Then the odd-index numbers in each sublist are combined to form a new sublist, as are the even-index numbers. This corresponds to the transpose (or "unshuffle") operation in Step 2 of columnsort. Next, each sublist is sorted, as is done in Step 3 of columnsort. (In odd-even merge, this sorting step is accomplished with a recursive merge.) After sorting, the sublists are shuffled together, as is done in Step 4 of columnsort. At this point, every number is within one of its correct position, so each number is compared to its neighbors and (possibly) interchanged, thus completing the sorting. The same maneuver is accomplished in a rather brute-force way by columnsort. In Step 5, all but the top and bottom numbers in each column are compared to their neighbors by sorting the columns. Steps 6-8 insure that comparisons are made between numbers at the bottom of one column and the top of the next column.

The action of columnsort for arbitrary $r \geq 2(s-1)^2$ is very much like that for odd-even merge. After Step 4, we will be guaranteed that every number is within $(s-1)^2$ of its correct sorted position. Then Steps 5-8 are sufficient to finish the sorting. We prove these two facts in what follows.

Consider a number x that is in position i, j of the matrix after Step 3. A simple calculation shows that x is sent to a position in Step 4 that corresponds to a rank of $p = is + j$ in the sorted list. (Recall our convention that the smallest number has rank zero.) From the position of x after Step 3, we know that x is greater than or equal to at least $i + 1$ numbers in the jth column of the matrix after Step 2. Let a_k denote the number of these $i + 1$ numbers that originally come from column k of the matrix (i.e., before Step 2 transposed the matrix). By definition,

$$i + 1 = \sum_{k=0}^{s-1} a_k .$$

Since only the jth and every sth number thereafter of the (sorted) kth column after Step 1 appear in the jth column after Step 2, this means that x is greater than or equal to at least $(a_k - 1)s + j + 1$ numbers in the kth column of the matrix after Step 1. Hence the true rank of x is at least

$$\sum_{k=0}^{s-1} [(a_k - 1)s + j + 1] - 1 .$$

Substituting $i + 1$ for $\sum_{k=0}^{s-1} a_k$ and simplifying, we find that the true rank of x is at least

$$si + sj - (s-1)^2 .$$

Hence the position of x after Step 4 is at most

$$(s-1)^2 - j(s-1) \leq (s-1)^2$$

beyond its correct position.

A symmetric argument shows that the true rank of x is at most $si + sj$. Hence the position of x after Step 4 is at most $j(s-1) \leq (s-1)^2$ short of its correct sorted position. Thus we have established that every number is within $(s-1)^2$ of its correct position after Step 4. When $s = 2$, we have the special case result for odd-even merge. When $s = r = \sqrt{N}$, we see that virtually nothing has been gained, which is why the algorithm doesn't work for square matrices.

We now show that Steps 5-8 are sufficient to complete the sorting. For simplicity, we assume in what follows only that every number is within $\lfloor r/2 \rfloor$ of its correct sorted position. Since $r \geq 2(s-1)^2$, we are always guaranteed that this condition is met after completion of Step 4.

After Step 4, every number that belongs in the top half of column j ($0 \leq j < s$) when sorted is in column j or in the bottom half of column $j-1$. Similarly, every number that belongs in the bottom half of column j is in column j or the top half of column $j+1$. Otherwise, some number would be more than $\lfloor r/2 \rfloor$ away from its correct position. After Step 5, every number that belongs in the top half of column j is in the top half of column j or the bottom half of column $j-1$. Were this not true and were such a number x to be in the bottom half of column j, then x, every number ahead of x in column j and, of course, every number in columns $0, 1, ..., j-1$ would have to have rank less than $rj + r/2$, which is impossible. Alternatively, were x in the top of column $j-1$ at this point, then there could be at most $r(j-1) + \frac{r}{2} - 1 + \frac{r}{2} = rj - 1$ numbers of rank less than or equal to $rj - 1$, which is also impossible. (Recall, that there are rj such numbers since the smallest number has rank zero.) This total is calculated by counting the $r(j-1)$ numbers in columns $0, 1, ..., j-2$, the $\frac{r}{2} - 1$ or fewer numbers ahead of x in column $j-1$ and the $\frac{r}{2}$ or fewer numbers in column j that could belong in column $j-1$. Using identical arguments, we can also show that after Step 5, every number that belongs in the bottom half of column j is in the bottom half of column j or the top half of column $j+1$.

Combining the two facts in the preceeding paragraph, we find that every number that should be in the bottom half of column j or the top half of column $j+1$ when sorted is in one of these two half-columns after Step 5. Hence, Steps 6-8 complete the sorting. This completes the proof that columnsort works.

Columnsort provides an efficient way to sort N numbers given that we know how to sort r numbers where $rs = N$, $s \mid r$ and $r \geq 2(s-1)^2$. For example, 24 numbers can be sorted by repeatedly sorting subsets of 8 numbers. It would be interesting to know how much the constraint on the size of r can be relaxed without radically changing the algorithm. Some improvement is definitely possible. For example, if Step 4 were replaced with the diagonalizing permutation shown in Figure 8, it would be necessary only that $r \geq s(s-1)$. This is because a number in the i, j position after Step 3 would then correspond to a rank of

$$si + sj - s(s-1)/2 + s - 1 - j$$

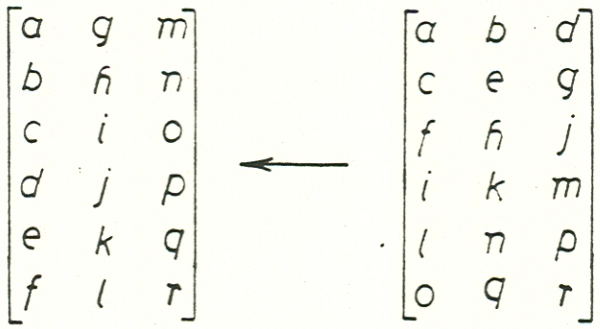

Figure 8: *A more effective, diagonalizing permutation for Step 4 of columnsort.*

for all but the extreme cases when $i+j > r$ and $i+j < s-1$. This would mean that every number would be at most $s(s-1)/2$ from its correct position after Step 4, and thus that $r \geq s(s-1)$ would be sufficient. In addition, the constraints that $rs = N$ and $s \mid r$ can be removed provided that r is one larger.

3. Bounds on the Number of Nodes

In this section, we prove that N nodes are sufficient to sort N numbers in the word model and that $\Theta(N \log N)$ nodes are necessary to sort N k-bit numbers in the bit model whenever $k \geq (1+\epsilon)\log N$ for some constant $\epsilon > 0$.

3.1 Results for the Word Model

In the following theorem, we show how to use columnsort to convert a family of $f(N)$-level circuits for sorting N numbers into a family of bounded-degree, $O(N)$-node circuits that can sort N numbers in $O(f(N))$ word steps. In the theorem, we choose $f(N) = o(N^{1/3})$ so that $r \geq 2(s-1)^2$ when columnsort is applied, where $r = \frac{N}{f(N)}$ and $s = f(N)$. As a consequence, we can transform the AKS circuit for Problem 1 into a solution for Problem 2.

Theorem 1: *Given a monotone function f such that $f(N) = o(N^{1/3})$ for all N and a family of $f(N)$-level circuits for sorting N numbers, one can construct a family of bounded-degree, $O(N)$-node networks that can sort N numbers in $O(f(N))$ word steps.*

Proof: Select a circuit from the family that sorts $\frac{N}{f(N)}$ numbers. Since f is monotone, this circuit has depth $f(\frac{N}{f(N)}) \leq f(N)$ and at most N nodes. By pipelining the columns of an $\frac{N}{f(N)} \times f(N)$ matrix through the circuit, the columns of the matrix can all be sorted in $2f(N)$ word steps. By simply hard-wiring the four fixed permutations used in Steps 2, 4, 6 and 8 of columnsort, a matrix of N numbers can be sorted in $O(f(N))$ word steps by columnsort. The network is pictured in Figure 9 for the special case when $f(N) = \Theta(\log N)$. The total number of processors used is clearly $O(N)$ and each processor has bounded degree. Moreover, the processors need to have only a finite amount of state information aside from the ability to store and compare $\Theta(\log N)$-bit numbers. ∎

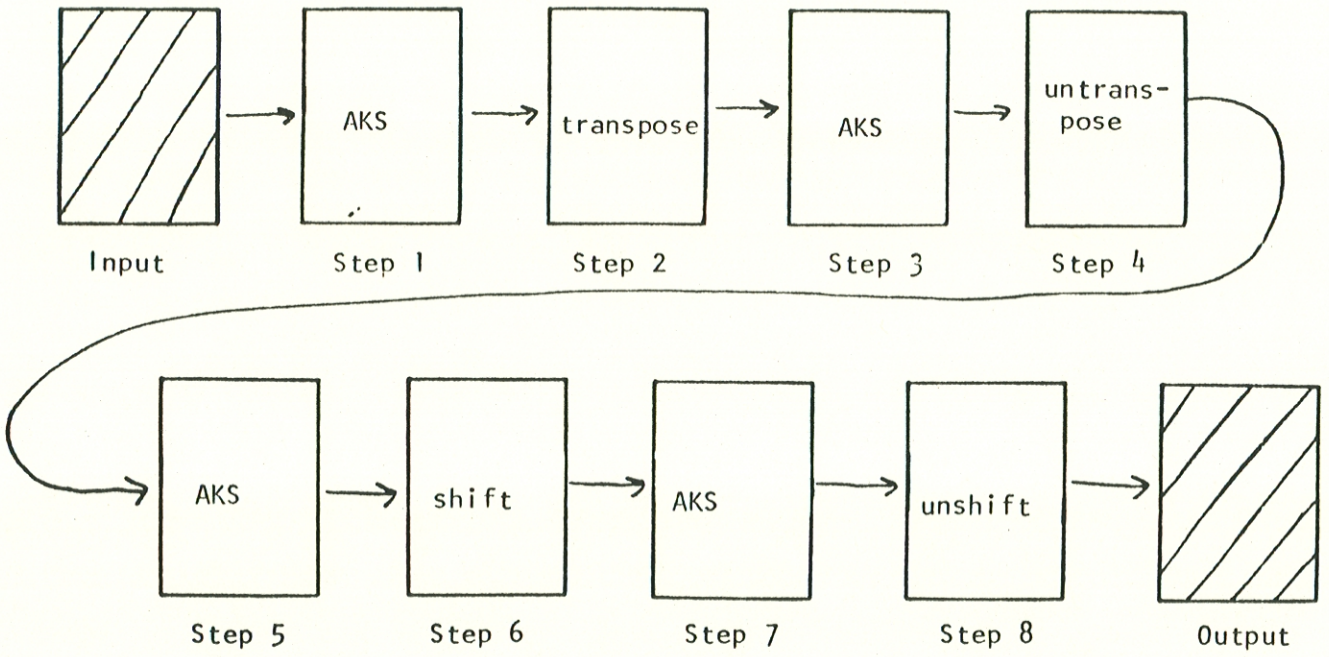

Figure 9: ***Construction of an $O(N)$-node, bounded-degree, $O(\log N)$-time sorting network. The AKS module represents an $(N/\log N)$-input, $O(\log N)$-level sorting circuit.***

Corollary: *When combined with the AKS sorting circuit, the construction in Theorem 1 gives an $O(N)$-node, bounded-degree network that sorts N numbers in $O(\log N)$ word steps.*

As Kruskal [13] observed, the constants in the preceding construction can be improved to the point where the N-number sorting network is degree-3 and has at most N nodes. To prove this, first note that every bounded-degree node can be locally expanded into a constant number of nodes each with degree at most 3. The resulting network still has cN nodes for some constant c and still sorts N numbers in $O(\log N)$ steps. The proof is completed by observing that the same network can actually sort a list of cN numbers by replacing each number with a list of c sorted numbers and each comparison of two numbers with a merging and halving of two c-number lists. The proof that the resulting algorithm actually sorts the cN numbers is not hard to work out. (For example, see [5] or Problem 38 of Section 5.3.4 in [12].)

Because every number to be sorted must be input before the rank of any number can be determined, all N numbers must be input and remembered before any of them can be output. If each node can remember at most a constant number of numbers, this means that at least $\Omega(N)$ nodes are required to sort N numbers in the word model. This kind of argument is used more carefully in the following section, where we prove an $\Omega(N \log N)$-node lower bound for the bit model.

3.2 Results for the Bit Model

As was mentioned in the introduction, the N-input AKS sorting circuit can also be used as an $O(N \log N)$-node bounded-degree network for sorting N numbers in $O(\log N)$ bit-steps. In what follows, we show no fewer nodes could have been used (up to a constant factor), no matter how much time is allowed. The proof applies only to network algorithms that are *when and where oblivious* (i.e., to algorithms for which the time and location of each input bit and output bit is fixed ahead of time, so as not to be dependent on the value of the inputs or the running of the algorithm). In addition, the inputs are supplied just once.

Theorem 2: *Any when and where oblivious network capable of sorting N $(2 \log N)$-bit numbers in the bit model must have $\Omega(N \log N)$ nodes.*

Proof: The basic idea is to show that a large portion of the input bits must be presented to the network before much of the information can be output, thus forcing the network to remember a large number of bits.

Let x_{ij} denote the jth most significant bit of the ith input word and y_{ij} denote the jth bit of the ith sorted word ($0 \leq i < N$, $1 \leq j \leq 2 \log N$). We first show that any input/output schedule for a correct algorithm must input x_{ij} before outputting y_{rs} whenever $s > j$. If this were not the case, then consider the action of the algorithm on the input numbers (written in binary) shown in Figure 10. Every input bit is specified except for x_{ij}. If $x_{ij} = 0$, then the rth sorted number is (in binary) all zeros except for a one in the jth position. Since $s > j$, this means that $y_{rs} = 0$. If $x_{ij} = 1$ on the other hand, then the rth sorted number is (in binary) all zeros except for a one in the jth and sth positions, and $y_{rs} = 1$. Hence, there is no way that the algorithm can always correctly output y_{rs} before seeing x_{ij}.

Because the circuit is when oblivious, the preceding argument means that any sorting circuit must (in particular) input x_{ij} for all $i < N$ and $j \leq \log N$ before outputting y_{rs} for any $r < N$ and $s > \log N$. Consider inputs for which the last $\log N$ bits of the ith input number are fixed to equal i for each $i < N$. Also consider the step of the algorithm in which the last of the first $\log N$ bits of each number is input to the network. At this point the algorithm has not output

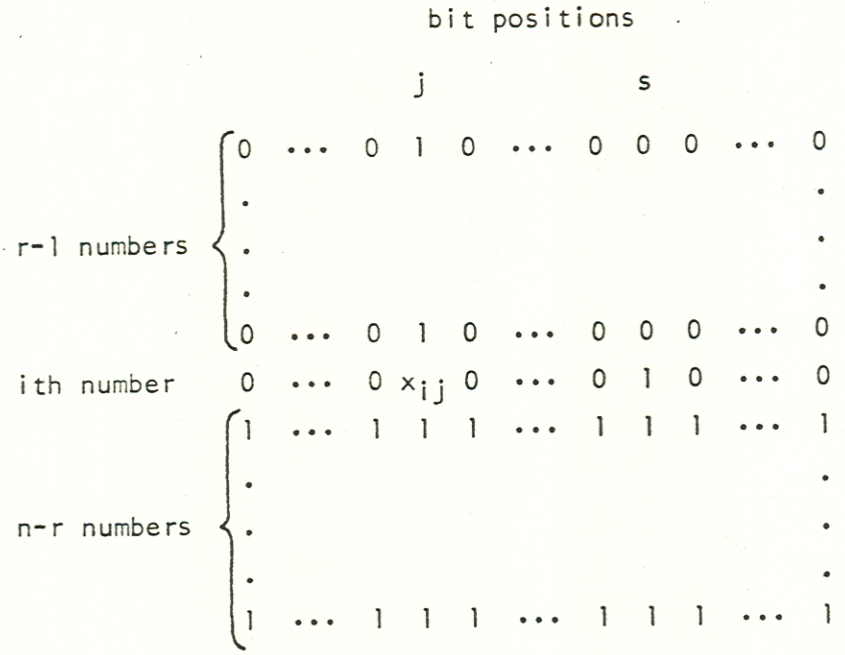

Figure 10: *An example of a set of inputs that forces the network to see x_{ij} before outputting y_{rs}.*

any of the last $\log N$ bits of the numbers and thus there are still $N!$ distinct possible sets of remaining output bits (depending, of course, on the values assigned to the first $\log N$ bits of each input number). Since the network has already received all the unfixed input bits at this point, there is only one possible set of remaining input bits. Hence, the network must have at least $\log(N!) = \Omega(N \log N)$ bits of state. Since each node has only a constant number of bits of state, this means that the network must contain at least $\Omega(N \log N)$ nodes. ∎

It is not difficult to prove the same lower bound for sorting N numbers, each with $(1+\epsilon)\log N$ bits for any constant $\epsilon > 0$. The bound does not hold for k-bit numbers when $k \leq \log N$, however, since, given enough time, it is possible to sort N $(\log N)$-bit numbers with $O(N)$ nodes. The exact number of nodes required to sort N k-bit numbers for arbitrary k has recently been worked out by Siegel [25].

4. Bounds on Information Transfer and Area

In this section, we establish tight bounds on the information transfer and wire area required to sort in both the word and bit models. The lower bounds are the most difficult and are established by lower bounding the information transfer. The upper bounds are established by upper bounding the area and time. Information transfer, area and time are related by Thompson's fundamental $AT^2 \geq \Omega(I^2)$ tradeoff. We refer the unfamiliar reader to [27] for a detailed explanation of area, time and information transfer and for a simple proof of the tradeoff.

4.1 Results for the Word Model

Thompson [27] showed that $I \geq \Omega(N)$ for the problem of sorting N numbers in the word model, and hence that $AT^2 \geq \Omega(N^2)$ for any such circuit. In what follows, we show how to

achieve this bound for any A and T in the range $\Omega(\log N) \leq T \leq O(\sqrt{N})$ with a bounded-degree network containing $O(N)$ nodes. In fact, a simple variant of the network described in Section 3.1 suffices. To prove this, we first consider the case when $T \leq \frac{N^{1/3}}{2}$, and we show how to implement an $\frac{N}{T} \times T$ columnsort procedure. The analysis is divided into two parts. In the first part we show how the permutations in Steps 2, 4, 6 and 8 can be implemented in $O(T)$ steps with an $O(N)$-node subnetwork that has $O(N^2/T^2)$ area for any T in the range $\log N \leq T \leq \sqrt{N}$. In the second part, we show how to sort the columns.

The permutations in Steps 2, 4, 6 and 8 have the nice property that numbers which are adjacent in the input matrix (columnwise) are also adjacent in the output matrix (rowwise for steps 2 and 4, and columnwise for steps 6 and 8). Hence for each of the four permutations, it is easy to partition the input matrix and the output matrix into N/T blocks of T consecutive matrix positions so that the numbers in each block of the input matrix are mapped to a corresponding block of the output matrix. Thus by linking each of the N/T blocks of T inputs to its corresponding block of outputs with a single wire, it is possible to complete the desired permutation in T parallel steps. (Unit-length wires are also used connect adjacent positions of the input and output matrices.) As there are only $O(N/T)$ non-unit-length wires, the resulting network consumes at most $O(N^2/T^2)$ area, as claimed.

It remains to bound the area and time required for the sorting steps. Each sorting step is accomplished with an N/T-number AKS sorting circuit. Applying the Bilardi-Preparata [8] result that the M-input AKS circuit has $O(M^2)$ area, we find that the area of the N/T-number sorting circuit is $O(N^2/T^2)$. Moreover, this circuit has $O(\frac{N}{T}\log\frac{N}{T}) \leq O(N)$ nodes and is capable (by pipelining) of sorting T $\frac{N}{T}$-number columns in $O(T)$ steps (since $T \geq \log N$), as claimed.

The algorithm just described works for any T in the range $\Omega(\log N) \leq T \leq \frac{N^{1/3}}{2}$. The construction can be extended for $T \leq O(\sqrt{N})$ by applying columnsort twice before plugging in an AKS network. In particular, we first use the preceding argument to construct a network with $O(M)$ nodes and $O(M^{4/3})$ area that is capable of sorting M numbers in $O(M^{1/3})$ word steps for arbitrary M. We then apply a $M \times \frac{M}{N}$ columnsort procedure to sort N numbers for T in the range $\frac{N^{1/3}}{2} \leq T \leq N^{1/2}$ where $M = \frac{N^{3/2}}{T^{3/2}}$. Since $T \leq \sqrt{N}$, $M \geq N^{3/4}$ and thus $M \geq 2(\frac{N}{M} - 1)^2$. Hence the $M \times \frac{N}{M}$ columnsort will complete the sorting. The fixed permutations are performed as before using $O(N)$ nodes, $O(T)$ steps and $O(N^2/T^2)$ area. The M-number columns are sorted in turn by the single $O(M)$-node circuit just constructed. The area of the sorting part of the N-number circuit is just $O(M^{4/3}) = O(N^2/T^2)$. The number of nodes is $O(M) = O(\frac{N^{3/2}}{T^{3/2}}) \leq O(N)$ for $T \geq \frac{N^{1/3}}{2}$. The total time taken to sort all the columns is $\frac{N}{M} \cdot O(M^{1/3}) = O(T)$. We summarize this result in the following theorem.

Theorem 3: *For any T in the range $\log N \leq T \leq \sqrt{N}$, there is an $O(N)$-node, bounded-degree network with $O(N^2/T^2)$ area that can sort N numbers in $O(T)$ word steps.*

As before, the constants in the construction can be improved to the point where the network consists of N nodes, each with degree at most three.

4.2 Results for the Bit Model

In what follows, we assume that each number to be sorted consists of k bits where, typically, $k = \Theta(\log N)$. If the k bits of each number are input and output locally in the network, then it is easy to show that $I \geq \Omega(kN)$ and thus that $AT^2 \geq \Omega(k^2N^2)$ for sorting N k-bit numbers in the bit model. For large k, however, it is possible to construct circuits for which $AT^2 \leq O(N^2 k \log N \log^2(\frac{k}{\log N}))$ which is much less than $\Theta(k^2N^2)$ [18]. Hence for large k, it is more efficient to input and output the bits of each number in vastly different parts of the network. If the same were true for $k = \Theta(\log N)$, then the intuition that $AT^2 \geq \Omega(N^2 \log^2 N)$ doesn't hold. Nonetheless, the result is still true, as we show in the following theorem.

Theorem 4: *The information transfer I in any when and where oblivious network for sorting N $(7 \log N)$-bit numbers must be at least $\Omega(N \log N)$ in the bit model.*

Proof: Consider a when and where oblivious network that sorts N $(7 \log N)$-bit numbers, and any partition of the network into *left* and *right halves* that evenly splits the location of the output bits. As before, we let x_{ij} denote the jth most significant bit of the ith input number and y_{ij} denote the jth bit of the ith sorted output number $(0 \leq i < N, 1 \leq j \leq 7 \log N)$. By definition, the partition splits the set of y_{ij}'s in half. (To be precise, the partition might not be exactly half-half, but anything close will do, without changing the structure of the proof.)

The proof consists of proving several seemingly unrelated lower bounds for I based on various "worst-case" computations. At the end, we combine the lower bounds to show that $I \geq \Omega(N \log N)$.

For j in the range $\log N < j \leq 7 \log N$, let r_j be the number of i such that x_{ij} is input to the right half of the network. Similarly, define l_j to be the number of i such that x_{ij} is input to the left half of the network. Clearly, $r_j + l_j = N$ for every j. We first show that

$$I \geq \frac{1}{2} \sum_{j=1+\log N}^{7 \log N} \min(l_j, r_j).$$

Construct a table with rows corresponding to "shifts" p and columns corresponding to output bits y_{ij} for $0 \leq i < N$ and $\log N < j \leq 7 \log N$. By setting the first $\log N$ bits of the ith input number to be $i + p \pmod N$ for each i, the network can be forced to shift (with wraparound) the low order $6 \log N$ bits of each number by any amount p $(0 \leq p < N)$. If for a particular shift p and output y_{ij}, the input bit that is sent to y_{ij} by the shift $(x_{i-p,j})$ is not in the same half as y_{ij}, then place a check in the corresponding position of the table. The number of checks in any row is a natural (and standard) measure of the information transfer that will be required to carry out the corresponding shift.

If y_{ij} is in the right half, then there are l_j checks in the corresponding column. Otherwise there are r_j checks in that column. In either case, the column corresponding to y_{ij} contains at least $\min(l_j, r_j)$ checks. Hence the table contains at least

$$N \sum_{j=1+\log N}^{7 \log N} \min(l_j, r_j)$$

checks overall. This means that at least one of the N rows contains at least

$$\sum_{j=1+\log N}^{7\log N} \min(l_j, r_j)$$

checks. Let p be the shift corresponding to this row and fix the first $\log N$ bits of each input number to force a p-shift of the less significant bits. Without loss of generality, at least

$$\frac{1}{2}\sum_{j=1+\log N}^{7\log N} \min(l_j, r_j)$$

of the less significant input/output bit pairs are input in the left half of the network and output in the right half. Set all the other input bits to zero. The network has now been reduced to accepting a string of

$$\frac{1}{2}\sum_{j=1+\log N}^{7\log N} \min(l_j, r_j)$$

input bits in the left half and outputting the same string in the right half. By a straightforward crossing sequence argument (e.g., see [20]), this means that

$$I \geq \frac{1}{2}\sum_{j=1+\log N}^{7\log N} \min(l_j, r_j),$$

as claimed.

As a special case of the preceding analysis, we can also prove that $I \geq \frac{1}{2}Z$ where Z is the number of i, j pairs ($j > \log N$) for which x_{ij} and y_{ij} are in different halves of the network. This is easily seen by fixing the first $\log N$ bits of each input to force a 0-shift of the less significant bits, and then following the same argument as before.

Since we will not use the first $\log N$ bits of the numbers henceforth, we fix them to be zero for the remainder of the proof. For $j > \log N$, label the jth bit position L or R depending on whether most of the N jth output bit positions are in the left (L) or right (R) half of the network. Unless $I \geq \Omega(N \log N)$ (in which case we are done), at least $2\log N$ of the last $6\log N$ bit positions are labeled L and at least $2\log N$ are lableled R. If not, then (without loss of generality) there are at most $2\log N$ positions with the majority of the output bits in the left half of the network. This means that at most

$$N\log N + 2N\log N + \sum_{\substack{\log N < j \leq 7\log N \\ l_j < r_j}} \min(l_j, r_j)$$

of the $7N\log N$ output bits are output in the left half of the circuit. (At worst, all $N\log N$ of the leading $\log N$ bits of each number are output in the left half, which accounts for the extra $N\log N$ term in the preceding sum.) By assumption, this quantity must be $\frac{7}{2}N\log N$ and hence

$$\sum_{j=1+\log N}^{7\log N} \min(l_j, r_j) \geq \frac{1}{2}N\log N.$$

By the first lower bound proved for I, this means that $I \geq \frac{1}{4} N \log N$. Hence, we can assume in what follows that there are at least $2 \log N$ bit positions of each type.

Divide the $6 \log N$ least significant bit positions into two contiguous segments so that one segment (say the one containing the most significant bits) contains at least $\log N$ bit positions labeled L and so that the other segment contains at least $\log N$ positions labeled R. This can be done by scanning the bit position labels from left to right (most significant positions first) until one of the labels (say L) has been seen $\log N$ times. Partitioning the bit positions at this point gives $\log N$ L labels in the *most significant segment* and at least $\log N$ R labels in the *least significant segment*. Next fix all input bits to zero except for those in the $\log N$ positions labeled L in the most significant segment and those in $\log N$ of the positions labeled R in the least significant segment. Fix the bits in the $\log N$ positions labeled R in the least significant segment so that the ith input number contains the binary representation of i in these positions $(0 \leq i < N)$. Let the input bits in the $\log N$ positions labeled L in the most significant segment vary to induce all $N!$ permutations of the values in the $\log N$ bit positions labeled R in the least significant segment. We are now almost done.

By providing the right half of the network with the values of the least significant $\log N$ output bits in positions labeled R that it is not required to output, the right half of the network must be able to produce all $N!$ combinations of the least significant output bits correctly. Aside from the information transfer I, the right half sees at most

$$Z + \sum_{r_j < l_j}^{\log N < j \leq 7 \log N} \min(l_j, r_j)$$

nontrivial bits of the input. Hence, we know that

$$\sum_{l_j < r_j}^{\log N < j \leq 7 \log N} \min(l_j, r_j) + I + Z + \sum_{r_j < l_j}^{\log N < j \leq 7 \log N} \min(l_j, r_j) \geq \log(N!) .$$

Simplifying and applying two of the previous lower bounds for I gives $5I \geq \log(N!)$ and thus $I \geq \Omega(N \log N)$, as claimed. ∎

The preceding result can be extended for sorting N k-bit numbers for $k < 7 \log N$, but we don't know how to prove it for $k = (1 + \epsilon) \log N$. These lower bounds can, in fact, be achieved by a variety of networks, as we show in the following theorem.

Theorem 5: *For any T in the range $\log N \leq T \leq \sqrt{N \log N}$, there is an $O(N \log N)$-node bounded-degree network with $O(\frac{N^2 \log^2 N}{T^2})$ area that is capable of sorting N $O(\log N)$-bit numbers in $O(T)$ bit steps.*

Proof: The construction for the case when $T = O(\sqrt{N})$ is nearly identical to that in Theorem 3. The first difference is that $\log N$ times as many subnetworks are needed to sort the columns, since the throughput for bit serial computations is slower by a factor of $\Theta(\log N)$. In addition, we must pipeline the bits of each word in the manner described in Section 1.2. The second difference is that the capacity of the $\frac{N}{T}$ interblock wires in the permuters in Steps 2, 4, 6 and 8 of

columnsort must be increased by a factor of $\log N$ for the same reason. As a result, the number of nodes is increased by a factor of $\Theta(\log N)$ and the area is increased by a factor of $\Theta(\log^2 N)$. The time is unchanged. Hence by Theorem 3, the network has $O(N \log N)$ nodes, $O\left(\frac{N^2 \log^2 N}{T^2}\right)$ area and sorts in $O(T)$ steps.

For the case when $\Omega(\sqrt{N}) \leq T \leq O(\sqrt{N \log N})$, we modify the construction in Theorem 3 for $T = \Theta(\sqrt{N})$ by creating $\frac{\sqrt{N} \log N}{T}$ times as many subnetworks to sort the columns, and by increasing the capacity of the $\sqrt{N}$ interblock wires in the permuters by a factor of $\frac{\sqrt{N} \log N}{T}$. As a result, the number of nodes is increased from $\Theta(N)$ by a factor of $\Theta(\log N)$ to a total of $\Theta(N \log N)$, the area is increased from $\Theta(N)$ by a factor of $\Theta\left(\left(\frac{\sqrt{N} \log N}{T}\right)^2\right)$ to a total of $\Theta\left(\frac{N^2 \log^2 N}{T^2}\right)$, and the time has increased from $\Theta(\sqrt{N})$ by a factor of $\Theta\left(\frac{\log N}{(\sqrt{N} \log N)/T}\right)$ to a total of $\Theta(T)$, as claimed. Notice that we cannot decrease A further due to the lower bound on area induced by the number of nodes (Theorem 2). ∎

5. Small-Constant-Factor Networks

All of the sorting networks described thus far involve the AKS sorting network. Although this circuit performs well asymptotically, it performs very poorly for any feasible value of N. In this section, we describe networks that are optimal in every respect except the number of nodes, and that do not use the AKS circuit. As a result, the networks constructed in this section will be what we refer to as *small-constant-factor layouts* (i.e., the true quantities are less than, say, ten times the quantities stated in the Big Oh notation). For simplicity, we will only derive the construction for the word model since the construction for the bit model will be nearly identical.

Theorem 6: *For any T in the range $\log N \leq T \leq \sqrt{N}$, there is a small-constant-factor, bounded-degree network with $O(N^2/T^2)$ area that can sort N numbers in $O(T)$ word steps.*

Proof: The construction is similar to that for Theorem 3, except that several meshes of trees are used to sort columns instead of a single AKS circuit. In [14, 15, 16], we showed how to sort M numbers in $O(\log M)$ word steps using an $O(M^2)$-node, $O(M^2 \log^2 M)$-area mesh of trees. In this application, we use $\log^2 N$ meshes of trees each of size sufficient to sort $\frac{N}{T \log^2 N}$ numbers. This collection of meshes of trees has at most $O(N^2/T^2)$ area and (by pipelining) is capable of sorting $T \log^2 N$ ($\frac{N}{T \log^2 N}$)-number columns in $O(T)$ steps. Hence they can be used in conjunction with a $\frac{N}{T \log^2 N} \times T \log^2 N$ columnsort to achieve the desired bounds. For T near or greater than $N^{1/3}$, two levels of columnsort are needed, just as in the proof of Theorem 3. ∎

Although we did not not analyze the constant factors in the proof of Theorem 6, they are not large. In fact, by using more, smaller meshes of trees and, say, a $\frac{N}{T \log^3 N} \times T \log^3 N$ columnsort, the sorting part of the circuit can be made to have only $o(N^2/T^2)$ area! The only major contribution to the area in such a network are the wires that permute the data before and after it is sorted in the columns, and they are easy to lay out.

The number of nodes in the construction of Theorem 6 is easily seen to be at most $O\left(N + \frac{N^2}{T^2 \log^2 N}\right)$. By using many levels of columnsort, the number of nodes can be reduced to $O(N + \frac{N^{1+\epsilon}}{T^2 \log^2 N})$ for any constant $\epsilon > 0$, without increasing the running time or the area by more than a constant factor. (In fact, the increase in time is polynomial in $1/\epsilon$.)

6. Remarks

Ideas similar to those used to develop columnsort can also be found in the work of Haggkvist and Hell [11]. It is likely that columnsort itself has also been discovered, although we don't know of any references. Judging from the applications developed in this paper, it is clearly an important technique that merits further study. The most important open question at this point is whether or not the technique can be used recursively to construct a simple $O(\log N)$-level sorting circuit. As yet, we have not seen how to do this, but there are many ways in which the technique can be applied. For example, the technique works equally well in a setting in which lists of numbers are *closesorted* (i.e., in a setting where every number is mapped close to its final position). This is similar but stronger than the notion of *nearsort* employed succesfully by Ajtai, Komlos and Szemeredi [2]. As another another example, the technique might work well when combined with some sort of recursive merge procedure. Lastly, the technique seems to work well in a probabilistic setting. At this point, it seems likely that a probabilistic circuit with depth near $2 \log N \log \log N$ can be constructed using columnsort-like ideas. (Similar observations have also been made by Ajtai [1].) There also seems to be plenty of room for improvement.

In addition to the questions relating to small-constant-factor $o(\log^2 N)$-depth circuits, it would be interesting to pin down the bounds for numbers of nodes, area and time for sorting N k-bit numbers for values of k not covered by the results in this paper. Substantial progress along these lines was recently made by Siegel [25].

7. References

[1] M. Ajtai, personal communication, 1984.

[2] M. Ajtai, J. Komlos and E. Szemeredi, "An $O(N \log N)$ Sorting Network," *Proc. 15th ACM Symp. on Theory of Computing*, 1983, pp. 1-9.

[3] D. Angluin and C. Thompson, personal communication, 1983.

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

[5] G. Baudet and D. Stevenson, "Optimal Sorting Algorithms for Parallel Computers," *IEEE Trans. on Computers*, Vol. C-27, No. 1, January 1978, pp. 84-87.

[6] G. Bilardi and F. Preparata, "An Architecture for Bitonic Sorting with Optimal VLSI Performance," *IEEE Transactions on Computers*, to appear.

[7] G. Bilardi and F. Preparata, "A Minimum Area Architecture for $O(\log N)$ Time Sorting," *Proc. 16th ACM Symp. on Theory of Computing*, 1984, pp. 64-70.

[8] G. Bilardi and F. Preparata, "The VLSI Optimality of the AKS Sorting Network," in preparation.

[9] A. Borodin and J. Hopcroft, "Routing, Merging and Sorting on Parallel Models of Computation," *Proc. 14th ACM Symp. on Theory of Computing*, 1982, pp. 338-344.

[10] A. El Gamal, personal communication, 1983.

[11] R. Haggkvist and P. Hell, "Sorting and Merging in Rounds," *SIAM J. of Algebraic and Discrete Methods*, Vol. 3, No. 4, December 1982, pp. 465-473.

[12] D. Knuth, *The Art of Computer Programming: Sorting and Searching*, Addison-Wesley.

[13] C. Kruskal, personal communication, 1984.

[14] F. T. Leighton, "New Lower Bound Techniques for VLSI," *Math Systems Theory*, Vol. 17, No. 1, April 1984, pp. 47-70.

[15] F. T. Leighton, *Complexity Issues in VLSI: Optimal Layouts for the Shuffle-Exchange Graph and Other Networks*, MIT Press, 1983.

[16] F. T. Leighton, "Parallel Computation Using Meshes of Trees," *Proc. 1983 Workshop on Graphtheoretic Concepts in Computer Science*, Osnabruck West Germany, Trauner Verlag, pp. 200-218.

[17] F.T. Leighton, "Tight Bounds on the Complexity of Parallel Sorting," *Proc. 16th ACM Symp. on Theory of Computing*, 1984, pp. 71-80.

[18] F.T. Leighton, *Theory of Parallel Computation and VLSI* lecture notes, 1984.

[19] F. T. Leighton, research in progress.

[20] R. Lipton and R. Sedgewick, "Lower Bounds for VLSI," *Proc. 13th ACM Symp. on Theory of Computing*, 1981, pp. 300-307.

[21] D. S. Parker, "Notes on Shuffle/Exchange-Type Switching Networks," *IEEE Trans. on Computers*, Vol. C-29, No. 3, 1980, pp. 213-222.

[22] M. Patterson, personal communication, 1984.

[23] N. Pippenger, "Parallel Communication with Limited Buffers," *Proc. 25th IEEE Symp. on Foundations of Computer Science*, 1984, pp. 127-136.

[24] J. Reif and L. Valiant, "A Logarithmic Time Sort for Linear Size Networks," *Proc. 15th*

ACM Symp. on Theory of Computing, 1983, pp. 10-16.

[25] A. Siegel, "Optimal Area VLSI Circuits for Sorting," unpublished manuscript, 1984.

[26] H. Stone, "Parallel Processing with the Perfect Shuffle," *IEEE Trans. on Computers*, Vol. C-20, No. 2, 1971.

[27] C. Thompson, "Area-Time Complexity for VLSI," *Proc. 11th ACM Symp. on Theory of Computing*, 1979, pp. 81-88.

[28] C. Thompson, "The VLSI Complexity of Sorting," *IEEE Trans. on Computers*, Vol. C-32, No. 12, 1983.

[29] J. Ullman, *Computational Aspects of VLSI*, Computer Science Press, 1983.

[30] L. Valiant and G. Brebner, "Universal Schemes for Parallel Communication," *Proc. 13th ACM Symp. on Theory of Computing*, 1981, pp. 263-277.