Technical Report No. 97-407

EFFICIENT SORTING ON THE STAR GRAPH INTERCONNECTION NETWORK

Selim G. Akl and Tanya Wolff

Department of Computing and Information Science
Queen's University
Kingston, Ontario
Canada K7L 3N6

June 30, 1998

Abstract

Two algorithms for sorting $n!$ numbers on an n-star interconnection network are described. Both algorithms are based on arranging the $n!$ processors of the n-star in a virtual $(n-1)$-dimensional array. The first algorithm runs in $O(n^3 \log n)$ time. This performance matches that of the fastest previously known algorithm for the same problem. In addition to providing a new paradigm for sorting on the n-star, the proposed algorithm has the advantage of being considerably simpler to state while requiring no recursion in its formulation. Its idea is to sort the input by repeatedly sorting the contents of all rows in each dimension of the $(n-1)$-dimensional array. The second algorithm presented in this paper is more efficient. It runs in $O(n^2)$ time and thus provides an asymptotic improvement over its predecessors. However, it is more elaborate as it uses an existing result for sorting optimally on an $(n-1)$-dimensional array.

1 Introduction

For a positive integer $n > 1$, an *n-star interconnection network*, denoted $\mathcal{S}_n$, is defined as follows:

1. $\mathcal{S}_n$ is an undirected graph with $n!$ vertices, each of which is a processor;
2. The label v of each processor P_v is a distinct permutation of the symbols $\{1, 2, \ldots, n\}$;
3. Processor P_v is directly connected by an edge to each of $n-1$ processors P_u, where u is obtained by interchanging the first and ith symbols of v, that is, if
$$v = v(1)v(2)\ldots v(n),$$
where $v(j) \in \{1, 2, \ldots, n\}$ for $j = 1, 2, \ldots, n$, then
$$u = v(i)v(2)\ldots v(i-1)v(1)v(i+1)\ldots v(n),$$
for $i = 2, 3, \ldots, n$.

Networks $\mathcal{S}_2$, $\mathcal{S}_3$, and $\mathcal{S}_4$ are shown in Fig. 1(a)–(c), respectively.

Because its degree and diameter are both sublogarithmic in the number of vertices [1], the n-star interconnection network has received a fair bit of attention lately [2], [3], [5] - [10], [13], [16], [18] - [26], [29], [43], [45], [47] - [55], [60]. Of particular interest in this paper is the problem of sorting on the n-star. Specifically, given $n!$ numbers, held one per processor of the n-star, it is required to sort these numbers in nondecreasing order. The previous best algorithm for sorting $n!$ numbers on an n-star has a running time of $O(n^3 \log n)$ [43, 54]. (Another algorithm for sorting on the n-star is described in [26], and a lower bound of $\Omega((\log n!)^2)$ on its running time is given, but no upper bound).

In this paper we describe two algorithms for sorting on $\mathcal{S}_n$. The central idea in both algorithms is to map the processors of the n-star onto an $(n-1)$-dimensional array. The first algorithm is relatively simple to state. It uses a generalization of the algorithm in [56] and [57] and runs in $O(n^3 \log n)$ time. This performance matches that of the algorithm in [43, 54]. However, the new algorithm is considerably simpler to state and provides an alternative paradigm for sorting on the n-star. Furthermore, it can be expressed in a concise and straightforward manner without the need for recursion and,

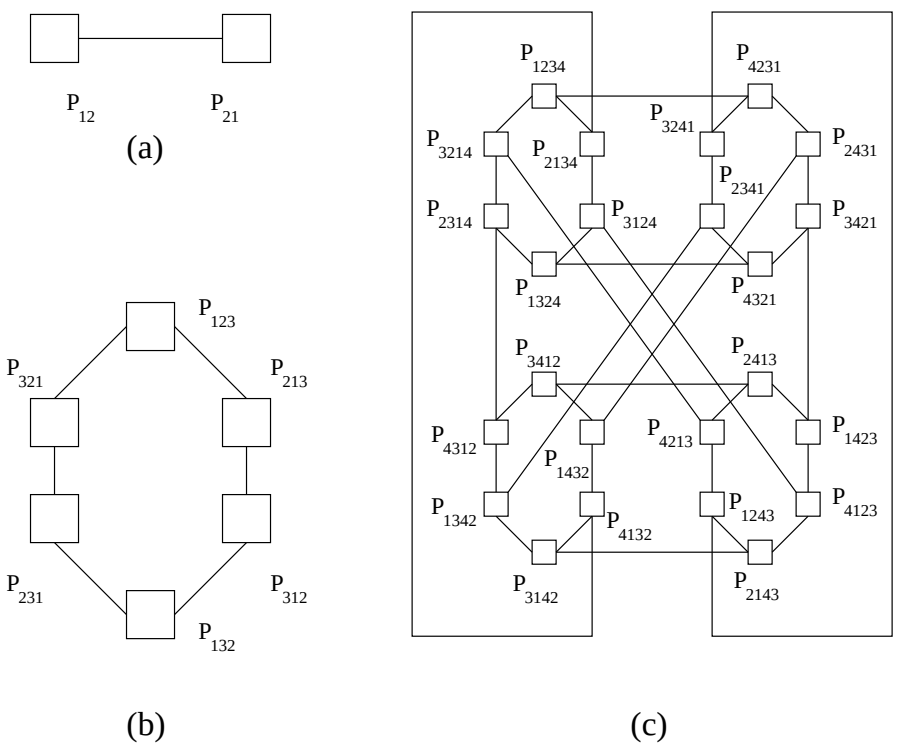

Figure 1: The 2-star, 3-star, and 4-star interconnection networks.

therefore, may be easier (and perhaps more efficient) to implement. The second algorithm is directly based on an algorithm developed in [35] for sorting on an $(n-1)$-dimensional lattice. It runs in $O(n^2)$ time, thus achieving an asymptotic improvement over the fastest previous algorithm [43, 54].

2 Ordering the Processors

As mentioned in Section 1, each of the $n!$ processors of the n-star holds one of the $n!$ numbers to be sorted. To properly define the problem of sorting, it is imperative to impose an order on the processors. This way, one can say that the set of numbers has been sorted once the smallest number of the set resides in the 'first' processor, the second smallest number of the set resides in the 'second' processor, and so on. (If two numbers are equal then the one with the smaller initial index precedes the other). One such order on the processors is now defined.

For a given permutation $v = v(1)v(2)\dots v(n)$, let

(a) $vmax(n) = \max\{v(k) \mid v(k) < v(n),\ 1 \le k \le n-1\}$, if such a $v(k)$ exists; otherwise, $vmax(n) = \max\{v(k) \mid 1 \le k \le n-1\}$.

(b) $vmin(n) = \min\{v(k) \mid v(k) > v(n),\ 1 \le k \le n-1\}$, if such a $v(k)$ exists; otherwise, $vmin(n) = \min\{v(k) \mid 1 \le k \le n-1\}$.

Beginning with the processor label $v = 12\dots n$, the remaining $n! - 1$ labels are arranged in the order produced by algorithm LABELS (n, j) given below. This algorithm is a modification of a procedure described in [28] (see also [58]) for generating permutations. It operates throughout on the array $v = v(1)v(2)\dots v(n)$. Initially, $v(i) = i$, $1 \le i \le n$, and the algorithm is called with $j = 1$. A new permutation is produced every time two elements of the array v are swapped.

Algorithm LABELS(n, j)

Step 1: $i \leftarrow 1$
Step 2: **while** $i \le n$ **do**
(2.1) **if** $n > 2$
then LABELS$(n-1, i)$
end if
(2.2) **if** $i < n$
then if j is odd

then $v(n) \leftrightarrow vmax(n)$
else $v(n) \leftrightarrow vmin(n)$
end if
end if
(2.3) $i \leftarrow i + 1$
end while. □

The order produced by the above algorithm satisfies the following properties:

1. All $(n-1)!$ permutations with $v(n) = \ell$ precede all permutations with $v(n) = \ell'$, for each ℓ and ℓ' such that $1 \leq \ell' < \ell \leq n$.

2. Consider the jth group of $(m-1)!$ consecutive permutations in which $v(m)v(m+1)\ldots v(n)$ are fixed, where $1 \leq j \leq n!/(m-1)!$ and $2 \leq m \leq n-1$. Within this group, the $m-1$ elements of the set

$$\{1, 2, \ldots, n\} - \{v(m), v(m+1), \ldots, v(n)\}$$

take turns in appearing as $v(m-1)$, each repeated $(m-2)!$ times consecutively. They appear in decreasing order if j is odd and in increasing order if j is even.

For example, for $n = 4$, the $4!$ labels, that is, the 24 permutations of $\{1, 2, 3, 4\}$, ordered according to algorithm LABELS, are as follows:

1. 1234	7. 4213	13. 1342	19. 4321
2. 2134	8. 2413	14. 3142	20. 3421
3. 3124	9. 1423	15. 4132	21. 2431
4. 1324	10. 4123	16. 1432	22. 4231
5. 2314	11. 2143	17. 3412	23. 3241
6. 3214	12. 1243	18. 4312	24. 2341

3 Rearranging the Data

An important operation in our sorting algorithms is data rearrangement whereby pairs of processors exchange their data. We now define this operation. Recall that in any interconnection network, two processors that are

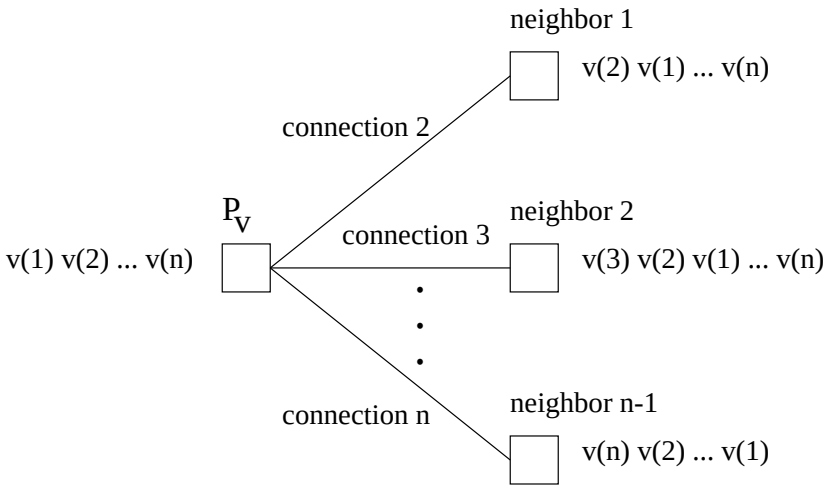

Figure 2: Rearranging the data.

directly connected by an edge are called *neighbors*. A processor can send a datum of constant size to a neighbor in constant time. In $\mathcal{S}_n$ each processor P_v has $n-1$ neighbors P_u. The label u of the kth neighbor of P_v is obtained by swapping $v(1)$ with $v(k+1)$, for $k = 1, 2, \ldots, n-1$. The edges connecting P_v to its neighbors are referred to as *connections* $2, 3, \ldots, n$. If each processor of $\mathcal{S}_n$ holds a datum, then the phrase "rearranging the data over connection i" means that all processors directly connected through connection i exchange their data. Specifically, each processor P_v, where $v = v(1)v(2)\ldots v(i-1)v(i)v(i+1)\ldots v(n)$ sends its datum to processor P_u, where $u = v(i)v(2)\ldots v(i-1)v(1)v(i+1)\ldots v(n)$. This is illustrated in Fig. 2.

4 The Star Viewed As a Multidimensional Array

The $n!$ processors of $\mathcal{S}_n$ are thought of as being organized in a virtual $(n-1)$-dimensional $2 \times 3 \times \cdots \times n$ orthogonal array. This organization is achieved in two steps: The processors are first listed according to the order obtained from algorithm LABELS, then they are placed (figuratively) in this order in the $(n-1)$-dimensional array. For example, for $n = 4$, the 24 processor labels appear in the $2 \times 3 \times 4$ array as shown in Fig. 3.

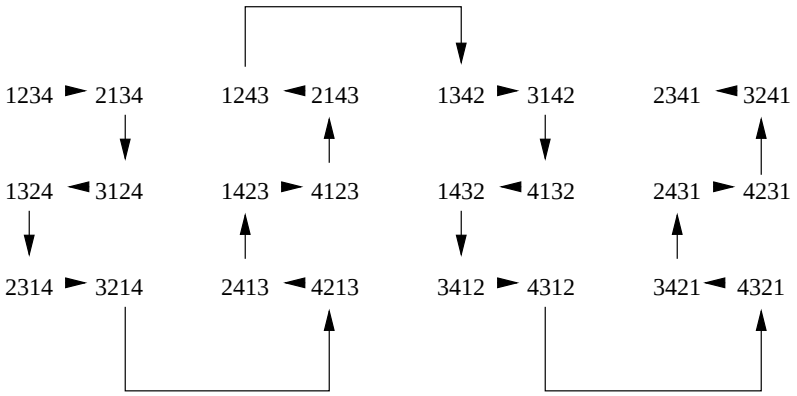

Figure 3: Snakelike order by dimension.

It is important to note that the processors are placed in *snakelike order by dimension*. When $n = 2$, this corresponds to the familiar *snakelike row-major order*. The rule for placing the processors is as follows. Each position of the $(n-1)$-dimensional array has coordinates $I(1), I(2), \ldots, I(n-1)$, where $1 \leq I(k) \leq k+1$. For example, for $n = 4$, the coordinates of the 24 positions in the $2 \times 3 \times 4$ array are as follows:

1,1,1	2,1,1	1,1,2	2,1,2	1,1,3	2,1,3	1,1,4	2,1,4
1,2,1	2,2,1	1,2,2	2,2,2	1,2,3	2,2,3	1,2,4	2,2,4
1,3,1	2,3,1	1,3,2	2,3,2	1,3,3	2,3,3	1,3,4	2,3,4

In dimension k, $1 \leq k \leq n-1$, there are

$$2 \times 3 \times \cdots \times k \times (k+2) \times \cdots \times n$$

(i.e., $n!/(k+1)$) groups of $k+1$ consecutive positions. The positions in each group have the same

$$I(1), I(2), \ldots, I(k-1), I(k+1), I(k+2), \ldots, I(n-1)$$

coordinates (in other words, they differ only in coordinate $I(k)$, with the first position in the group having $I(k) = 1$, the second $I(k) = 2$, and the last $I(k) = k+1$).

The following recursive function $snake_k$ (adapted from [35] for our purposes) maps the coordinates of each position of a multidimensional array into an integer: $snake_1(I(1)) = I(1)$, for $I(1) = 1, 2$, and

$$snake_k(I(1), I(2), \ldots, I(k)) = k!(I(k) - 1) + \begin{cases} snake_{k-1}(I(1), I(2), \ldots, I(k-1)) & \text{if } I(k) \text{ is odd} \\ k! + 1 - snake_{k-1}(I(1), I(2), \ldots, I(k-1)) & \text{if } I(k) \text{ is even,} \end{cases}$$

for $2 \leq k \leq n-1$, $1 \leq I(i) \leq i+1$, and $1 \leq i \leq k$. Thus the function $snake_k$ maps the coordinates $I(1), I(2), \ldots, I(n-1)$ of each position of the $(n-1)$-dimensional $2 \times 3 \times \cdots \times n$ array into a unique integer in the set $\{1, 2, \ldots, n!\}$ according to snakelike order by dimension.

As a result of the (figurative) arrangement just described, processor P_v, such that $v = v(1)v(2)\ldots v(n)$, occupies that position of the $(n-1)$-dimensional array whose coordinates $I(1), I(2), \ldots, I(n-1)$ are given by

$$I(k) = k + 1 - \sum_{j=1}^{k} [v(k+1) > v(j)],$$

for $1 \leq k \leq n-1$, where $[v(k+1) > v(j)]$ equals 1 if $v(k+1) > v(j)$ and equals 0 otherwise.

It should be stressed here that two processors occupying adjacent positions in some dimension k, $1 \leq k \leq n-1$, on the $(n-1)$-dimensional array are not necessarily directly connected on the n-star. Now suppose that in dimension k, each processor in a group of $k+1$ processors occupying consecutive positions holds a datum. Our purpose in what follows is to show that after rearranging the data over connection $k+1$, these $k+1$ data are stored in $k+1$ processors forming a linear array (i.e., the first of the $k+1$ processors is directly connected to the second, the second is directly connected to the third, and so on). To simplify the presentation, we assume in what follows that the $k+1$ consecutive positions (of the multidimensional array) occupied by the $k+1$ processors begin at the position with $I(k) = 1$ and end at that with $I(k) = k+1$ (the argument being symmetric for the case where the $k+1$ processors are placed in the opposite direction, that is, from the position with $I(k) = k+1$ to that with $I(k) = 1$). We proceed in two steps.

1. First we show how the labels of these $k+1$ processors, which are permutations of $\{1, 2, \ldots, n\}$, can be obtained from one another. Let

$v(1)v(2)\ldots v(k+1)\ldots v(n)$ be the label of the last processor in the group, and let d_m, $1 \leq m \leq k+1$, be the position of the mth smallest symbol among $v(1)v(2)\ldots v(k+1)$, where $d_1 = k+1$. Then, the label of the $(k+1-i)$th processor in the group, $1 \leq i \leq k$, is obtained from the label of the $(k+1-i+1)$st processor by exchanging the symbols in positions $k+1$ and d_{i+1}. For example, for $n = 4$, consider the four processors, occupying consecutive positions over dimension 3, whose labels are 1324, 1423, 1432, and 2431. The second of these (i.e., 1423) can be obtained from the third (i.e., 1432) by exchanging the symbol in position 4 (i.e., 2) with the third smallest symbol among 2, 4, 3, and 1 (i.e., 3). This property follows directly from the definition of the LABELS ordering.

2. Now consider a group of $k+1$ processors occupying consecutive positions over dimension k, with each processor holding a datum and the label of the $(k+1)$st processor in the group being $v(1)v(2)\ldots v(k+1)\ldots v(n)$. From the property established in the previous paragraph, the label of the $(k+1-i)$th processor has the $(i+1)$st smallest symbol among $v(1)v(2)\ldots v(k+1)$ in position $k+1$, for $0 \leq i \leq k$. Rearranging the data over connection $k+1$ moves the data from this group to another group of $k+1$ processors in which the label of the $(k+1-i)$th processor has the $(i+1)$st smallest symbol among $v(1)v(2)\ldots v(k+1)$ in position 1, for $0 \leq i \leq k$. In this new group, the label of the $(k+1-i)$th processor can be obtained from that of the $(k+1-i+1)$st by exchanging the symbol in position 1 with that in position d_{i+1}, for $1 \leq i \leq k$. It follows that the processors in the new group are connected to form a linear array. For example, consider once again the four processors, occupying consecutive positions over dimension 3, whose labels are 1324, 1423, 1432, and 2431, as illustrated in Fig. 4. Rearranging the data held by these processors over connection 4 moves them to the processors whose labels are 4321, 3421, 2431, and 1432, respectively.

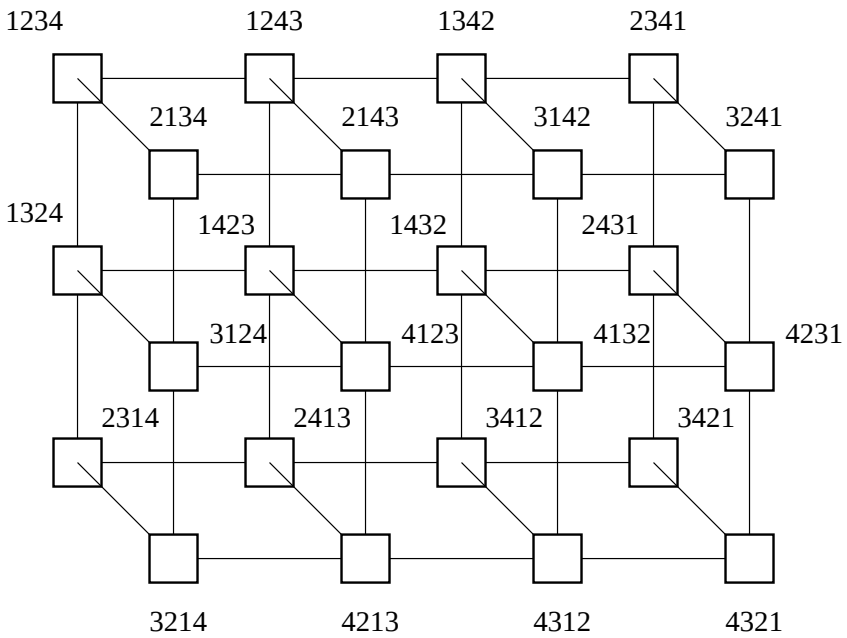

Figure 4: A 4-star viewed as a 3-dimensional array.

5 Sorting on the Linear Array

The algorithm for sorting on the n-star is based on the idea of repeatedly sorting sets of data held by processors forming a linear array. For completeness, we describe an algorithm to perform this operation. Let $P(1), P(2), \ldots, P(k+1)$ be a set of processors forming a linear array (such that, for $2 \leq i \leq k$, $P(i)$ is directly connected to $P(i-1)$ and $P(i+1)$, and no other connections are present), as shown in Fig. 5. Each processor holds a number. We denote the number held by $P(i)$ at any given time by $x(i)$, for $1 \leq i \leq k+1$.

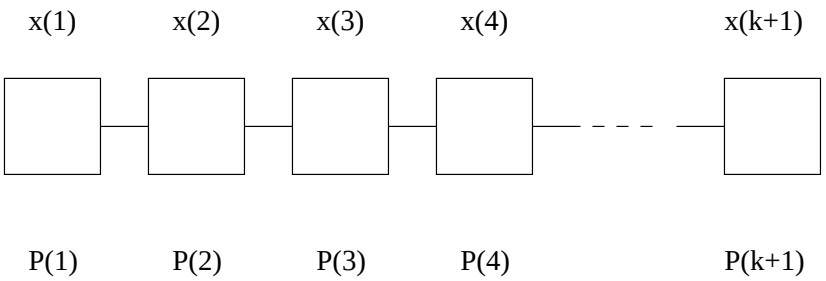

Figure 5: A linear array of processors.

It is desired to sort the set of numbers held by the processors in non-decreasing order (such that, upon termination of the sorting process, the smallest number is held by $P(1)$, the second smallest is held by $P(2)$, and so on). The only operation allowed is a 'comparison-exchange' applied to $x(i)$ and $x(i+1)$, whereby the smaller of the two data ends up in $P(i)$ and the larger in $P(i+1)$. The following algorithm satisfies these conditions.

Algorithm SORTING ON LINEAR ARRAY

for $j = 1$ **to** $k+1$ **do**
  for $i = 1$ **to** k **do in parallel**
    if $i \bmod 2 = j \bmod 2$
    then if $x(i) > x(i+1)$
      then $x(i) \leftrightarrow x(i+1)$
      end if
    end if
  end for
end for. □

A proof that this algorithm sorts correctly in $k+1$ steps, that is, $O(k)$ time, is presented in [5]. It is interesting to note in this context that this algorithm is *oblivious*, that is, the sequence of comparisons it performs is predetermined. Thus, the algorithm's behavior and the number of iterations it requires are not affected by the actual set of numbers to be sorted.

6 A Simple Sorting Algorithm for $\mathcal{S}_n$

Let each of the $n!$ processors of the n-star hold a number. It is required to sort these numbers in nondecreasing order. We now present an algorithm that solves this problem by viewing the n-star as an $(n-1)$-dimensional array. The $n!$ numbers are sorted on the array in snakelike order by dimension, as defined in Section 4. This order matches that of the processors on the $(n-1)$-dimensional array.

The algorithm consists of a number of iterations. During an iteration, the numbers are sorted either in the forward or reverse direction along each of the $n-1$ dimensions in turn. Here, "sorting in the forward (reverse) direction in dimension k" means sorting the numbers in nondecreasing (nonincreasing) order from the processor with $I(k)=1$ to the processor with $I(k)=k+1$. This sorting is performed using the algorithm of Section 5. Note that since in dimension k the $k+1$ processors in each group are not necessarily connected on the n-star to form a linear array, their contents are copied in constant time to another set of $k+1$ processors which *are* connected on the n-star as a linear array. This is done simply by rearranging the numbers to be sorted over connection $k+1$, as described in Section 3. After the sorting is performed, the numbers are brought back to the original processors, also in constant time. Henceforth, we refer to each group of consecutive positions in a given dimension of the multidimensional array, as a 'row'. Thus, a row in dimension k consists of $k+1$ consecutive positions. Let

$$D(k) = \sum_{r=k+1}^{n-1} (I(r) - 1),$$

for $k = 1, 2, \ldots, n-1$, where the empty sum (when $k = n-1$) is equal to 0 by definition, and let

$$N = \sum_{k=2}^{n} \lceil \log k \rceil.$$

The algorithm is as follows:

Algorithm SORTING ON STAR

for $i = 1$ **to** N **do**
  for $k = 1$ **to** $n - 1$ **do**
    for each row in dimension k **do in parallel**
      if $D(k)$ is even
      then sort in the forward direction
      else sort in the reverse direction
      end if
    end for
  end for
end for. □

For example, let $n = 5$, and let the set to be sorted be $\{1, 2, \ldots, 120\}$. The numbers are initially stored in the 120 processors of $\mathcal{S}_5$, one number per processor, in arbitrary order. Once algorithm SORTING ON STAR has been applied, the following arrangement results:

1	2	12	11	13	14	24	23
4	3	9	10	16	15	21	22
5	6	8	7	17	18	20	19
48	47	37	38	36	35	25	26
45	46	40	39	33	34	28	27
44	43	41	42	32	31	29	30
49	50	60	59	61	62	72	71
52	51	57	58	64	63	69	70
53	54	56	55	65	66	68	67
96	95	85	86	84	83	73	74
93	94	88	87	81	82	76	75
92	91	89	90	80	79	77	78
97	98	108	107	109	110	120	119
100	99	105	106	112	111	117	118
101	102	104	103	113	114	116	115

As pointed out earlier, algorithm SORTING ON STAR is nonrecursive, making it straightforward to implement. Its idea of repeatedly going through

the dimensions of the multidimensional array during each iteration of the outer **for** loop is reminiscent of the ASCEND paradigm (originally proposed for the hypercube and related interconnection networks [46]).

Concerning a minor implementation detail, note that each processor computes $D(k)$ over its coordinates $I(1), I(2), \ldots, I(n-1)$ using two variables C and D as follows. Initially, it computes

$$C = \sum_{r=1}^{n-1} (I(r) - 1).$$

Then, prior to the second **for** loop, it computes $D(1)$ as $D = C - (I(1) - 1)$. Finally, at the end of the kth iteration of the second **for** loop, where $k = 1, 2, \ldots, n-2$, it computes $D(k+1)$ as $D = D - (I(k+1) - 1)$.

Analysis. We now show that N iterations, each of which sorts the rows in dimensions $1, 2, \ldots, n-1$, suffice to correctly sort $n!$ numbers stored in an $(n-1)$-dimensional $2 \times 3 \times \cdots \times n$ array. Because algorithm SORTING ON LINEAR ARRAY is oblivious, then so is algorithm SORTING ON STAR. This property allows us to use the 0-1 principle [32]: If we can show that the algorithm correctly sorts any sequence of 0s and 1s, it will follow that the algorithm correctly sorts any sequence of numbers. Suppose then that the input to algorithm SORTING ON STAR consists of an arbitrary sequence of 0s and 1s. Once sorted in nondecreasing order, this sequence will consist of a (possibly empty) subsequence of 0s followed by a (possibly empty) subsequence of 1s. In particular, the sorted sequence will contain at most one (0, 1) pattern, that is, a 0 followed by a 1. (If the input consists of all 0s or all 1s, then of course the output contains no such (0, 1) pattern.) If present, the (0, 1) pattern will appear either in a row in dimension 1, or at the boundary between two adjacent dimension 1 rows (consecutive in snakelike order) such that the 0 appears in one row and the 1 in the next.

Now consider the rows in dimension k, $1 \leq k \leq n-1$. There are $n!/(k+1)$ such rows. A row in dimension k is said to be *clean* if it holds all 0s (all 1s) and there are no 1s preceding it (no 0s following it) in snakelike order; otherwise, the row is *dirty*.

Suppose that an iteration of the algorithm has just sorted the contents of the rows in dimension $n-2$, and let us focus on two such rows, adjacent in dimension $n-1$. For example, for $n = 4$, the coordinates of two dimension 2 rows, adjacent in dimension 3 of the $2 \times 3 \times 4$ array, are

$$\begin{array}{ccc} 1,1,1 & & 1,1,2 \\ 1,2,1 & \text{and} & 1,2,2 \\ 1,3,1 & & 1,3,2, \end{array}$$

respectively. Because these two dimension $n-2$ rows were sorted in snakelike order, they yield at least one clean dimension $n-2$ row when the dimension $n-1$ rows are sorted. To illustrate, suppose that the two dimension 2 rows above contained

$$\begin{array}{ccc} 0 & & 1 \\ 0 & \text{and} & 0 \\ 1 & & 0, \end{array}$$

respectively after they were sorted. Then (regardless of what the other dimension 2 rows contained) a clean dimension 2 row containing all 0s is created after the dimension 3 rows are sorted. Thus, after $\lceil \log n \rceil$ iterations all dimension $n-2$ rows are clean, except possibly for $(n-2)!$ dimension $n-2$ rows contiguous in dimension $n-2$. All dimension $n-1$ rows are now permanently sorted.

By the same argument, $\lceil \log(n-1) \rceil$ iterations are subsequently needed to make all dimension $n-3$ rows clean (except possibly for $(n-3)!$ dimension $n-3$ rows contiguous in dimension $n-3$). In general, $\lceil \log(k+1) \rceil$ iterations are needed to make all dimension $k-1$ rows clean (except possibly for $(k-1)!$ dimension $k-1$ rows contiguous in dimension $k-1$) after all dimension k rows have been permanently sorted, for $k = n-1, n-2, \ldots, 2$. This leaves possibly one dirty dimension 1 row holding the $(0,1)$ pattern. Since that row may not be sorted in the proper direction for snakelike ordering, one final iteration completes the sort. The algorithm therefore sorts correctly in

$$\begin{aligned} N &= \lceil \log n \rceil + \lceil \log(n-1) \rceil + \cdots + \lceil \log 3 \rceil + 1 \\ &< \lceil \log n! \rceil + n \\ &= O(n \log n) \end{aligned}$$

iterations.

Finally, observe that algorithm SORTING ON LINEAR ARRAY is used in dimensions $1, 2, \ldots, n-1$, and hence each iteration of the outer **for** loop in algorithm SORTING ON STAR requires

$$\sum_{k=1}^{n-1} O(k) = O(n^2)$$

time. Since the outer loop is executed $O(n \log n)$ times, the algorithm has a running time of $O(n^3 \log n)$.

The previous discussion suggests that the running time of algorithm SORTING ON STAR may in fact be reduced by a constant factor as follows. Because all dimension $n-1$ rows are permanently sorted after $\lceil \log n \rceil$ iterations of the outer **for** loop, there is no purpose in applying any sorting operations to these rows in subsequent iterations. After another $\lceil \log(n-1) \rceil$ iterations, dimension $n-2$ rows require no more attention. In general, after

$$\sum_{k=j+1}^{n} \lceil \log k \rceil$$

iterations, dimension j rows need not be sorted.

7 An Efficient Sorting Algorithm for $\mathcal{S}_n$

Suppose that M processors, where $M = n_1 n_2 \cdots n_r$ and each n_i is a positive integer, reside in an r-dimensional $n_1 \times n_2 \times \cdots \times n_r$ array such that the processors in each row in dimension k, $1 \leq k \leq r$, are connected to form a linear array. This interconnection network is known as an *r-dimensional lattice* (or *grid*, or *mesh*). Clearly, the special case where $M = n!$, $r = n-1$, and $n_i = i+1$, $1 \leq i \leq n-1$, is of particular interest in this paper, and we refer to the $(n-1)$-dimensional $2 \times 3 \times \cdots \times n$ network as an *$n!$-lattice*. The *diameter* of the $n!$-lattice, that is, the shortest distance separating the farthest two processors, is

$$1 + 2 + \cdots + n - 1 = n(n-1)/2.$$

It follows therefore that a lower bound on the number of steps required to sort $n!$ numbers on the $n!$-lattice is $\Omega(n^2)$.

Algorithms and lower bounds for the problem of sorting $n!$ numbers on the $n!$-lattice can be derived from [12], [14], [15], [27], [30], [31], [33] - [41], [44], [59], [61], [62] and [63]. Some of these algorithms are oblivious (see, for example, [12], [14], [15], [35], [36], and [37]), while others are *input-sensitive*, that is, nonoblivious (see, for example, [30], and [59]); some are *deterministic* (see, for example, [44], [61],[62] and [63]), while others are *randomized* (see, for example, [31]); and, finally, some are efficient *in the worst case* (see, for example, [39]), while others have a good *average* running time (see, for example, [41]). The deterministic and oblivious algorithm of Section 6

can also be used to sort $n!$ numbers on the $n!$-lattice. However, many of these algorithms have running times that exceed n^2 asymptotically and, as a result, are not (worst-case) optimal in light of the $\Omega(n^2)$ lower bound. By contrast, the (deterministic and oblivious) algorithms obtained from [35], [36], and [37] run in $O(n^2)$ time and are therefore optimal in the worst case. Furthermore, like algorithm SORTING ON STAR, the algorithms of [35], [36], and [37] arrange the processors in snakelike order by dimension.

In what follows we provide a brief description of how the algorithm of [37] sorts $M = n_1 n_2 \cdots n_r$ data stored in an r-dimensional lattice (for details, see also [35] and [36]). For each n_i, let u_i and v_i be positive integers such that $u_i v_i = n_i$, $1 \leq i \leq r$. The r-dimensional lattice is viewed as consisting of $u_1 u_2 \cdots u_r$ r-dimensional $v_1 \times v_2 \times \cdots \times v_r$ sublattices, called *blocks*, numbered in snakelike order by dimension from 1 to $u_1 u_2 \cdots u_r$. Alternatively, the r-dimensional lattice consists of $u_1 u_2 \cdots u_r$ $(r-1)$-dimensional *hyperplanes* of $v_1 v_2 \cdots v_r$ processors each, numbered from 1 to $u_1 u_2 \cdots u_r$. The algorithm is given below.

Algorithm SORTING ON LATTICE

Step 1: Sort the contents of block i, $1 \leq i \leq u_1 u_2 \cdots u_r$.
Step 2: Move the data held by the processors in block i to the corresponding processors in hyperplane i, $1 \leq i \leq u_1 u_2 \cdots u_r$.
Step 3: Sort the contents of all r-dimensional $v_1 \times v_2 \times \cdots \times n_r$ 'towers' of blocks.
Step 4: Move the data held by the processors in hyperplane i to the corresponding processors in block i, $1 \leq i \leq u_1 u_2 \cdots u_r$.
Step 5: Sort the contents of
(5.1) All pairs of consecutive blocks i and $i+1$, for all odd i, $1 \leq i \leq u_1 u_2 \cdots u_r$.
(5.2) All pairs of consecutive blocks $i-1$ and i, for all even i, $2 \leq i \leq u_1 u_2 \cdots u_r$. □

The algorithm requires $O(n_1 + n_2 + \cdots + n_r)$ elementary steps (in which a datum is sent from a processor to its neighbor). For an $n!$-lattice (i.e., when $r = n-1$ and $n_i = i+1$, $1 \leq i \leq n-1$) the running time is $O(n^2)$.

The $n!$-lattice is essentially the $(n-1)$-dimensional array of Section 4, augmented with edges connecting the processors. Therefore, any constant-

time exchange of data between two neighboring processors on the $n!$-lattice can be executed on the n-star in constant time. Consequently, using the techniques described in Sections 2 - 5, algorithm SORTING ON LATTICE directly translates into an algorithm for sorting on $\mathcal{S}_n$ in $O(n^2)$ time.

8 Conclusion

Two algorithms for sorting $n!$ numbers on an n-star interconnection network were described. Both algorithms are based on arranging the $n!$ processors of $\mathcal{S}_n$ in a virtual $(n-1)$-dimensional array. The first algorithm sorts the input by using the simple idea of repeatedly sorting all the rows in each dimension. Its running time is $O(n^3 \log n)$. While our main objective was to develop an algorithm for sorting on the n-star, the algorithm of Section 6 is effectively an algorithm for sorting on an $n!$-lattice and is sufficiently interesting in its own right. It is nonrecursive and does not require that data be exchanged between pairs of distant processors: The only operation used, namely, 'comparison-exchange', applies to neighboring processors. Thus, because no routing is needed, the algorithm has virtually no control overhead. Note also that this algorithm, while intended in Section 6 for a $2 \times 3 \times \cdots \times n$ array, is easily extended to arbitrary $n_1 \times n_2 \times \cdots \times n_r$ r-dimensional lattices. In particular, when $n_i = 2$ for $1 \leq i \leq r$, a very simple algorithm is obtained for sorting $M = 2^r$ numbers on a hypercube with M processors in $O(\log^2 M)$ time. (A different derivation of this algorithm appears in [26] where it is shown to be equivalent to the *odd-even-merge* [11] and to the *balanced* [17] sorting circuits.) Other algorithms for sorting on the hypercube, including those with the same running time (i.e., $O(\log^2 M)$) as well as asymptotically faster ones, are considerably more complicated [4, 42].

The second algorithm described in this paper for sorting on $\mathcal{S}_n$ is essentially an adaptation of an algorithm appearing in the literature for sorting on a multidimensional lattice of processors. It runs in $O(n^2)$ time. In view of the $\Omega(n^2)$ lower bound on the running time of any algorithm for sorting on the lattice, this performance is the best that one can hope for using the approach taken in this paper. Nonetheless, this represents an improvement by a factor of $O(n \log n)$ over the best previous algorithm for sorting on $\mathcal{S}_n$. For the problem of sorting on a lattice, however, note that, algorithm SORTING ON LATTICE, while appealing in theory due to its time optimality, is significantly more complex than an algorithm obtained from SORTING ON STAR. Deriving a simple time-optimal algorithm for sorting

on an r-dimensional lattice remains an interesting open problem.

Since $n!$ numbers are sorted optimally using one processor in $O(n! \log n!)$ time, a parallel algorithm using $n!$ processors must have a running time of $O(\log n!)$, that is, $O(n \log n)$, in order to be time optimal. In that sense, the algorithm of Section 7 is suboptimal by a factor of $O(n/\log n)$. However, it is not known at the time of this writing whether an $O(n \log n)$ running time for sorting $n!$ numbers is achievable on the n-star.

A related open question is formulated as follows. Let G be a graph whose set of vertices is V and set of edges is E. A *Hamilton cycle* in G is a cycle that starts at some vertex v of V, traverses the edges in E, visiting each vertex in V exactly once, and finally returns to v. Not all graphs possess a Hamilton cycle; those that do are said to be *Hamiltonian*. As it turns out, it is often useful in parallel computation to determine whether the graph underlying an interconnection network is Hamiltonian. When this is the case, the processors can be viewed as forming a *ring*, that is, a linear array with an additional link connecting the first and last processors. Several useful operations can thus be performed efficiently on the data held by the processors. For example, one such operation is a circular shift. Now, while the n-star can be shown to be Hamiltonian, it is easily verified that the order defined on the processors by algorithm LABELS does not yield a Hamilton cycle for $n > 3$. Thus, for $n = 4$, P_{1234} and P_{2341}, the first and last processors, respectively, are not neighbors. Does there exist an efficient sorting algorithm for the n-star when the processors are ordered to form a Hamilton cycle?

Finally, a randomized sorting algorithms that sorts $n!$ numbers on the n-star in $O(n^3)$ time with high probability is described in [55]. In light of the results of this paper, another interesting question is to develop a faster randomized algorithm for sorting on the n-star.

9 Acknowledgments

This work was supported by the Natural Sciences and Engineering Research Council of Canada. The authors wish to thank Christian Lavault of the University of Paris for interesting discussions.

References

[1] S.B. Akers, D. Harel, and B. Krishnamurthy. The star graph: An

attractive alternative to the n-cube. *Proceedings of the International Conference on Parallel Processing*, 1987, pp. 393–400.

[2] S.B. Akers and B. Krishnamurthy. The fault tolerance of star graphs. *Proceedings of the International Conference on Supercomputing* , Vol. 3, 1987, pp. 270–276.

[3] S.B. Akers and B. Krishnamurthy. A group theoretic model for symmetric interconnection networks. *IEEE Transactions on Computers*, Vol. 38, 1989, pp. 555–566.

[4] S.G. Akl. *Parallel Sorting Algorithms*. Academic Press, Orlando, Florida, 1985.

[5] S.G. Akl. *Parallel Computation: Models and Methods*. Prentice Hall, Upper Saddle River, New Jersey, 1997.

[6] S.G. Akl, J. Duprat, and A.G. Ferreira. Hamiltonian circuits and paths in star graphs. In *Advances in Parallel Algorithms*, I. Dimov and O. Tonev, eds. IOS Press, Sofia, Bulgaria, 1994, pp. 131–143.

[7] S.G. Akl and K. Qiu. Les réseaux d'interconnexion star et pancake. In *Algorithmique parallèle*, M. Cosnard, M. Nivat, and Y. Robert, eds. Masson, Paris, 1992, pp. 171–181.

[8] S.G. Akl and K. Qiu. A novel routing scheme on the star and pancake networks and its applications. *Parallel Computing*, Vol. 19, 1993, pp. 95–101.

[9] S.G. Akl, K. Qiu, and I. Stojmenović. Computing the Voronoi diagram on the star and pancake interconnection networks. *Proceedings of the Canadian Conference on Computational Geometry*, 1992, pp. 353–358.

[10] S.G. Akl, K. Qiu, and I. Stojmenović. Fundamental algorithms for the star and pancake interconnection networks with applications to computational geometry. *Networks*, *Special Issue: Interconnection Networks and Algorithms*, Vol. 23, 1993, pp. 215–226.

[11] K.E. Batcher. Sorting networks and their applications. *Proceedings of the AFIPS Spring Joint Computer Conference*, 1968, pp. 307–314. (Reprinted in *Interconnection Networks for Parallel and Distributed Processing*, C.L. Wu and T.S. Feng, eds. IEEE Computer Society, 1984, pp. 576–583.)

[12] K. Brockmann and R. Wanka. Efficient oblivious parallel sorting on the MasPar MP-1. *Proceedings of the Hawaii Conference on System Sciences*, Vol. 1, 1997, pp. 200–208.

[13] W.K. Chiang and R.J. Chen. The (n, k)-star graph: A generalized star graph. *Information Processing Letters*, Vol. 56, 1995, pp. 259–264.

[14] P.F. Corbett and I.D. Scherson. A unified algorithm for sorting on multidimensional mesh-connected processors. *Information Processing Letters*, Vol. 37, 1991, pp. 225–231.

[15] R. Cypher and J.L.C. Sanz. Optimal sorting on reduced architectures. *Proceedings of the International Conference on Parallel Processing*, Vol. 3, 1988, pp. 308–311.

[16] M. Dietzfelbinger, S. Madhavapeddy, and I.H. Sudborough. Three disjoint path paradigms in star networks. *Proceedings of the IEEE Symposium on Parallel and Distributed Processing*, 1991, pp. 400–406.

[17] M. Dowd, Y. Perl, L. Rudolph, and M. Saks. The balanced sorting network. *Proceedings of the Conference on Principles of Distributed Computing*, 1983, pp. 161–172.

[18] P. Fragopoulou. *Communication and Fault Tolerance Algorithms on a Class of Interconnection Networks.* Ph.D. Thesis, Department of Computing and Information Science, Queen's University, Kingston, Ontario, 1995.

[19] P. Fragopoulou and S.G. Akl. A parallel algorithm for computing Fourier transforms on the star graph. *IEEE Transactions on Parallel and Distributed Systems*, Vol. 5, 1994, pp. 525–531.

[20] P. Fragopoulou and S.G. Akl. Optimal communication algorithms on star graphs using spanning tree constructions. *Journal of Parallel and Distributed Computing*, Vol. 24, 1995, pp. 55–71.

[21] P. Fragopoulou and S.G. Akl. Fault tolerant communication algorithms on the star network using disjoint paths. *Proceedings of the Hawaii International Conference on System Sciences*, Vol. 2, 1995, pp. 5–13.

[22] P. Fragopoulou and S.G. Akl. A framework for optimal communication on a subclass of Cayley graph based networks. *Proceedings of the International Conference on Computers and Communications*, 1995, pp. 241–248.

[23] P. Fragopoulou and S.G. Akl. Efficient algorithms for global data communication on the multidimensional torus network. *Proceedings of the International Parallel Processing Symposium*, 1995, pp. 324–330.

[24] P. Fragopoulou and S.G. Akl. Edge-disjoint spanning trees on the star network with applications to fault tolerance. *IEEE Transactions on Computers*, Vol. 45, 1996, pp. 174–185.

[25] P. Fragopoulou, S.G. Akl, and H. Meijer. Optimal communication primitives on the generalized hypercube network. *Journal of Parallel and Distributed Computing*, Vol. 32, 1996, pp. 173–187.

[26] D.M. Gordon. Parallel sorting on Cayley graphs. *Algorithmica*, Vol. 6, 1991, pp. 554–564.

[27] Y. Han and Y. Igarashi. Time lower bounds for sorting on multidimensional mesh-connected processor arrays. *Proceedings of the International Conference on Parallel Processing*, Vol. 3, 1988, pp. 194–197.

[28] B.R. Heap. Permutations by interchanges. *The Computer Journal*, Vol. 6, 1963, pp. 293–294.

[29] J.S. Jwo, S. Lakshmivarahan, and S.K. Dhall. Embedding of cycles and grids in star graphs. *Proceedings of the IEEE Symposium on Parallel and Distributed Processing*, 1990, pp. 540–547.

[30] M. Kaufmann, J.F. Sibeyn, and T. Suel. Beyond the worst-case bisection bound: Fast sorting and routing on meshes. *Proceedings of the European Symposium on Algorithms*, Lecture Notes in Computer Science No. 979, Springer-Verlag, Berlin, 1995, pp. 75–88.

[31] M. Kaufmann, and J.F. Sibeyn. Randomized multipacket routing and sorting on meshes. *Algorithmica*, Vol. 17, 1997, pp. 224–244.

[32] D.E. Knuth. *The Art of Computer Programming*, Vol. 3. Addison-Wesley, Reading, Massachusetts, 1973.

[33] M. Kunde. A general approach to sorting on 3-dimensionally mesh-connected arrays. *Proceedings of the Conference on Parallel Processing (CONPAR)*, Lecture Notes in Computer Science No. 237, Springer-Verlag, Berlin, 1986, pp. 329–337.

[34] M. Kunde. Lower bounds for sorting on mesh-connected architectures. *Acta Informatica*, Vol. 24, 1987, pp. 121–130.

[35] M. Kunde. Optimal sorting on multi-dimensionally mesh-connected computers. *Proceedings of the Symposium on Theoretical Aspects of Computer Science*, Lecture Notes in Computer Science No. 247, Springer-Verlag, Berlin, 1987, pp. 408–419.

[36] M. Kunde. Routing and sorting in mesh-connected arrays. *Proceedings of the Aegean Workshop on Computing*, Lecture Notes in Computer Science No. 319, Springer-Verlag, Berlin, 1988, pp. 423–433.

[37] M. Kunde. Concentrated regular data streams on grids: Sorting and routing near the bisection bound. *Proceedings of the Symposium on Foundations of Computer Science*, 1991, pp. 141–150.

[38] M. Kunde. *Routing and Sorting on Grids.* Habilitationsschrift, Department of Computer Science, Technical University of Munich, Munich, Germany, 1992.

[39] M. Kunde. Block gossiping on grids and tori: Deterministic sorting and routing match the bisection bound. *Proceedings of the European Symposium on Algorithms*, Lecture Notes in Computer Science No. 726, Springer-Verlag, Berlin, 1993, pp. 272–283.

[40] M. Kunde, R. Niedermeier, and P. Rossmanith. Faster sorting and routing on grids with diagonals. *Proceedings of the Symposium on Theoretical Aspects of Computer Science*, Lecture Notes in Computer Science No. 775, Springer-Verlag, Berlin, 1994, pp. 225–236.

[41] M. Kunde, R. Niedermeier, K. Reinhardt, and P. Rossmanith. Optimal average case sorting on arrays. *Proceedings of the Symposium on Theoretical Aspects of Computer Science*, Lecture Notes in Computer Science No. 900, Springer-Verlag, Berlin, 1995, pp. 503–514.

[42] F.T. Leighton. *Introduction to Parallel Algorithms and Architectures.* Morgan Kaufmann, San Mateo, California, 1992.

[43] A. Menn and A.K. Somani. An efficient sorting algorithm for the star graph interconnection network. *Proceedings of the International Conference on Parallel Processing*, Vol. 3, 1990, pp. 1–8.

[44] D. Nassimi and S. Sahni. Bitonic sort on a mesh-connected parallel computer. *IEEE Transactions on Computers*, Vol. C-27, 1979, pp. 2–7.

[45] M. Nigam, S. Sahni, and B. Krishnamurthy. Embedding Hamiltonians and hypercubes in star interconnection graphs. *Proceedings of the International Conference on Parallel Processing*, Vol. 3, 1990, pp. 340–343.

[46] F.P. Preparata and J.E. Vuillemin. The cube-connected cycles: A versatile network for parallel computation. *Communications of the ACM*, Vol. 24, 1981, pp. 300–309.

[47] K. Qiu. *The Star and Pancake Interconnection Networks: Properties and Algorithms*. Ph.D. Thesis, Department of Computing and Information Science, Queen's University, Kingston, Ontario, 1992.

[48] K. Qiu and S.G. Akl. Load balancing, selection and sorting on the star and pancake interconnection networks. *Journal of Parallel Algorithms and Applications*, Vol. 2, 1994, pp. 27–42.

[49] K. Qiu and S.G. Akl. On some properties of the star graph. *Journal of VLSI Design, Special Issue on Interconnection Networks*, Vol. 2, 1994, pp. 389–396.

[50] K. Qiu, S.G. Akl, and H. Meijer. On some properties and algorithms for the star and pancake interconnection networks. *Journal of Parallel and Distributed Computing*, Vol. 22, 1994, pp. 16–25.

[51] K. Qiu, H. Meijer, and S.G. Akl. Parallel routing and sorting on the pancake network. *Proceedings of the International Conference on Computing and Information*, Lecture Notes in Computer Science, No. 497, Springer-Verlag, Berlin, 1991, pp. 360–371.

[52] K. Qiu, H. Meijer, and S.G. Akl. Decomposing a star graph into disjoint cycles. *Information Processing Letters*, Vol. 39, 1991, pp. 125–129.

[53] K. Qiu, H. Meijer, and S.G. Akl. On the cycle structure of star graphs. *Congressus Numerantium*, Vol. 96, 1993, pp. 123–141.

[54] S. Rajasekaran and D.S.L. Wei. Selection, routing and sorting on the star graph. *Proceedings of the International Parallel Processing Symposium*, 1993, pp. 661–665.

[55] S. Rajasekaran and D.S.L. Wei. Selection, routing, and sorting on the star graph. *Journal of Parallel and Distributed Computing*, Vol. 41, 1997, pp. 225–233.

[56] K. Sado and Y. Igarashi. Some parallel sorts on a mesh-connected processor array and their time efficiency. *Journal of Parallel and Distributed Computing*, Vol. 3, 1986, pp. 398–410.

[57] I. Scherson, S. Sen, and A. Shamir. Shear-sort: A true two-dimensional sorting technique for VLSI networks. *Proceedings of the International Conference on Parallel Processing*, 1986, pp. 903–908.

[58] R. Sedgewick. Permutation generation methods. *Computing Surveys*, Vol. 9, 1977, pp. 137–164.

[59] J.F. Sibeyn. Sample sort on meshes. *Proceedings of the Euro-par Conference*, Lecture Notes in Computer Science, Springer-Verlag, Berlin, 1997, to appear.

[60] S. Sur and P.K. Srimani. A fault tolerant routing algorithm in star graphs. *Proceedings of the International Conference on Parallel Processing*, Vol. 3, 1991, pp. 267–270.

[61] C.D. Thompson and H.T. Kung. Sorting on a mesh-connected parallel computer. *Communications of the ACM*, Vol. 20, 1977, pp. 263–271.

[62] R. Wanka. Fast general sorting on meshes of arbitrary dimension without routing. Technical Report No. 87, Department of Computer Science, Paderborn University, Paderborn, Germany, 1991.

[63] R. Wanka. *Paralleles Sortieren auf mehrdimensionalen Gittern.* Ph.D. Thesis, Paderborn University, 1994. In German.