Generalized Algorithm for Parallel Sorting on Product Networks

Antonio Fernández, *Member*, *IEEE*, and Kemal Efe, *Member*, *IEEE*

Abstract—We generalize the well-known odd-even merge sorting algorithm, originally due to Batcher [2], and show how this generalized algorithm can be applied to sorting on product networks.

If G is an arbitrary factor graph with N nodes, its r-dimensional product contains N^r nodes. Our algorithm sorts N^r keys stored in the r-dimensional product of G in $O(r^2F(N))$ time, where $F(N)$ depends on G. We show that, for any factor graph G, $F(N)$ is, at most, $O(N)$, establishing an upper bound of $O(r^2\,N)$ for the time complexity of sorting N^r keys on any product network.

For product networks with bounded r (e.g., for grids), this leads to the asymptotic complexity of $O(N)$ to sort N^r keys, which is optimal for several instances of product networks. There are factor graphs for which $F(N) = O(\log^2 N)$, which leads to the asymptotic running time of $O(\log^2 N)$ to sort N^r keys. For networks with bounded N (e.g., in the hypercube $N = 2$, fixed), the asymptotic complexity becomes $O(r^2)$.

We show how to apply the algorithm to several cases of well-known product networks, as well as others introduced recently. We compare the performance of our algorithm to well-known algorithms developed specifically for these networks, as well as others. The result of these comparisons led us to conjecture that the proposed algorithm is probably the best deterministic algorithm that can be found in terms of the low asymptotic complexity with a small constant.

Index Terms—Sorting, interconnection networks, product networks, algorithms, odd-even merge.

1 Introduction

RECENTLY, there has been an increasing interest in product networks in literature. These networks have interesting topological properties that make them especially suitable for parallel algorithms. Well-known examples of product networks include hypercubes, grids, and tori. Many other product networks have been proposed recently, such as products of de Bruijn networks [9], [29], products of Petersen graphs [26], and mesh-connected trees [8], [9] (which are products of complete binary trees). As a general class, routing properties of product networks have been studied in [4], [12]. Topological and embedding properties of product networks have been analyzed in [9], and VLSI complexity of product networks has been analyzed in [10].

There is a large body of literature on algorithms developed specifically for some of the popular product networks like hypercubes and grids. The problem with these algorithms is that they are not portable between different architectures. For example, a sorting algorithm developed for a hypercube architecture will not run on a grid architecture, even though both hypercubes and grids are product networks. The question we ask in this paper, and in [11], is the following: Is it possible to develop algorithms for product networks capitalizing on their common properties only, so that the same algorithm can be made to run on all product networks? We show in this paper that, at least for the sorting problem, the answer is “yes.” In [11], we presented a collection of similarly general algorithms for other problems, including matrix multiplication, pointer-jumping, FFT computation, transitive closure of a matrix, etc. What is most interesting about these algorithms is that, when mapped to specific architectures, their running times turn out to be either optimal, or as efficient as the best known algorithms specifically developed for the corresponding architectures. For example, running time of the sorting algorithm presented in this paper is optimal for grids, while at the same time, it meets the running time of the well-known Batcher algorithm when mapped to hypercubes.

The sorting algorithm of this paper is based on a generalization of the classic Batcher algorithms. In [2], Batcher presented two efficient sorting networks. Algorithms derived from these networks have been presented for a number of different parallel architectures, like the shuffle-exchange network [31], the grid [23], [32], the cube-connected cycles [28], and the mesh of trees [25].

One of Batcher’s sorting networks has, as main components, subnetworks that sort bitonic sequences. A bitonic sequence is the concatenation of a nondecreasing sequence of keys with a nonincreasing sequence of keys, or the rotation of such a sequence. Sorting algorithms based on this method are generally called “bitonic sorters.” Several papers have been devoted to generalizing bitonic sorters [3], [18], [22], [24].

- *A. Fernández is with the Dipartimento de Arquitectura y Tecnogia de Computadores, Escuela Universitaria de Informatica, Ctra. Valencia Km. 7, 28031 Madrid, Spain. E-mail: anto@eui.upm.es.*
- *K. Efe is with the Center for Advanced Computer Studies, University of Southwestern Louisiana, Lafayette, LA 70504-4330. E-mail: efe@cacs.usl.edu.*

Manuscript received 5 Jan. 1995.
For information on obtaining reprints of this article, please send e-mail to: tpds@computer.org, and reference IEEECS Log Number 101074.

The main components of the other sorting network proposed by Batcher in [2] are subnetworks that merge two sorted sequences into a single sorted sequence. He called these "odd-even merging" networks. Several papers generalized this network to merging of k sorted sequences, where $k > 2$. These are generally called k-way merging networks. Examples are Green [14], who constructed a network based on four-merge, and Drysdale and Young [7], van Voorhis [34], Tseng and Lee [33], Parker and Parberry [27], Liszka and Batcher [21], and Lee and Batcher [17], who constructed networks based on multiway merging.

Similarly, other algorithms based on a multiway-merge concept have been presented, the most commonly known being Leighton's Columnsort algorithm [20]. Initially, the objective of this algorithm was to show the existence of bounded-degree $O(n)$-node networks that can sort n keys in $O(\log n)$ time. In this network, the permutations at each phase are hard-wired and the sortings are done with AKS networks, which limits its applicability for practical purposes. However, Aggarwal and Huang [1] showed that it is possible to use Columnsort as a basis and apply it recursively. Parker and Parberry's network cited above is also based on a modification of Columnsort. These algorithms behave nicely when the number of keys is large compared with the number of processors.

In this paper, we develop another multiway-merge algorithm that merges several sorted sequences into a single sorted sequence. From this multiway-merge operation, we derive a sorting algorithm, and we show how to use this approach to obtain an efficient sorting algorithm for any homogeneous product network. In its basic spirit, our multiway-merge algorithm has some similarities with a recent version of Columnsort [19, p. 261], but ours outperforms Columnsort due to some fundamental differences in the interpretation of this basic concept. First, our algorithm is based on a series of merge processes recursively applied, while Columnsort is based on a series of sorting steps. The only time we use sorting is for N^2 keys. Columnsort, on the other hand, uses several recursive calls to itself in order to merge. Second, by observing some fundamental relationships between the structural properties of product networks and the definition of sorted order, we are able to avoid most of the routing steps required in the Columnsort algorithm.

Among the main results of this paper, we also show that the time complexity of sorting N^r keys for any N^r-node r-dimensional product graph is bounded above as $O(r^2N)$. We also illustrate special cases of product networks for which the running time of our algorithm reduces to $O(r^2)$, $O(N)$, and $O(\log^2 N)$ to sort N^r keys.

On the grid and the mesh-connected trees [8], [9] with bounded number of dimensions, the proposed algorithm runs in asymptotically-optimal $O(N)$ time. On the r-dimensional hypercube, the algorithm has asymptotic complexity $O(r^2)$, which is the same as that of Batcher's odd-even merge sorting algorithm on the hypercube [2]. Although there are asymptotically faster sorting algorithms for the hypercube [6], they are not practically useful for a reasonable number (less than 2^{20}) of keys [19]. We note, however, that there are randomized algorithms which perform better on hypercubic networks than the Batcher algorithm in practice [5]. Adaptation of such approaches for product networks appears to be an interesting problem for future research.

For products of de Bruijn networks [9], [29], our approach yields the asymptotic complexity of $O(r^2 \log^2 N)$ time to sort N^r keys, which reduces to $O(\log^2 N)$ time when the number of dimensions is fixed. The same running time can be obtained for products of shuffle-exchange networks also, because products of shuffle-exchange networks are equivalent in computational power (i.e., in asymptotic complexity of algorithms) to products of de Bruijn networks [9]. This running time is the same as the asymptotic complexity of sorting N^r keys on the N^r-node de Bruijn or shuffle-exchange network by Batcher algorithm.

Finally, we can summarize the main contributions of this paper to

- Develop a new multiway merging algorithm as a basis for the sorting algorithm,
- Show how to effectively implement it for homogeneous product networks, regardless of the topology of the factor network used to build it,
- Obtain generalized upper bounds on the running time required for sorting on any homogeneous product network,
- Show that, for several important instances of homogeneous product networks, the upper bound derived matches the running time of the most-popular algorithms developed specifically for these networks.

This paper is organized as follows. In Section 2, we present the basic definitions and the notation used in this paper. We also discuss some of the topological properties of product networks needed for the proposed sorting algorithm. In Section 3, we present our multiway-merge algorithm and show how to use it for sorting. In Section 4, we show how to implement the multiway-merge sorting algorithm on any homogeneous product network and analyze its time complexity. In Section 5, we apply the algorithm to several homogeneous product networks and obtain the corresponding time complexities. The conclusions of this paper are given in Section 6.

2 Definitions, Notation, and Relevant Properties of Product Networks

Let G be an N-node connected graph.

Definition 1. *Given a graph G with vertex set* $V_G = \{0, 1, \cdots, N - 1\}$ *and arbitrary edge set* E_G, *the* r-dimensional homogeneous *product of G, denoted* PG_r, *is the graph whose vertex set is* $V_{PG_r} = \{0, 1, \cdots, N-1\}^r$ *and whose edge set is* E_{PG_r}, *defined as follows: Two vertices* $x = x_r x_{r-1} \cdots x_1$ *and* $y = y_r y_{r-1} \cdots y_1$ *are adjacent in* PG_r *if and only if both of the following conditions are true:*

1) *x and y differ in exactly one symbol position,*
2) *if i is the differing symbol index, then* $(x_i, y_i) \in E_G$.

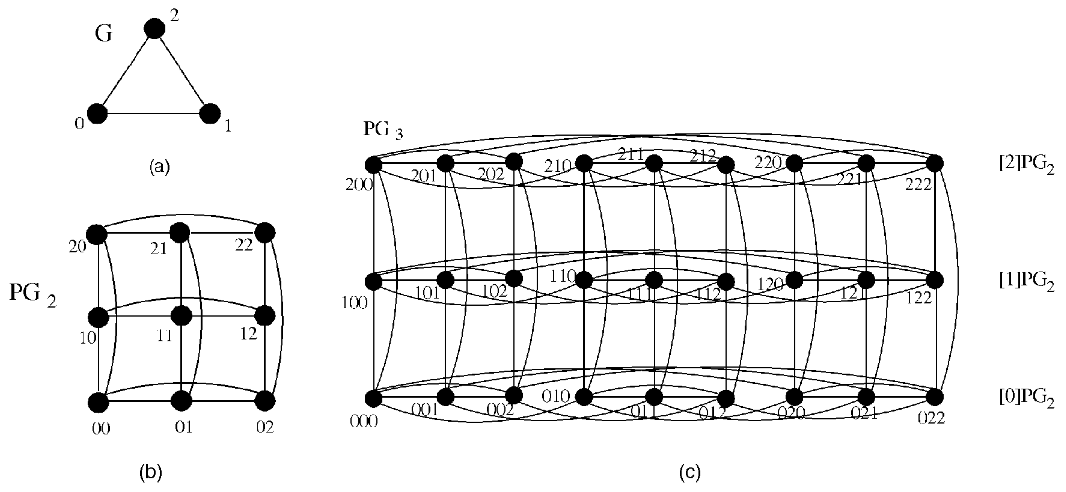


Fig. 1. Recursive construction of multidimensional product networks: (a) the factor graph; (b) two-dimensional product; (c) three-dimensional product.

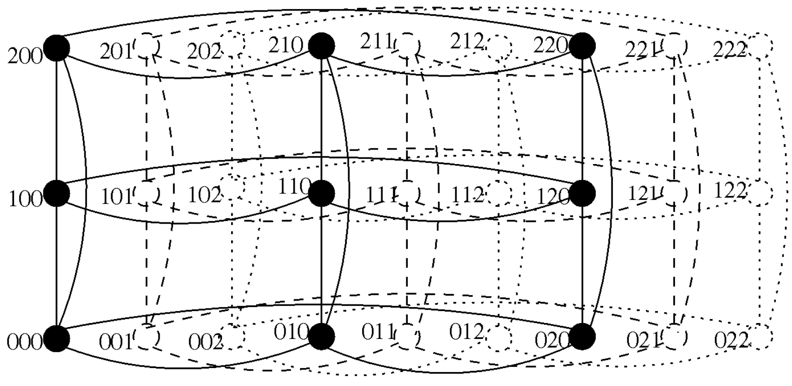


Fig. 2. Two-dimensional product graphs obtained by erasing the dimension-one connections from the three-dimensional product graph of Fig. 1c. The solid lines define the $[0]PG_2^1$ graph as they connect the nodes that have "0" at position 1 of their labels. Similarly, dashed lines define the $[1]PG_2^1$ graph, and dotted lines define the $[2]PG_2^1$ graph.

In this paper, we assume that the r-tuple label for a node of PG_r is indexed as $1 \cdots r$, with 1 referring to the rightmost position index and r referring to the leftmost position index.

At a more intuitive level, the construction of PG_r from PG_{r-1}, where $PG_1 = G$, can be described by referring to Fig. 1. Let x be a node of PG_{r-1}, and let $[u]PG_{r-1}$ be the graph obtained by prefixing every vertex x in PG_{r-1} by u, so that a vertex x becomes ux. First, place the vertices of PG_{r-1} along a straight line, as shown in Fig. 1. Then, draw N copies of PG_{r-1} such that the vertices with identical labels fall in the same column. Next, extend the vertex labels to obtain $[u]PG_{r-1}$, for $u = 0, 1, \cdots, N-1$. Finally, connect the columns in the interconnection pattern of the factor graph G, such that ux is connected to $u'x$, if and only if $(u, u') \in E_G$. Fig. 1 illustrates this construction process for two and three dimensional products of the factor graph shown in Fig. 1a.

In this construction, we use $[u]PG_{r-1}$ to refer to the uth copy of PG_{r-1}, whose labels are extended by the prefix u. For example, in Fig. 1c, vertical edges are dimension-three edges, connecting three copies of PG_2 graphs denoted as $[0]PG_2$, $[1]PG_2$, and $[2]PG_2$. We can extend this notation to allow us to add a new symbol at *any* position of the vertex labels. For this purpose, we use $[u]PG_{r-1}^i$ to mean that the vertex labels of PG_{r-1} are extended by inserting the value u at position i. As a result, the symbol at position j moves to position $j + 1$, for $j = i, \cdots, r - 1$. Definition 1 allows us to observe that the construction of the preceding paragraph could be restated for $[u]PG_{r-1}^i$ for any position i, not just the leftmost position.

Another way we can obtain the $[u]PG_{r-1}^i$ subgraphs, for $u = 0, 1, \cdots, N-1$, is by erasing all the dimension-i edges in PG_r, and keeping the nodes whose labels have u at position i. For example, in Fig. 2, we illustrate the two-dimensional product graphs obtained by erasing the dimension-one connections of the three-dimensional product graph of Fig. 1c. This process can be repeated recursively, and described by a simple extension of our notation: We use $[u, v]PG_{r-2}^{i,j}$ to refer to subgraphs isomorphic to PG_{r-2} obtained by erasing the connections at dimensions i and j from PG_r. A particular subgraph so obtained can be distinguished by its unique combination of $[u, v]$ values at index positions i and j, respectively. The notation is similarly extended for erasing arbitrary number of dimensions, and the order of the values in square brackets corresponds to the order of the superscripts.

The notation $[u]PG_{r-1}^{i}$ naturally defines an ordering between the PG_{r-1} subgraphs of PG_r. In general, $[u]PG_{r-1}^{i}$ is the uth copy of the PG_{r-1} subgraph at dimension i, since u is at position i in the labels of every node in $[u]PG_{r-1}$. This notion of subgraph ordering is important for the purposes of this paper. This subgraph ordering rule can be carried to the general case of $[u_1, \cdots, u_k]PG_{r-k}^{i_1,\cdots,i_k}$ in a number of different ways. We will define a particular subgraph ordering method with certain useful properties that will ultimately induce an ordering rule for the individual nodes of the product graph, representing the order of sorted data. Before doing so, we need a convention for labeling the nodes of the factor graph comprising the product graph.

For an arbitrary factor graph G with N nodes, the vertex labels $0, \cdots, N-1$ define the ascending order of data when sorted. As a matter of convention, if G contains a Hamiltonian path, then it is beneficial (although not required for the correctness of the proposed sorting algorithm) to label the nodes in the order they appear in the Hamiltonian path. If G does not contain a Hamiltonian path, then it is always possible to embed a linear array in G with dilation three and congestion two, and then label the nodes in the order they appear on the linear array [19]. Again, this is not required for the correctness of the proposed algorithm, but such labeling of nodes would provide a speed improvement over an arbitrary labeling, by a constant factor.

For the product graph, our algorithm uses the *snake order*, defined as follows.

DEFINITION 2. Snake order *for the r-dimensional product graph PG_r:*

1) *If $r = 1$, the snake order corresponds to the order used for labeling the nodes of G.*
2) *If $r > 1$, suppose that the snake order has been already defined for PG_{r-1}. Then,*
   (a) *$[u]PG_{r-1}^{r}$ has the same order as PG_{r-1} if u is even, and reverse order if u is odd.*
   (b) *if $u < v$ then any value in $[u]PG_{r-1}^{r}$ precedes any value in $[v]PG_{r-1}^{r}$.*

The snake order for product graphs is closely related to Gray-code sequences in that, when the data is sorted in the snake order, tracing the data in the sorted order visits the nodes of the product network in the same order that they would appear if the node labels are written in a Gray-code sequence. The binary Gray-code sequence is well-known, and it has the fundamental property that any two consecutive terms in the sequence differ in exactly one bit. Here we are dealing with N-ary symbols instead of binary symbols. Therefore, we need to use N-ary Gray-code sequences.

First, recall the definition of Hamming distance and Hamming weight. Let s, z be r-tuples from $\{0, 1, \cdots, N-1\}^r$, then the Hamming distance between s and z is $D(s, z) = \sum_{i=1}^{r} |s_i - z_i|$, where $|s_i - z_i|$ is the absolute value of $s_i - z_i$. The Hamming weight of an r-tuple s is $W(s) = \sum_{i=1}^{r} s_i$. Here, we allow one or more of the elements of the r-tuples to be the special "all" symbol "$*$." If any of the symbols in the r-tuple is the "all" symbol, then its index position is omitted whenever the r-tuple is involved in the computation of Hamming distances and Hamming weights.

We say that a sequence Q_r is an *N-ary Gray-code sequence of order r* if its elements are all the r-tuples in $\{0, 1, \cdots, N-1\}^r$, and any two consecutive elements in it have unit Hamming distance. Consequently, the Hamming weights of two consecutive terms will have different parity. We use $R(Q_r)$ to denote the sequence obtained by listing the elements of Q_r in reverse order.

The definition below shows one way to construct N-ary Gray-code sequences of arbitrary order recursively. Let $[u]Q_k$ denote the sequence obtained by prefixing each element of Q_k with the symbol u if u is even, or by prefixing each element of $R(Q_k)$ with u, if u is odd.

DEFINITION 3. *An N-*ary Gray-code sequence of order *r, denoted Q_r, can be obtained as*

1) $Q_1 = \{0, 1, \cdots, N-1\}$,
2) $Q_r = CON\{[u]Q_{r-1} \mid u = 0, 1, \cdots, N-1\}$, *where "$CON\{\}$" indicates concatenation of the sequences inside the curly brackets.*

EXAMPLE. If $N = 3$, the three-ary Gray-code sequences of order r, for $r = 1$, 2 and 3, are:

- for $r = 1$, $Q_1 = \{0, 1, 2\}$,
- for $r = 2$, $Q_2 = \{00, 01, 02, 12, 11, 10, 20, 21, 22\}$,
- for $r = 3$, $Q_3 = \{000, 001, 002, 012, 011, 010, 020, 021, 022, 122, 121, 120, 110, 111, 112, 102, 101, 100, 200, 201, 202, 212, 211, 210, 220, 221, 222\}$,

In Fig. 3, this snake order is highlighted for the product graph of Fig. 1c.

The reverse operation of recursively constructing a Gray-code sequence Q_r from N sequences of Q_{r-1} is splitting the sequence Q_r into N sequences of Q_{r-1}. One way to do this is to take the first N^{r-1} terms of the original sequence as the first Q_{r-1} sequence, the next N^{r-1} terms as the second Q_{r-1} sequence, and so on. In the first sequence, all the labels will have 0 as their leftmost symbol, so we denote it as $[0]Q_{r-1}$. Similarly, all the labels in the uth sequence will have u as their leftmost symbol, so the corresponding sequence is denoted as $[u]Q_{r-1}$. Again, there is nothing special about the leftmost symbol; we can split the Q_r sequence into N sequences of Q_{r-1} based on the similarity of labels at any symbol position. We use $[u]Q_{r-1}^{i}$ to denote the subsequences obtained from Q_r, such that each term in the uth subsequence contains the value u in position i. We are mainly interested in the subsequences $[u]Q_{r-1}^{1}$, for $u = 0, \cdots, N-1$. For given u, the elements of $[u]Q_{r-1}^{1}$ come from positions u,

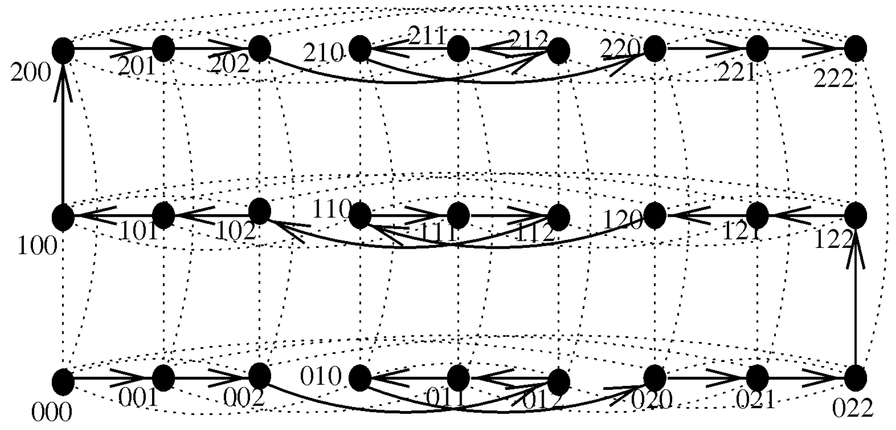


Fig. 3. Snake order for the example product graph.

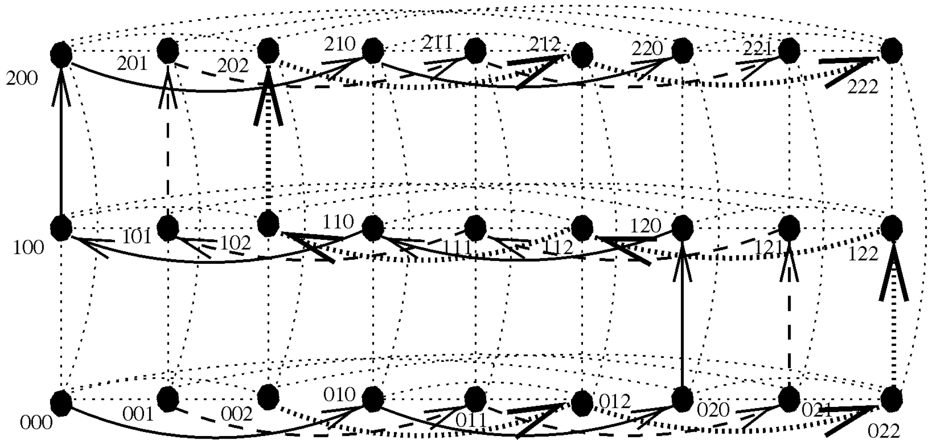


Fig. 4. Subsequences obtained from a snake-ordered sequence. Solid lines connect the nodes with 0 as their rightmost term, yielding the $[0]G_2^1$ sequence. The dashed and dotted lines yield the $[1]G_2^1$ and $[2]G_2^1$ sequences, respectively.

$2N - u - 1$, $2N + u$, $4N - u - 1$, $4N + u$, and so on, in Q_r. For example, if the solid lines of Fig. 3 show the Q_3 sequence, Fig. 4 illustrates the three $[u]Q_2^1$ subsequences obtained from it.

From the identity relationship between the Gray-code sequence Q_r, and the snake order for the nodes of PG_r, it follows that, if PG_r contains a sequence of keys sorted in snake order, the keys on the subgraph $[u]PG_{r-1}^1$ are also sorted in snake order, and are in the positions u, $2N - u - 1$, $2N + u$, $4N - u - 1$, $4N + u$, etc., of the whole sequence. This observation is important for the proposed sorting algorithm.

We are now ready to define a similar ordering method between subgraphs of a product graph. We define this ordering with the aid of Q_r sequences. Consider dividing Q_r into N^{r-1} groups of N *consecutive* terms each. We observe from Definition 3 that any two elements in the same group would differ in their rightmost symbols only. These groups can also be obtained by replacing the rightmost symbol of each label in Q_r by the "$*$" symbol, and then grouping together all the labels with a Hamming distance of zero. Members belonging to the same group can be identified by the common values at positions 2, $\cdots$, r of their labels, called the "group labels." In the product graph, the members of such a group correspond to the labels of nodes in the same G subgraph along dimension one. Now, these G subgraphs of a product graph can be ordered by simply following the Gray-code order defined on their corresponding group labels.

More specifically, we use $[*]Q_{r-1}^1$ to denote the group sequence obtained from Q_r in this fashion, where $*$ stands for all of 0, 1, $\cdots$, $N - 1$. For example, given Q_3 as above, its group sequence is

$$[*]Q_2^1 = \{00*, 01*, 02*, 12*, 11*, 10*, 20*, 21*, 22*\}$$

where $*$ stands for the set of elements in a group, which, for this example, consists of $\{0, 1, 2\}$. More explicitly, the above sequence corresponds to

$$[*]Q_2^1 = \{00\{0,1,2\}, 01\{2,1,0\}, 02\{0,1,2\}, 12\{2,1,0\}, 11\{0,1,2\}, 10\{2,1,0\}, 20\{0,1,2\}, 21\{2,1,0\}, 22\{0,1,2\}\}.$$

Here, a group label q (i.e., an element q of $[*]Q_{r-1}^1$) is of the form $q = q_r q_{r-1} \cdots q_2\{Q_1\}$ if the Hamming weight of q is even, or of the form $q = q_r q_{r-1} \cdots q_2\{R(Q_1)\}$ if the Hamming weight of q is odd. Moreover, two successive group labels still have unit Hamming distance. As an example, Fig. 5

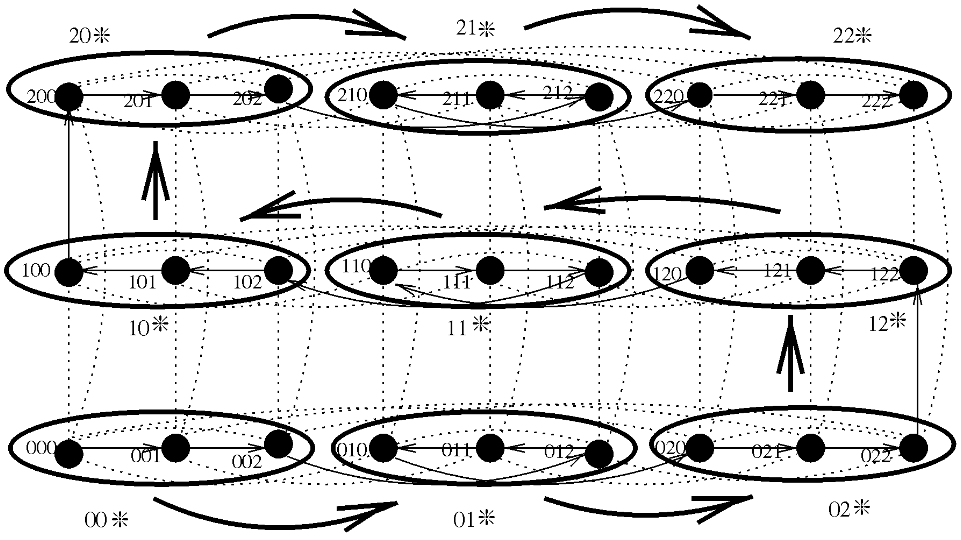


Fig. 5. Snake order for the *G* subgraphs of a product graph.

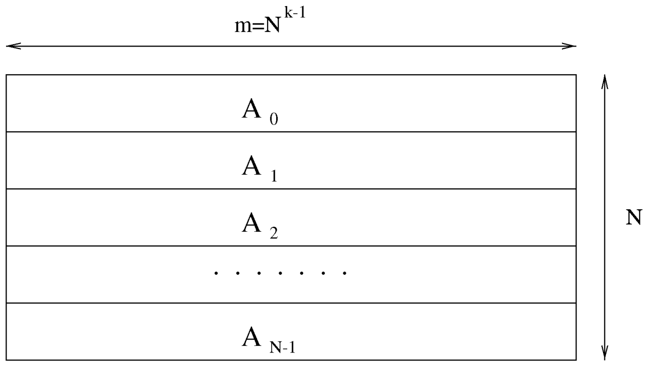


Fig. 6. Initial situation before the merge process starts. Each sorted sequence is represented as a horizontal block.

shows the ordering of the *G* subgraphs of the three-dimensional product graph of Fig. 1c. We say that a *G*-subgraph is an even (resp. odd) subgraph, if the Hamming weight for its group label is even (resp. odd).

In this paper, we will be mainly interested in ordering the PG_2 subgraphs of the *r*-dimensional product graph. To define such an order, we can extend the above notation and write $[*,*]Q^{1,2}_{r-2}$ to identify the set of PG_2-subgraphs at dimensions {1, 2}. Two consecutive elements of $[*,*]Q^{1,2}_{r-2}$ will again have unit Hamming distance and, thus, the elements of $[*,*]Q^{1,2}_{r-2}$ will be ordered in Gray-code sequence, corresponding to the snake order between PG_2 subgraphs. Again, a group label (i.e., an element *q* of $[*,*]Q^{1,2}_{r-2}$) can have even or odd Hamming weight, and the corresponding PG_2 subgraph can be said to be even or odd.

3 Multiway-Merge Sorting Algorithm

This section develops the basic steps of the proposed sorting algorithm without regard to any specific network. For this discussion, it does not even matter whether the algorithm is performed sequentially or in parallel. The subsequent sections will give the implementation details for product networks.

A *sorted sequence* is defined as a sequence of keys $(a_0, a_1, \cdots, a_{m-1})$ such that $a_0 \le a_1 \le \cdots \le a_{m-1}$. The *multiway-merge algorithm* combines *N* sorted sequences

$$A_u = (a_{u,0}, a_{u,1}, \cdots, a_{u,m-1}),$$

for $u = 0, \cdots, N-1$, into a single sorted sequence

$$S = (s_0, s_1, \cdots, s_{mN-1}).$$

We will assume *m* to be some power of *N*, $m = N^{k-1}$, where $k > 2$ and, hence, the resulting sorted sequence, *S*, will contain N^k keys.

The heart of the proposed sorting algorithm is the multiway-merge operation. Thus, we will spend much of our time discussing this merging process. In order to build an intuitive understanding of the basic idea of the merge operation, we assume that the keys are arranged in a two-dimensional block, as shown in Fig. 6. Here, each row is a sorted sequence that is going to be merged with the other rows. This is not to imply a two-dimensional organization of the data in product networks. When implementing the algorithm in product networks, each row of data (containing $m = N^{k-1}$ keys) in Fig. 6 will be initially stored on a $(k-1)$-dimensional subgraph of the product graph. The two-dimensional organization in Fig. 6 is for the reader's convenience in visualizing what happens to the data at various steps of the algorithm, so that we can use the terms "row" and "column" in order to refer to groups of keys that are subjected to the same step of algorithm. Other than this, our use of the terms "row" and "column" should not be interpreted to imply the physical organization of data in a two dimensional array.

We also assume the existence of an algorithm which can sort N^2 keys. We make no assumption about the efficiency of this algorithm as yet. In Section 5, we discuss several possible ways to obtain efficient algorithms for this purpose. The purpose of this assumption is to maintain the generality of the discussions, independent of the factor network used to build the product network.

To show the correctness of the algorithm, we will use the zero-one principle due to Knuth [15]. The zero-one principle states that if an algorithm based on compare-exchange operations is able to sort any sequence of zeros and ones, then it sorts any sequence of arbitrary keys.

3.1 Multiway-Merge Algorithm

Here, we consider how to merge N sorted sequences, $A_i = (a_{u,0}, a_{u,1}, \cdots, a_{u,m-1})$, for $u = 0, \cdots, N-1$, into a single large sorted sequence. The initial situation is pictured in Fig. 6. The merge operation consists of the following steps:

Step 1. Distribute the keys of *each* sorted sequence A_u among N sorted subsequences $B_{u,v}$, for $u = 0, \cdots, N-1$ and $v = 0, \cdots, N-1$. The subsequence $B_{u,v}$ will have the form $(a_{u,v}, a_{u,2N-v-1}, a_{u,2N+v}, a_{u,4N-v-1}, a_{u,4N+v}, \cdots)$, for $u = 0, \cdots, N-1$ and $v = 0, \cdots, N-1$. This is equivalent to writing the keys of each A_u on a $\frac{m}{N} \times N$ array in snake order (as shown in Fig. 7) and then reading the keys column-wise, so that column v of the array becomes $B_{u,v}$, for $v = 0, \cdots, N-1$. Note that each subsequence $B_{u,v}$ is sorted, since the keys in it are in the same relative order as they appeared in A_u.

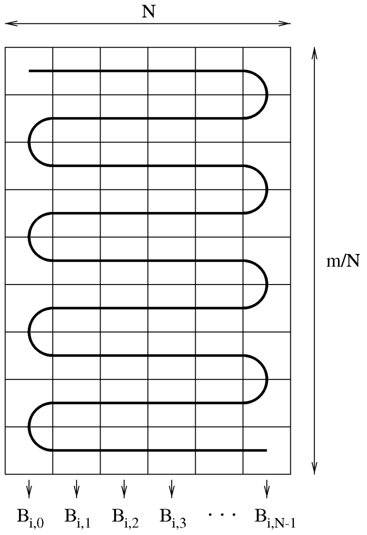
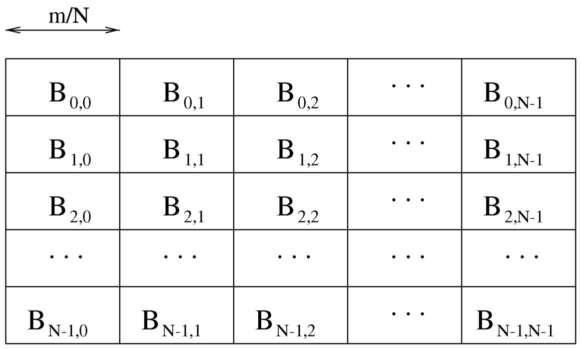


Fig. 7. Distribution of the keys of A_u (i.e., the uth row in Fig. 6) among the N subsequences $B_{u,v}$. The thick line represents the keys of A_u in snake order. $B_{u,v}$ sequence is obtained by reading the vth column from top to bottom.



$B_{0,0}$	$B_{0,1}$	$B_{0,2}$	. . .	$B_{0,N-1}$
$B_{1,0}$	$B_{1,1}$	$B_{1,2}$	. . .	$B_{1,N-1}$
$B_{2,0}$	$B_{2,1}$	$B_{2,2}$	. . .	$B_{2,N-1}$
. . .	. . .	. . .	. . .	. . .
$B_{N-1,0}$	$B_{N-1,1}$	$B_{N-1,2}$	. . .	$B_{N-1,N-1}$

Fig. 8. Situation after Step 1: Each sequence A_u, has been distributed into N subsequences $B_{u,v}$. Each of the subsequences contains m/N elements and is still sorted.

Fig. 8 illustrates the situation after the completion of this process. Each of the N rows contains N sorted subsequences $B_{u,v}$, where each $B_{u,v}$ box in Fig. 8 corresponds to a column of keys in Fig. 7 written horizontally.

EXAMPLE. If for some u, $A_u = \{1, 2, 3, 4, 5, 6, 7, 8, 9\}$ and $N = 3$, then $B_{u,0} = \{1, 6, 7\}$; $B_{u,1} = \{2, 5, 8\}$; $B_{u,2} = \{3, 4, 9\}$.

Step 2. Merge the N subsequences $B_{u,v}$ found in column v of Fig. 8 into a single sorted sequence C_v, for $v = 0, \cdots, N-1$. This is done in parallel for all columns by a recursive call to the multiway-merge process if the total number of keys in the column, m, is at least N^3. If the number of keys in a column of Fig. 8 is N^2, a sorting algorithm for sequences of length N^2 is used (we already assumed the existence of such an algorithm above), because a recursive call to the merge process would not make much progress when $m = N^2$ (this point will be cleared at the end of this section). At the end of this step, we write the resulting subsequences vertically in N columns of length m each. The situation after this step is illustrated in Fig. 9.

Step 3. Interleave the sequences C_v into a single sequence

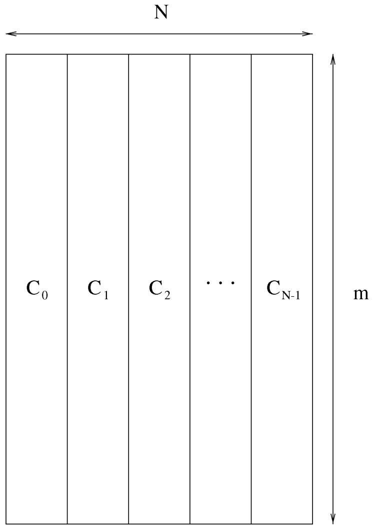


Fig. 9. Situation after merging the subsequences in each column. The keys are sorted from top to bottom.

$D = (d_0, d_1, \cdots, d_{mN-1})$. The sequence D is formed simply by reading the $m \times N$ array of Fig. 9 in row-major order starting from the top row. The sequence D is redrawn in Fig. 10 from the C_v sequences of Fig. 9, with no change in the organization of data. Fig. 10 is identical to Fig. 9, except that we regard it as one big sequence to be read in row-major order.

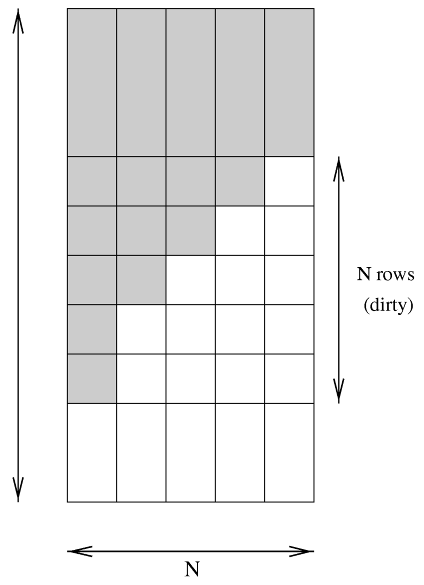


Fig. 10. Sequence *D* obtained after interleaving. The order goes from top to bottom by reading the data in row-major order. The shaded area is filled with zeros and the white area is filled with ones. The boundary area has at most *N* rows, as shown in Lemma 1.

We prove below that the sequence D is now "almost" sorted. This situation is shown in Fig. 10. If the keys being sorted can only take values of zero or one, the shaded area represents the position of zeros and the white area represents the position of ones. As D is obtained by reading the values in row-major order, the potential *dirty area* (window of keys not sorted) has length no larger than N^2. This fact will be shown in Lemma 1.

Step 4. Clean the dirty area. To do so we start by dividing the sequence D into m/N subsequences of N^2 consecutive keys each. We denote these subsequences as E_z, for $z = 0, \cdots, \frac{m}{N} - 1$. The zth subsequence has the form $E_z = (d_{zN^2}, d_{zN^2+1}, \cdots, d_{zN^2+N^2-1})$. That is, the first N rows in Fig. 10 (or, equivalently, in Fig. 9) are concatenated to obtain E_1, the next N rows are concatenated to obtain E_2, and so on (see Fig. 11a).

We then independently sort the subsequences (rows in Fig. 11a) in alternate orders by using the algorithm which we assumed available for sorting N^2 keys. E_z is transformed into a sequence F_z (see Fig. 11b), where F_z contains the keys of E_z sorted in nondecreasing order if z is even or in nonincreasing order if z is odd, for $z = 0, \cdots, \frac{m}{N} - 1$.

Now, we apply two steps of odd-even transposition between the sequences F_z, for $z = 0, \cdots, \frac{m}{N} - 1$ (i.e., in the vertical direction of Fig. 11b). In the first step of odd-even transposition, each pair of sequences F_z and F_{z+1}, for z even, are compared element by element. Let $f_{z,t}$ denote the tth term in F_z, where $t = 0 \cdots N^2 - 1$. Two sequences G_z and G_{z+1} are formed (not shown in the figure) where $g_{z,t} = min\{f_{z,t}, f_{z+1,t}\}$ and $g_{z+1,t} = max\{f_{z,t}, f_{z+1,t}\}$. In the second step of the odd-even transposition, G_z and G_{z+1} for z odd are compared in a similar manner to form the sequences H_z and H_{z+1}. Fig. 11c shows the situation after the two steps of odd-even transposition.

Finally, we independently sort each sequence H_z in nondecreasing order if z is even, and nonincreasing order if z is odd. This generates sequences I_z, for $z = 0, \cdots, \frac{m}{N} - 1$ (see Fig. 7d). The final sorted sequence S is the concatenation of the sequences I_z in snake order, and this completes the merging algorithm.

We need to show that the process described actually merges the sequences. To do so, we use the zero-one principle mentioned earlier.

LEMMA 1. *When sorting an input sequence of zeros and ones, the sequence D obtained after the completion of Step 3 is sorted except for a dirty area which is never larger than* N^2.

PROOF. Assume that we are merging sequences of zeros and ones. Let x_u be the number of zeros in sequence A_u, for $u = 0, \cdots, N - 1$. The rest of keys in A_u are ones. Step 1 breaks each sequence A_u into N subsequences $B_{u,v}$, $v = 0, \cdots, N - 1$. It is easy to observe, from the way Step 1 is implemented, that the number of zeros in a subsequence $B_{u,v}$ is $\lfloor x_u / N \rfloor + \epsilon_{uv}$, where ϵ_{uv} is either zero or one. Therefore, for a given u, the sequences $B_{u,v}$ can differ from one another in their number of zeros by, at most, one.

At the start of Step 2, each column v is composed of the subsequences $B_{u,v}$ for $u = 0, \cdots, N - 1$. At the end of Step 2, all the zeros are at the beginning of each sequence C_v. The number of zeros in each sequence C_v is the sum of the number of zeros in $B_{u,v}$ for fixed v and $u = 0, \cdots, N - 1$. Thus, two sequences C_v can differ from each other by, at most, N zeros. In Step 3, we interleave the N sorted sequences into the sequence D by taking one key at a time from each sequence C_v. Since any two sequences C_v can differ in their number of zeros by, at most, N, and since there are N sequences being interleaved, the length of the window of keys where there is a mixture of ones and zeros is, at most, N^2. □

Now we can show how the last step actually cleans the dirty area in the sequence.

LEMMA 2. *The sequence S obtained (by concatenation of sequences* I_z *in snake order) after the completion of Step 4 is sorted.*

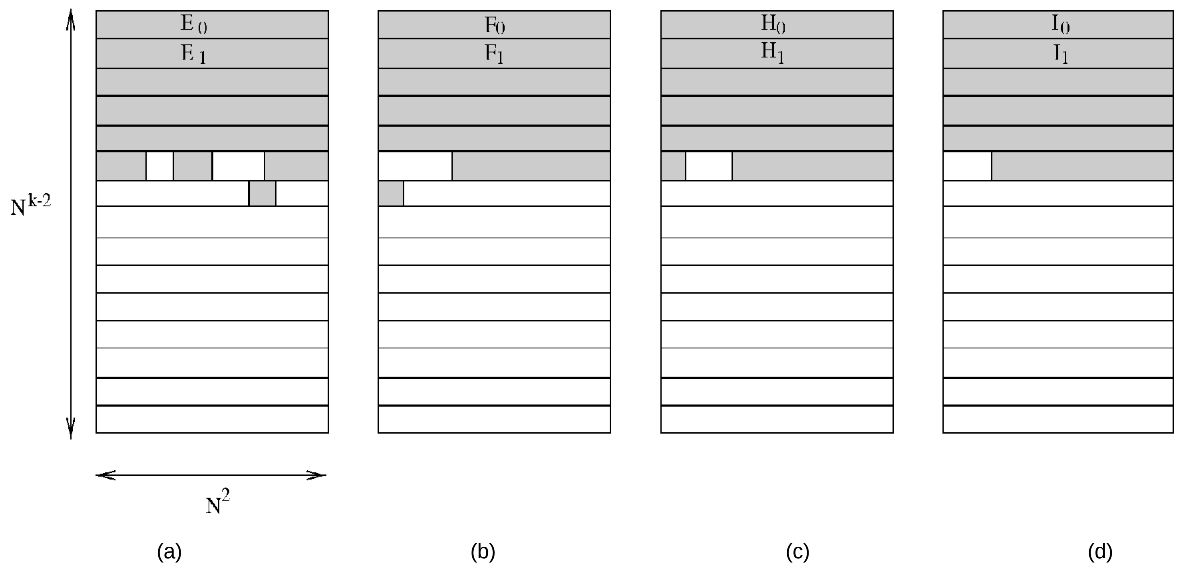


Fig. 11. Cleaning of the dirty area.

PROOF. We know that the dirty area of the sequence D, obtained in Step 3, has, at most, length N^2. If we divide the sequence D into consecutive subsequences, E_z, of N^2 keys each, the dirty area can either fit in exactly one of these subsequences or be distributed between two adjacent subsequences.

If the dirty area fits in one subsequence E_z, then after the initial sorting and the odd-even transpositions, the sequences H_z contain exactly the same keys as the sequences E_z, for $z = 0, \cdots, \frac{m}{N} - 1$. Then, the last sorting in each sequence H_z and the final concatenation of the I_z sequences yield a sorted sequence S.

However, if the dirty area is distributed between two adjacent subsequences, E_z and E_{z+1}, we have two subsequences containing both zeros and ones. Fig. 11a presents an example of this initial situation. After the first sorting, the zeros are located at one side of F_z and at the other side of F_{z+1} (see Fig. 11b).

One of the two odd-even transpositions will not affect this distribution, while the other is going to move zeros from the second sequence to the first and ones from the first to the second. After these two steps, H_z is filled with zeros or H_{z+1} is filled with ones (see Fig. 11c). Therefore, only one sequence contains zeros and ones combined. The last step of sorting will sort this sequence. Then, the entire sequence S will be sorted (see Fig. 11d). □

3.2 The Need for a Special Algorithm for N^2 Keys

The reader can observe that, at the end of Step 3, the dirty area will still have length N^2, even when we are merging N sequences of length N each. Thus, we do not make any progress when we apply the multiway-merge process to this case recursively. This difficulty can be overcome in a number of ways to keep the running time low, depending on the application area of the basic idea of the merge algorithm. For example, if we are interested in building a sorting network, we can implement subnetworks based on recursively updating N to a smaller value M and then merging M sequences of length $M^{k-1} = N$ for some $k > 2$, and repeat this recursion until a single sequence is obtained.

In this paper, our focus is developing sorting algorithms for product networks with r dimensions. Here, we assume the availability of a special sorting algorithm designed for the two-dimensional version of the product network under consideration. We use this assumed algorithm to sort N^2 keys when merging is no longer viable in the recursion. In subsequent sections, we discuss several methods to obtain such algorithms to sort N^2 keys as we consider more specific product networks. The efficiency of that special algorithm has an important effect on the overall complexity of the final sorting algorithm by the proposed approach. For all the product graphs considered in this paper, it will turn out that the resulting running time is either asymptotically optimal, or close to optimal, when the number of dimensions is bounded.

3.3 Sorting Algorithm

Using the above algorithm, and an algorithm to sort sequences of length N^2, it is easy to obtain a sorting algorithm to sort a sequence of length N^r, for $r \geq 2$.

First, divide the sequence into subsequences of length N^2 and sort each subsequence independently. Then, apply the following process until only one sequence remains:

1) Group all the sorted sequences obtained into sets of N sequences each as in Fig. 2. (If we are sorting N^r keys, then, initially, there will be N^{r-3} groups, each containing N sorted sequences of length N^2.)
2) Merge the sequences in each group into a single sorted sequence using the algorithm shown in the previous section. If now there is only one sorted sequence, then terminate. Otherwise, go to Step 1.

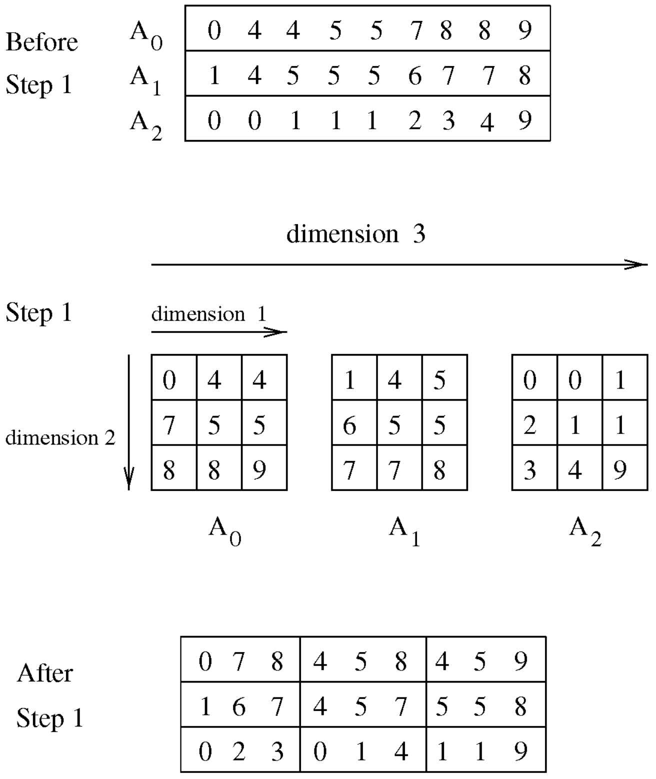


Fig. 12. Before Step 1, each A_u is stored in the $[u]PG^k_{r-1}$ subgraph (in this case in the corresponding arrays as shown) in snake order. Step 1 does not require any data movement; we simply change our view of how the data is stored. In this example, reading the data stored in the jth column of array A_i from top to bottom yields the $B_{i,j}$ sequence.

4 Implementation on Homogeneous Product Networks

Here, we mainly focus on the implementation of the multiway-merge algorithm on a k-dimensional product network PG_k in detail, where PG_k could be a subgraph of PG_r, performing some step of recursion in the overall sorting algorithm above. The initial scenario is N sorted sequences, of N^{k-1} keys each, stored on the N subgraphs $[u]PG^k_{k-1}$ of PG_k in snake order. Before the sorting algorithm starts, each processor holds one of the keys to be sorted. During the sorting algorithm, each processor needs enough memory to hold at most two values being compared. Throughout the discussions, the steps of implementation are illustrated by a three-dimensional product of some graph G of $N = 3$ nodes. The interconnection pattern of G is irrelevant for this discussion.

Step 1. This step does not need any computation or routing. Recall from Section 2 that each of the subgraphs $[u, v]PG^{k,1}_{k-2}$ of $[u]PG^k_{k-1}$ contains a subsequence of keys sorted in snake order, and that the positions of the keys in that subsequence, with respect to the total sorted sequence, are v, $2N - v - 1$, $2N + v$, $4N - v - 1$, $4N + v$, etc. Therefore, the sequence $B_{u,v}$ is already stored on the subgraph $[u, v]PG^{k,1}_{k-2}$, sorted in snake order.

This is illustrated in Fig. 12, where the three sequences to be merged are available in snake order on the three subgraphs formed by removing the edges of dimension-three. The subgraph $[0]PG^3_2$ (leftmost subgraph in Fig. 12) contains A_0, the subgraph $[1]PG^3_2$ (center subgraph) contains A_1, and the subgraph $[2]PG^3_2$ (rightmost subgraph) contains A_2. In this example, each $B_{u,v}$ contains only N keys, which fit in just one G subgraph. In general, $B_{u,v}$ will be available in snake order on $[u, v]PG^{k,1}_{k-2}$. In this example, they are at $[u, v]PG^{3,1}_1$, which really correspond to G-subgraphs at dimension two (i.e., columns of Fig. 12).

Step 2. This step is implemented by merging together the sequences on subgraphs $[u, v]PG^{k,1}_{k-2}$ with the same u value into one sequence on $[v]PG^1_{k-1}$. If $k - 1 = 2$, the merging is done by directly sorting with an algorithm for PG_2. If $k - 1 > 2$, this step is done by a recursive call to the multiway-merge algorithm, where each subgraph $[v]PG^1_{k-1}$ merges the sorted sequences stored on their $[u, v]PG^{k,1}_{k-2}$ subgraphs.

We illustrate this step in Fig. 13. For clarity, we first show the initial situation in Fig. 13a. This is same as the situation in Fig. 12, but dimensions one and three are exchanged to show the subsequences that will be merged together more explicitly. The $B_{u,v}$ sequences to be merged together are the columns of Fig. 13a. The result of merging is shown in Fig. 13b. Each C_v is sorted in snake order and is found in the subgraph $[v]PG^1_2$.

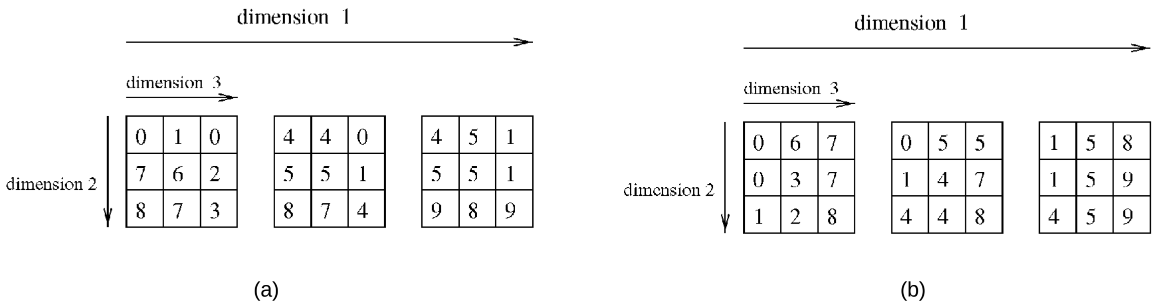


Fig. 13. Step 2 of the multiway-merge algorithm.

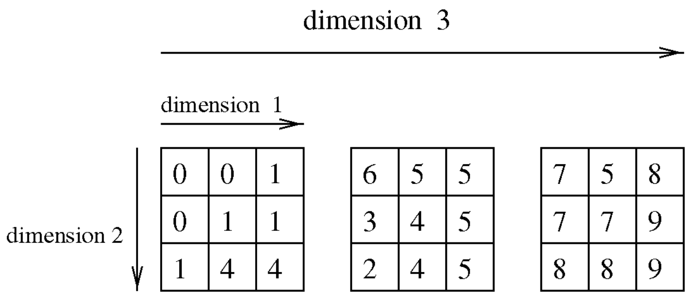


Fig. 14. Step 3 of the multiway-merge algorithm.

Step 3. This step is directly done by reintroducing the dimension-one connections of PG_k and reading the keys in snake order for the PG_k graph. No movement of data is involved in this step. We explicitly show the resulting sequence for our example in Fig. 14 by switching dimensions one and three in Fig. 13b. Recall from Figs. 10 and 11 that the keys now appear to be close to a fully sorted order. In fact, we know from Lemma 1 that, in the case of sorting zeros and ones, we are left with a small dirty area. This implies that every key is within a distance of N^2 from its final position.

Step 4. This last step cleans the potential dirty area. Recall from the last paragraph of Section 2 that the two-dimensional subgraphs of PG_k can be ordered in snake order by their group labels in the form of group sequences $[*,*]Q^{1,2}_{k-2}$. In this step, we first independently sort the keys in each PG_2 subgraph at dimensions {1, 2}, where the sorted order alternates for "consecutive" subgraphs. This sorting is done in snake order by using an algorithm which we assumed available. The result of this step is illustrated in Fig. 15a.

We now perform two steps of odd-even transposition between the two dimensional subgraphs. In the first step, the keys on the nodes of the "odd" PG_2 subgraphs are compared with the keys on the corresponding nodes of their "predecessor" subgraphs. The keys are exchanged if the key in the predecessor subgraph is larger. Fig. 15b shows the result of this first step in our example. The keys 3 and 2 in nodes (1, 2, 1) and (1, 2, 2) have been exchanged with two keys both with value four in nodes (0, 2, 1) and (0, 2, 2).

In the second step of odd-even transposition, the keys on the nodes of the "even" PG_2 subgraphs are compared (and possibly exchanged) with those of their predecessor subgraphs. Fig. 15c shows the result of this second step. In this figure, the key 5 in node (2, 0, 0) has been exchanged with the key 6 in node (1, 0, 0).

Finally, a sorting within each of the two-dimensional subgraphs ends the merge process (Fig. 15d).

One point which needs to be examined in more detail here is that, depending on the factor graph G, the nodes holding the two keys that need to be compared and possibly exchanged with each other may or may not be adjacent in PG_k. If G has a Hamiltonian path, then the nodes of G can be labeled in the order they appear on the Hamiltonian path to define the sorted order for G. Then, the two steps of odd-even transposition are easy to implement, since they involve communication between adjacent nodes.

If, however, G is not Hamiltonian (e.g., a complete binary tree), the two nodes whose keys need to be compared may not be adjacent, but they will always be in a common G subgraph. In this case, permutation routing within G may be used to perform the compare-exchange step as follows: First, two nodes that need to compare their keys send their keys to each other. Then, depending on the result of comparison, each node can either keep its original key, if the keys were already in correct order, or they drop the original key and keep the new key if they were out of order. To cover the most general case in the computation of running time below, we will assume that G is not Hamiltonian and, thus, we will implement these compare-exchange steps by using permutation routing algorithms. We will see that whether or not G is Hamiltonian only effects the constant terms in the running time complexity function.

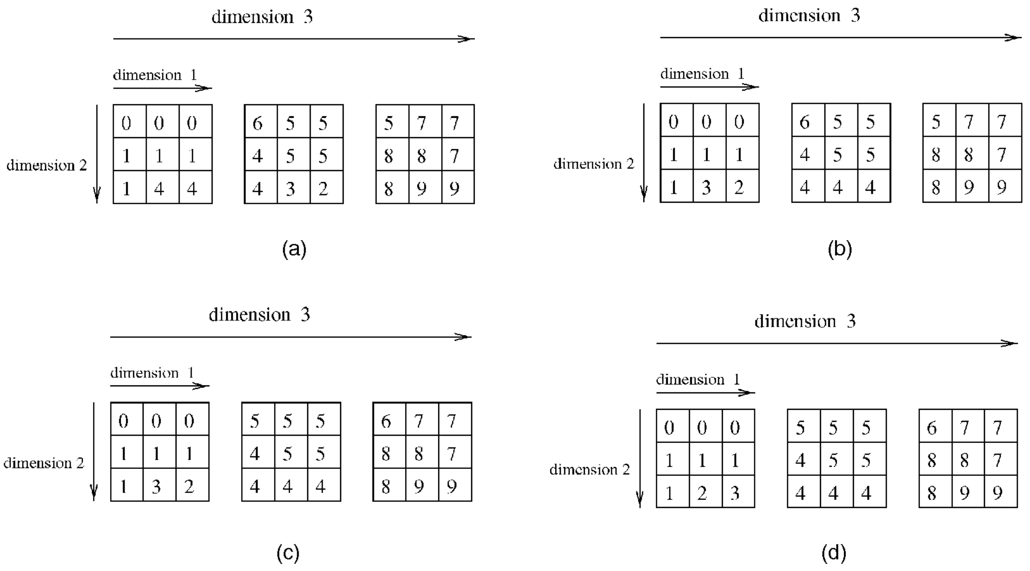


Fig. 15. Step 4 of the multiway-merge algorithm.

4.1 Analysis of Time Complexity

To analyze the time taken by the sorting algorithm, we will initially study the time taken by the merge process on a k-dimensional network. This time will be denoted as $M_k(N)$. Also, let $S_2(N)$ denote the time required for sorting on PG_2 and $R(N)$ denote the time required for a permutation routing on G.

LEMMA 3. *Merging N sorted sequences of N^{k-1} keys on PG_k takes $M_k(N) = 2(k - 2)(S_2(N) + R(N)) + S_2(N)$ time steps.*

PROOF. Step 1 does not take any computation time. Step 2 is a recursive call to the merge procedure for $k - 1$ dimensions, and hence will take $M_{k-1}(N)$ time. Step 3 does not take any computation time. Finally, Step 4 takes the time of one sorting on PG_2, two permutation routings on G (for the steps of odd-even transposition), and one more sorting on PG_2.

Therefore, the value of $M_k(N)$ can be recursively expressed as:

$$M_k(N) = M_{k-1}(N) + 2\big(S_2(N) + R(N)\big),$$

with initial condition

$$M_2(N) = S_2(N)$$

that yields

$$M_k(N) = 2(k-2)(S_2(N) + R(N)) + S_2(N).$$

□

We can now derive the value of $S_r(N)$.

THEOREM 1. *For any factor graph G, the time complexity of sorting N^r keys on PG_r is $S_r(N) = (r - 1)^2 S_2(N) + (r - 1)(r - 2) R(N) = O(r^2 S_2(N))$.*

PROOF. By the algorithm of Section 3.2, the time taken to sort N^r keys on PG_r is the time taken to sort in a two-dimensional subgraph and then merge blocks of N sorted sequences into increasing number of dimensions. The expression of this time is as follows:

$$\begin{aligned} S_r(N) &= S_2(N) + M_3(N) + M_4(N) + \cdots + M_{r-1}(N) + M_r(N) \\ &= (r-1)S_2(N) + 2\big(S_2(N) + R(N)\big)\sum_{i=3}^{r}(i-2) \\ &= (r-1)^2 S_2(N) + (r-1)(r-2)R(N). \end{aligned}$$

Since $S_2(N)$ is never smaller than $R(N)$, the time obtained is $S_r(N) < 2(r - 1)^2 S_2(N) = O(r^2 S_2(N))$. □

The following corollary presents the asymptotic complexity of the algorithm and one of the main results of this paper.

COROLLARY. *If G is a connected graph, the time complexity of sorting N^r keys on PG_r is at most* $18(r - 1)^2 N + o(r^2 N) = O(r^2 N)$.

PROOF. To prove the claim, we first compute the complexity of sorting by our algorithm on the r-dimensional torus. Then, we refer to a result in [8] that showed that, if G is a connected graph, PG_r can emulate any computation on the N^r-node r-dimensional torus by embedding the torus into PG_r with dilation three and congestion two. Since this embedding has constant dilation and congestion, the emulation has constant slowdown. (In fact, the slowdown is no more than six, and needed only when G does not have a Hamiltonian cycle). Finally, we use these slowdown values to compute the exact running time for PG_r

Now, we compute the complexity of sorting on the r-dimensional torus. We basically need a sorting algorithm from the literature that sorts N^2 keys in two-dimensional torus in snake order. We also need an algorithm for permutation routing on the N-node cycle. For example, we can use the sorting algorithm proposed by Kunde [16], which has complexity $2.5N + o(N)$. It is also known that any permutation routing can be done on the N-node cycle in no more than $N/2$ steps. Hence, we can sort on the N^r-node r-dimensional torus in at most $3(r-1)^2 N + o(r^2 N)$ steps.

Since the emulation of this algorithm by PG_r requires a slowdown factor of ,at most, six, any arbitrary N^r-node r-dimensional product network can sort with complexity $18(r-1)^2 N + o(r^2 N) = O(r^2 N)$. □

5 Application to Specific Networks

In this section, we obtain the time complexity of sorting using the multiway-merge sorting algorithm presented for several product networks in the literature. To do so, we obtain upper bounds for the values of $S_2(N)$ and $R(N)$ for each network. Using these values in Theorem 1 will yield the desired running time.

5.1 Grid

Schnorr and Shamir [30] have shown that it is possible to sort N^2 keys on an N^2-node two-dimensional grid in $3N + o(N)$ time steps. It is also trivial to show that the time to perform a permutation on the N-node linear array is, at most, $R(N) = N - 1$. These values of $S_2(N)$ and $R(N)$ imply that our algorithm will take, at most, $4(r-1)^2 N + o(r^2 N) = O(r^2 N)$ time steps to sort N^r keys on an N^r-node r-dimensional grid. If the number of dimensions, r, is bounded, this expression simplifies to $O(N)$.

This algorithm is asymptotically optimal when r is fixed, since the diameter of the grid with bounded number of dimensions is $O(N)$, and a value may need to travel as far as the diameter of the network. If r is not bounded, then the diameter of the N^r-node grid is $r(N-1)$, which means that the running time of our algorithm is off the optimal value by at most a factor of r.

5.2 Mesh-Connected Trees (MCT)

This network was introduced in [9] and extensively studied in [8]. It is obtained as the product of complete binary trees. Due to Corollary 1, we can sort on the N^r-node r-dimensional mesh-connected trees in $O(r^2 N)$ time steps. If r is bounded, we again have $O(N)$ as the running time.

This running time is asymptotically optimal when r is fixed, because the bisection width of the N^r-node r-dimensional MCT is $O(r^2 N)$, as shown in [8], and, in the worst case, we may need to move $\Omega(N^r)$ values across the bisection of the network. When r is not fixed, the algorithm is off the bisection-based lower bound by a factor of r^2. The diameter-based lower bound used above for grids does not help to tighten this lower bound any further, because the diameter of the MCT is logarithmic in the number of nodes [8]. It appears interesting to investigate if it is possible to sort with lower running time than $O(r^2 N)$ when r is not bounded. If such an algorithm exists, it must use a completely different approach than ours, because the value of $S_2(N)$ in Theorem 1 cannot be less than $O(N)$ due to the $O(N)$ bisection width of the two-dimensional MCT network.

5.3 Hypercube

The hypercube has fixed $N = 2$. It is not hard to sort in snake order on the two-dimensional hypercube in three steps. A permutation routing on the one-dimensional hypercube takes only one step. Therefore, the time to sort on the hypercube with our algorithm is $3(r-1)^2 + (r-1)(r-2) = O(r^2)$. This running time is same as the running time of the well-known Batcher odd-even merge algorithm for hypercubes. In fact, Batcher algorithm is a special case of our algorithm.

5.4 Petersen Cube

The Petersen cube is the r-dimensional product of the Petersen graph, shown in Fig. 16. The Petersen graph contains 10 nodes and consists of an outer five-cycle and an inner five-cycle, connected by five spokes. Product graphs obtained from the Petersen graph are studied in [26]. Like the hypercube, the product of Petersen graphs has fixed N, and therefore, the only way the graph grows is by increasing the number of dimensions. Since the Petersen graph is Hamiltonian, its two-dimensional product contains the 10×10 two-dimensional grid as a subgraph. Thus, we can use any grid algorithm for sorting 100 keys on the two-dimensional product of Petersen graphs in constant time. Consequently, the r-dimensional product of Petersen graphs can sort 10^r keys in $O(r^2)$ time. The constant involved is not small, but it is not going to be unreasonably large either. It may very well be possible to improve this constant by developing a special sorting algorithm for the two-dimensional product of Petersen graphs. This is, however, outside the scope of this paper.

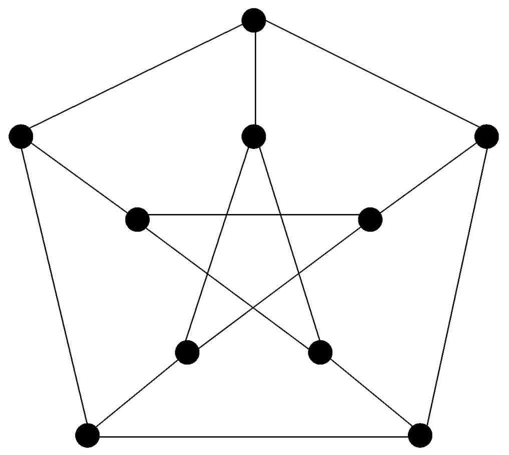


Fig. 16. Petersen graph.

5.5 Product of de Bruijn and Shuffle-Exchange Networks

To sort on their two-dimensional instances, we can use the embeddings of their factor networks presented in [9], which have small constant dilation and congestion. In particular, an N^2-node shuffle-exchange network can be embedded into the N^2-node two-dimensional product of shuffle-exchange networks with dilation four and congestion two. Also, an N^2-node de Bruijn network can be embedded into the N^r-node two-dimensional product of de Bruijn networks with dilation two and congestion two. Sorting N^2 keys on the N^2-node shuffle-exchange or de Bruijn networks can be done in $O(\log^2 n)$ time by using Batcher algorithm [31]. Thus, we can sort on the N^2-node two-dimensional product of shuffle-exchange or de Bruijn network by emulation of the N^2-node shuffle-exchange or de Bruijn network in $S_2(N) = O(\log^2 N^2) = O(\log^2 N)$ time steps. Using this in Theorem 1, our algorithm will take $O(r^2 \log^2 N)$ time steps to sort N^r keys. Again, if r is bounded the expression simplifies to $O(\log^2 N)$. If r is not bounded, the running time of our algorithm is asymptotically the same as the running time of sorting N^r keys on the N^r-node de Bruijn, or shuffle-exchange graphs by Batcher algorithm. Here again, we come across an interesting open problem, to see if it is possible to sort on products of these networks in asymptotically less time for unbounded number of dimensions.

6 Conclusions

In this paper, we have presented a unified approach to sorting on homogeneous product networks. To do so, we present an algorithm based on a generalization of the odd-even merge sorting algorithm [2]. We obtain $O(r^2 N)$ as an upper bound on the complexity of sorting on any product network of r dimensions and N^r nodes.

The time taken by the sorting algorithm on the grid and the mesh-connected trees with bounded number of dimensions is $O(N)$, which is optimal. On the hypercube, the algorithm takes $O(r^2)$ time steps, reaching the asymptotic complexity of the odd-even merge sorting algorithm on the hypercube.

On other product networks, our algorithm has the same running time as those of other comparable networks. For instance, on the product of de Bruijn or shuffle-exchange graphs, the running time is $O(r^2 \log^2 N)$. This is asymptotically the same as the running time of Batcher algorithm on the N^r-node shuffle-exchange or de Bruijn graphs. These results show that the generality of the proposed algorithm does not come at any additional expense of the running time in comparison to sorting algorithms specifically developed for these networks.

From a theoretical point of view, it will be interesting to investigate if there are better algorithms for product networks when r is not bounded. Several interesting alternatives appear to be feasible, although we have not had the time to investigate them. For instance, we could try to generalize the hypercube randomized algorithms for product networks.

Acknowledgments

Dr. Antonio Fernández's research was funded by a Fulbright Scholarship. Dr. Kemal Efe's research was funded by Louisiana Board of Regents, contract number LEQSF-(1995-97)-RD-A-33. The authors are thankful to Nancy Eleser for providing the running example in Figs. 12, 13, 14, and 15.

References

[1] A. Aggarwal and M.-D.A. Huang, "Network Complexity of Sorting and Graph Problems and Simulating CRCW PRAMS by Interconnection Networks," *Proc. Third Aegean Workshop Computing, AWOC '88: VLSI Algorithms and Architectures,* J.H. Reif, ed., *Lecture Notes in Computer Science*, vol. 319, pp. 339–350, Corfu, Greece. Springer-Verlag, July 1988.

[2] K. Batcher, "Sorting Networks and their Applications," *Proc. AFIPS Spring Joint Computing Conf.*, vol. 32, pp. 307–314, 1968.

[3] K.E. Batcher, "On Bitonic Sorting Networks," *Proc. 1990 Int'l Conf. Parallel Processing,* vol. I, pp. 376-379, 1990.

[4] M. Baumslag and F. Annexstein, "A Unified Framework for Off-Line Permutation Routing in Parallel Networks," *Math. Systems Theory*, vol. 24, no. 4, pp. 233–251, 1991.

[5] G.E. Blelloch, C.E. Leiserson, B.M. Maggs, C.G. Plaxton, S. Smith, and M. Zagha, "A Comparison of Sorting Algorithms for the Connection Machine CM-2," *Proc. Third Ann. ACM Symp. Parallel Algorithms and Architectures*, pp. 3–16, July 1991.

[6] R. Cypher and C.G. Plaxton, "Deterministic Sorting in Nearly Logarithmic Time on the Hypercube and Related Computers," *J. Computer and System Sciences*, vol. 47, pp. 501–548, Dec. 1993.

[7] R.L.S. Drysdale III and F.H. Young, "Improved Divide/Sort/Merge Sorting Network," *SIAM J. Computing,* vol. 4, pp. 264–270, Sept. 1975.

[8] K. Efe and A. Fernández, "Mesh-Connected Trees: A Bridge Between Grids and Meshes of Trees," *IEEE Trans. Parallel and Distributed Systems,* vol. 7, no. 12, pp. 1,281-1,291, Dec. 1996.

[9] K. Efe and A. Fernández, "Products of Networks with Logarithmic Diameter and Fixed Degree," *IEEE Trans. Parallel and Distributed Systems*, vol. 6, pp. 963–975, Sept. 1995.

[10] A. Fernández and K. Efe, "Efficient VLSI Layouts for Homogeneous Product Networks," *IEEE Trans. Computers,* vol. 46, no. 10, pp. 1,070-1,082, Oct. 1997.

[11] A. Fernández, K. Efe, A.L. Broadwater, M.A. Lorenzo, and D. Calzada, "A Unified Approach to Algorithm Development for Product Networks," *Math. Modelling and Scientific Computing,* to appear, vol. 8, 1997.

[12] T. El-Ghazawi and A. Youssef, "A General Framework for Developing Adaptive Fault-Tolerant Routing Algorithms," *IEEE Trans. Reliability*, vol. 42, pp. 250–258, June 1993.

[13] A. Fernández, "Homogeneous Product Networks for Processor Interconnection," PhD thesis, Univ. of Southwestern Louisiana, Lafayette, Oct. 1994.

[14] M.W. Green, "Some Improvements in Non-Adaptative Sorting Algorithms," *Proc. Sixth Princeton Conf. Information Sciences and Systems*, pp. 387–391, 1972.

[15] D. Knuth, *Searching and Sorting, The Art of Computer Programming,* vol. 3. Reading, Mass.: Addison-Wesley, 1973.

[16] M. Kunde, "Optimal Sorting on Multi-Dimensionally Mesh-Connected Computers," *Proc. Fourth Ann. Symp. Theoretical Aspects of Computer Science,* F.J. Brandenburg, G. Vidal-Naquet, and M. Wirsing, eds., *Lecture Notes in Computer Science*, vol. 247, pp. 408–419. Springer-Verlag, 1987.

[17] D.-L. Lee and K.E. Batcher, "A Multiway Merge Sorting Network," *IEEE Trans. Parallel and Distributed Systems,* vol. 6, no. 2, pp. 211–215, Feb. 1995.

[18] D.-L. Lee and K.E. Batcher, "On Sorting Multiple Bitonic Sequences," *Proc. 1994 Int'l Conf. Parallel Processing,* vol. I, pp. 121-125, Aug. 1994.

[19] F.T. Leighton, *Introduction to Parallel Algorithms and Architectures: Arrays, Trees, and Hypercubes*. San Mateo, Calif.: Morgan Kaufmann, 1992.

[20] T. Leighton, "Tight Bounds on the Complexity of Parallel Sorting," *IEEE Trans. Computers,* vol. 34, no. 4, pp. 344–354, Apr. 1985.

[21] K.J. Liszka and K.E. Batcher, "A Modulo Merge Sorting Network," *Proc. Fourth Symp. Frontiers of Massively Parallel Computation*, McLean, Va., pp. 164–169, Oct. 1992.

[22] K.J. Liszka and K.E. Batcher, "A Generalized Bitonic Sorting Network," *Proc. 1993 Int'l Conf. Parallel Processing,* vol. I, pp. 105–108, 1993.

[23] D. Nassimi and S. Sahni, "Bitonic Sort on a Mesh-Connected Parallel Computer," *IEEE Trans. Computers,* vol. 27, no. 1, pp. 2–7, Jan. 1979.

[24] T. Nakatani, S.-T. Huang, B.W. Arden, and S.K. Tripathi, "K-Way Bitonic Sort," *IEEE Trans. Computers,* vol. 38, no. 2, pp. 283–288, Feb. 1989.

[25] D. Nath, S.N. Maheshwari, and P.C.P. Bhatt, "Efficient VLSI Networks for Parallel Processing Based on Orthogonal Trees," *IEEE Trans. Computers*, vol. 32, no. 6, pp. 569–581, June 1983.

[26] S.R. Öhring and S.K. Das, "The Folded Petersen Cube Network: New Competitors for the Hypercubes," *IEEE Trans. Parallel and Distributed Systems*, vol. 7, no. 2, pp. 151-168, Feb. 1996.

[27] B. Parker and I. Parberry, "Constructing Sorting Networks from *k*-Sorters," *Information Processing Letters*, vol. 33, pp. 157–162, Nov. 1989.

[28] F. Preparata and J. Vuillemin, "The Cube-Connected Cycles: A Versatile Network for Parallel Computation," *Comm. ACM*, vol. 24, pp. 300–309, May 1981.

[29] A.L. Rosenberg, "Product-Shuffle Networks: Toward Reconciling Shuffles and Butterflies," *Discrete Applied Math.*, vols. 37/38, pp. 465–488, July 1992.

[30] C.P. Schnorr and A. Shamir, "An Optimal Sorting Algorithm for Mesh Connected Computers," *Proc. 18th Ann. ACM Symp. Theory of Computing*, pp. 255–263, Berkeley, Calif., May 1986.

[31] H. Stone, "Parallel Processing with the Perfect Shuffle," *IEEE Trans. Computers*, vol. 20, no. 2, pp. 153–161, Feb. 1971.

[32] C.D. Thompson and H.T. Kung, "Sorting on a Mesh-Connected Parallel Computer," *Comm. ACM*, vol. 20, pp. 263–271, Apr. 1977.

[33] S.S. Tseng and R.C.T. Lee, "A Parallel Sorting Scheme whose Basic Operation Sorts *n* Elements," *Int'l J. Computer and Information Sciences*, vol. 14, no. 6, pp. 455–467, 1985.

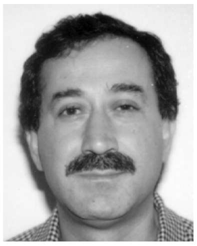
[34] D.C. van Voorhis, "An Economical Construction for Sorting Networks," *Proc. AFIPS Nat'l Computer Conf.*, vol. 43, pp. 921–927, 1974.

Antonio Fernández received the degree of Diplomado en Informática in March 1988 and the degree of Licenciado en Informática in July 1991 from the Universidad Politécnica de Madrid. He received the MS degree in computer science in the fall of 1992 and the PhD degree in computer science in the fall of 1994 from the University of Southwestern Louisiana, supported by a Fulbright Scholarship.

From April 1995 to March 1997, he was a visiting scientist at the Theory of Computation Group of the MIT Laboratory for Computer Science, supported by a grant from the Spanish Ministry of Education.

He is an associate professor in the Departamento de Arquitectura y Tecnología de Computadores at the Universidad Politécnica de Madrid, where he has served on the faculty since 1988.

His research interests include parallel architectures and algorithms, interconnection networks, distributed systems, and data communication.



Kemal Efe received the BSc degree in electronic engineering from Istanbul Technical University (1977), the MS degree in computer science from UCLA (1980), and the PhD degree in computer science from the University of Leeds (1985).

He is currently an associate professor of computer science at the Center for Advanced Computer Studies, University of Southwestern Louisiana. He is also a member of the founding team, and an associate director of the Laboratory for Internet Computing (LINC), recently formed by a $2.3M grant from the Department of Energy. Previously, he was on the faculty of the Computer Science Department, University of Missouri-Columbia. Dr. Efe has served on technical committees of many international conferences and given invited talks in the U.S., Europe, and Japan. In 1995, Dr. Efe received the "Certificate of Recognition" from NASA for his research contributions as a faculty research fellow. He is a member of the ACM and the IEEE.

Dr. Efe's research interests are in parallel and distributed computing, in which he made many significant contributions. His expertise includes parallel algorithms and architectures, interconnection networks, computer networks, distributed operating systems, distributed algorithms, performance evaluation, VLSI complexity models, and distributed multimedia information processing.