Generalized Algorithm for Parallel Sorting on Product Networks

Antonio Fernández,
MIT Laboratory for Computer Science
545 Technology Square
Cambridge, MA 02139

Nancy Eleser, and Kemal Efe
Center for Advanced Computer Studies
University of Southwestern Louisiana
Lafayette, LA 70504

Abstract

If G is a connected graph with N nodes, its r dimensional product contains N^r nodes. We present an algorithm which sorts N^r keys stored in the r-dimensional product of any graph G in $O(r^2 S(N))$ time where $S(N)$ depends on G. We show that for any graph G, $S(N)$ is bounded above by $O(N)$, establishing an upper bound of $O(r^2 N)$ for the time complexity of sorting N^r keys on any product network. When r is fixed, this leads to the asymptotic complexity $O(N)$ to sort N^r keys, which is optimal for several instances of product networks. There are graphs for which $S(N) = O(Log^2 N)$ which leads to the asymptotic running time of $O(Log^2 N)$.

Keywords: sorting, interconnection networks, product networks, algorithms, odd-even merge.

1 Introduction

In [1], 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 [10], the grid [11, 5], the cube-connected cycles [8], and the mesh of trees [6].

In this paper we generalize Batcher's algorithm to merge N 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. Among the main results of this paper, we show that the time complexity of sorting N^r keys for any r dimensional N^r-node product graph is bounded above as $O(r^2 N)$. We also illustrate special cases of product networks with running times of $O(r^2)$, or $O(N)$, or $O(Log^2 N)$.

2 Definitions and Notations

Let G be a N-node connected graph. We define its r-dimensional homogeneous product as follows.

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 product of G, denoted PG_r, is the graph whose vertex set is $V_{PG} = \{0, 1, \cdots, (N-1)\}^r$ and whose edge set is E_{PG}, 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 $(x_i, y_i) \in E_G$.*

Note that we can split the r-dimensional product network into N copies of $r-1$ dimensional product networks by erasing all the edges at an arbitrary dimension. When we do so, the uth copy obtained is denoted as $[u]PG^i_{r-1}$ where i is the dimension being erased. For example, if $i = r$ (*i.e.* the highest dimension) then we obtain N copies of $[u]PG^r_{r-1}$, each isomorphic to PG_{r-1}. Similarly, we can split the PG_r into N^2 copies of smaller product networks, each isomorphic to PG_{r-2} by erasing all the edges in two of the dimensions. In this case we extend the notation as $[u,v]PG^{i,j}_{r-2}$, where i and j denote the dimensions of edges being erased, and the pair $[u,v]$ uniquely identifies a copy obtained. This notation can be extended to erasing multiple dimensions in the obvious way, with the order of terms in square brackets corresponding to the order in superscripts.

When we focus on a k-dimensional subgraph of the r-dimensional product network, where $k < r$, it sometimes gets too long to write all the dimensions being erased, and it may be easier to just specify the dimensions of edges not erased. In particular, we will have occasion to refer to subgraphs obtained by erasing all dimensions but one, and thus the remaining subgraph will be isomorphic to the factor graph G. We will use $PG^{\{i\}}_1$ to denote such a subgraph, where i denotes the dimension of edges not erased. We also extend this notation similarly to above case, and use $PG^{\{i,j\}}_2$ to denote a two-dimensional subgraph with i and j indicating the dimension of edges not erased.

For an arbitrary factor graph G, vertex labels $0 \cdots N-1$ can define the ascending order of data when sorted. However we need to define an order for the nodes of PG_r, which will determine the final location of the sorted keys. The order defined is known as "snake" order.

Definition 2 *(Snake Order) for the r-dimensional product graph:*

1. *If $r = 1$, snake order corresponds to the order defined for G.*

2. *If $r > 1$, suppose that snake order has been already defined for PG_{r-1}. Then,*
   (a) *$[u]PG^r_{r-1}$ 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 $[u]PG^r_{r-1}$ precedes any value in $[v]PG^r_{r-1}$.*

The hamming weight of a vertex s is defined as $W(s) = \sum_{i=1}^{r} s_i$, where s_i is the symbol value at position i of the vertex label s. If the symbol at ith index position is the don't care symbol "*" then this symbol position is omitted when computing the Hamming weight. Depending on the parity of its Hamming weight, we say that a vertex is even or odd.

Similarly, we can define a Hamming weight for the $PG_1^{\{1\}}$ subgraphs of the product graph, by simply starting the summation from 2 when computing the Hamming weight. If the Hamming weight of a $PG_1^{\{1\}}$ is even, we say that it is an even subgraph. Otherwise it is an odd subgraph. We can also compute the Hamming weight for a $PG_2^{\{1,2\}}$ subgraph and say that it is even or odd.

Just like the definition of snake order for vertices, we can also define snake order for subgraphs $PG_1^{\{1\}}$ or $PG_2^{\{1,2\}}$ in the obvious way. This will be useful later.

Suppose that a sorted sequence is stored in some r-dimensional product network in snake order. The following lemma shows how to split the sequence into N subsequences such that the subsequence u contains every Nth term beginning with uth term.

Lemma 1 *Let S be a sorted sequence stored in some r-dimensional product network in snake order. By reversing the values at odd $PG_1^{\{1\}}$ subgraphs and then erasing the lowest-dimension edges from the product network, we obtain N copies of the product network with $r-1$ dimensions. The uth copy (where $u \in \{0 \cdots N-1\}$) will contain every Nth term of the original sequence beginning with the uth value.*

Proof is omited due to space limitations. The interested reader can refer to [4].

3 Sorting Algorithm

The heart of the proposed sorting algorithm is the multiway-merge operation. The multiway-merge algorithm combines N sorted sequences $A_i = (a_{i,0}, a_{i,1}, \ldots, a_{i,n-1})$, for $i = 0, \ldots, N-1$, into a single sorted sequence $J = (j_0, j_1, \ldots, j_{nN-1})$. For simplicity, we assume n to be a power of N, N^r, where $r > 1$.

To merge N sequences of N^r keys each, we initially 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 discussions, independent of the factor network used to build the product network.

3.1 Multiway-Merge Algorithm

Here we consider how to merge N sorted sequences, $A_i = (a_{i,0}, a_{i,1}, \ldots, a_{i,n-1})$, for $i = 0, \ldots, N-1$, into a single large sorted sequence.

The merge operation consists of the following steps:

1. Distribute the keys of each sorted sequence A_i among N sorted subsequences $B_{i,j}$, for $i = 0, \ldots, N-1$ and $j = 0, \ldots, N-1$. The subsequence $B_{i,j}$ will have the form $(a_{i,j}, a_{i,j+N}, a_{i,j+2N}, \ldots, a_{i,j+(n-N)})$, for $i = 0, \ldots, N-1$ and $j = 0, \ldots, N-1$. This is equivalent to picking every Nth element of A_i starting with the jth element and putting them in $B_{i,j}$. Note that each subsequence $B_{i,j}$ is sorted since we put the elements in the same order as they appeared in A_i.

2. Merge the N subsequences $B_{i,j}$ into a single sorted sequence C_j, for $j = 0, \ldots, N-1$. This is done with a recursive call to the multiway-merge process if the total number of keys in $B_{i,j}$ is at least N^2. If the number of keys in $B_{i,j}$ is N, a sorting algorithm for sequences of length N^2 is used to obtain a single sequence, because a recursive call to the merge process would not make much progress in this case (this point will be clear by the end of this subsection).

3. Interleave the sequences C_j into a single sequence $D = (d_0, d_1, \cdots, d_{N^{r+1}-1})$. The first N terms of the sequence D is obtained by reading the first element from each C_j, $j = 1 \cdots N$. The next set of N terms in D are obtained by reading the second value from each C_j, $j = 1 \cdots N$, and so on.

   We prove below that D is now "almost" sorted; the potential dirty area (window of keys not sorted) has length no larger than N^2.

4. Clean the dirty area. To do so we start by dividing the sequence D into N^{r-1} subsequences of N^2 consecutive keys each. That is, the first N^2 terms of D are labeled as E_1, the next N^2 terms are labeled as E_2, and so on.

   We then independently sort the E_i subsequences in alternate orders by using the algorithm which we assumed available for sorting N^2 keys. E_i is transformed into a sequence F_i where F_i contains the keys of E_i sorted in non-decreasing order if i is even or in non-increasing order if i is odd, for $i = 0, \ldots, N^{r-1} - 1$.

   Now, we apply two steps of odd-even transposition between the sequences F_i, for $i = 0, \ldots, N^{r-1} - 1$. In the first step of odd-even transposition, each pair of sequences F_i and F_{i+1}, for i even, are compared element by element. Two sequences G_i and G_{i+1} are formed where $g_{i,k} = min(f_{i,k}, f_{i+1,k})$ and $g_{i+1,k} = max(f_{i,k}, f_{i+1,k})$. In the second step of the odd-even transposition, G_i and G_{i+1} for i odd are compared in a similar manner to form the sequences H_i and H_{i+1}.

Finally, we sort each sequence H_i in non-decreasing order, generating sequences I_i, for $i = 0, \ldots, N^{r-1} - 1$. The final sorted sequence J is the concatenation of the sequences I_i.

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

Lemma 2 *When sorting an input sequence of zeroes 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 zeroes and ones. Let z_i be the number of zeroes in sequence A_i, for $i = 0, \ldots, N-1$. The rest of elements in A_i are ones. Step 1 breaks each sequence A_i into N subsequences $B_{i,j}$, $j = 0, \ldots, N-1$. The number of zeroes in a subsequence $B_{i,j}$ is $\lfloor z_i/N \rfloor + q_{ij}$, where $q_{ij} = 1$ if $j \leq z_i \ mod \ N$ and $q_{ij} = 0$ otherwise. Observe that, for a given i, the sequences $B_{i,j}$ can differ from each other in their number of zeroes by at most one.

At the start of step 2, each column j is composed of the subsequences $B_{i,j}$ for $i = 0, \ldots, N-1$. At the end of step 2, all the zeroes are at the beginning of each sequence C_j. The number of zeroes in each sequence C_j is the sum of the number of zeroes in $B_{i,j}$ for fixed j and $i = 0, \cdots, N-1$. Thus, two sequences C_j can differ from each other by at most N zeroes. In step 4 we interleave the N sorted sequences into the sequence D by taking one key at a time from each sequence C_j. Since any two sequences C_j can differ in their number of zeroes 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 zeroes is at most N^2. ■

Lemma 3 *Step 4 cleans the dirty area.*

Proof: We know that the dirty area of the sequence D, obtained in step 4, has at most length N^2. If we divide the sequence D into consecutive subsequences, E_i, 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_i then, after the initial sorting and the odd-even transpositions, the sequences H_i contain exactly the same keys as the sequences E_i, for $j = 0, \ldots, N^{r-1}$. Then, the last sorting in each sequence H_i and the final concatenation yield a sorted sequence J.

However, if the dirty area is distributed between two adjacent subsequences, E_i and E_{i+1}, we have two subsequences containing both zeroes and ones. After the first sorting, the zeroes are located at one side of F_i and at the other side of F_{i+1}.

One of the two odd-even transposition steps will not affect this distribution, while the other step is going to move zeroes from the second sequence to the first and ones from the first to the second. After these two steps, H_i is filled with zeroes or H_{i+1} is filled with ones. Therefore, only one sequence contains zeroes and ones combined. The last step of sorting will sort this sequence. Then, the entire sequence J will be sorted. ■

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 much progress when we apply the multiway-merge process in this case. Here we assume the availability of a special sorting algorithm designed for the two-dimensional version of the product network. In subsequent sections we discuss several methods to obtain such algorithms as we consider more specific product networks.

3.2 Application of Merging Algorithm to Sort

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. (If we are sorting N^{r+1} keys, then initially there will be N^{r-2} groups, each containing N 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.

4 Implementation in Homogeneous Product Networks

Here we mainly focus on the implementation of the multiway-merge algorithm in PG_r in detail. The sorting algorithm trivially follows from the merge operation as described above. The initial scenario is N sorted sequences, of N^{r-1} keys each, stored in the N subgraphs $[u]PG^r_{r-1}$ 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.

Step 1: To explain how this step can be implemented, we refer to Lemma 1. By Lemma 1, if we reverse the order in "odd" $PG^{\{1\}}_1$ subgraphs, then we obtain the sequence $B_{u,v}$ in the subgraph $[u,v]PG^{r,1}_{r-2}$ sorted in snake order.

Reversing the order in a G subgraph can be performed by a permutation routing algorithm available for G.

Step 2 This step is implemented by merging together the sequences in subgraphs $[u,v]PG^{r,1}_{r-2}$ with the same u value into one sequence in $[v]PG^1_{r-1}$. If

$r-1=2$, the merging is done by directly sorting with an algorithm for PG_2. If $r-1>2$, this step is done by a recursive call to the multiway-merge algorithm.

Step 3 No movement of data is involved in this step, and we obtain a sequence sorted almost completely except for a small dirty area, as shown.

Step 4 This step cleans the dirty area. The $PG_2^{\{1,2\}}$ subgraphs are ordered by the snake order. In this step we independently sort the keys in $PG_2^{\{1,2\}}$ subgraphs, where the sorted order alternates in "consecutive" subgraphs. We now perform two steps of odd-even transposition between these subgraphs. In the first step, the nodes in the "odd" $PG_2^{\{1,2\}}$ subgraphs are compared with corresponding nodes in their "predecessor" subgraphs. The values are exchanged if the value in the predecessor subgraph is larger. In the second step of odd-even transposition, the values in the nodes of the "even" $PG_2^{\{1,2\}}$ subgraphs are compared (and possibly exchanged) with those of their predecessor subgraphs. A final sorting within each of the $PG_2^{\{1,2\}}$ subgraphs ends the merge process.

One point which needs to be examined in more detail here is that, depending on the factor graph G, the two elements that need to be compared and possibly exchanged with each other may or may not be adjacent in PG_r. 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 sort is easy to implement since it involves communication between adjacent nodes in PG_r. If however G is not Hamiltonian (e.g. a complete binary tree), the two elements that 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: two nodes that need to compare their values send their values to each other. Then, depending on the result of comparison, each node can either keep its original value if the values were already in correct order, or they drop the original value and keep the new value if they were out of order.

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 in a k-dimensional network. This time will be denoted as $M_k(N)$. Also let $S_2(N)$ denote the time required for sorting in PG_2 and $R(N)$ denote the time required for permutation routing in G.

Lemma 4 *Merging N sorted sequences of N^{k-1} keys in PG_k takes $M_k(N)=2(k-2)S_2(N)+3(k-2)R(N)+S_2(N)$ time steps.*

Proof: The time taken by step 1 of the merge process is just the time to reverse the order of the keys in $PG_1^{\{1\}}$-subgraphs. This process can be done with a permutation routing algorithm for G, that takes time $R(N)$. 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 in PG_2, two permutation routings in G (for the steps of odd-even transposition), and one more sorting in PG_2.

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

$$M_k(N) = M_{k-1}(N) + 2S_2(N) + 3R(N)$$

with initial condition

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

that yields

$$M_k(N) = 2(k-2)S_2(N) + 3(k-2)R(N) + S_2(N)$$

■

Theorem 1 *For any factor graph G, the time complexity of sorting N^r keys in PG_r is $O(r^2 S_2(N))$.*

Proof: By the algorithm of Section 3.2 the time taken to sort N^r keys in PG_r is the time taken to sort in a 2-dimensional subgraph and then merge blocks of N sorted sequences into increasing number of dimensions. The expression of this time is as follows:

$$S_r(N) = S_2(N)+M_3(N)+M_4(N)+\cdots+M_{r-1}(N)+M_r(N)$$

$$= (r-1)S_2(N) + (2S_2(N)+4R(N))\sum_{i=3}^{r}(i-2)$$

$$= (r-1)^2 S_2(N) + 1.5(r-1)(r-2)R(N).$$

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

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

Corollary 1 *If G is a connected graph, the time complexity of sorting N^r keys in PG_r is at most $O(r^2N)$.*

Proof: The basic observation is that, if G is a connected graph, it is always possible to obtain an algorithm for PG_2 with complexity $S_2(N)=O(N)$. To do so we simply emulate the 2-dimensional grid in PG_2 with constant dilation and congestion [2]. Then, the $O(N)$ algorithm presented by Schnorr and Shamir [9] for sorting N^2 keys on two-dimensional $N\times N$ grid can be emulated by PG_2 with complexity $O(N)$, leading to $S_2(N)=O(N)$. Hence, any arbitrary N^r-node r-dimensional product network can sort with complexity $O(r^2N)$. ■

5 Application to Specific Networks

In this section we obtain the time complexity of sorting for several product networks in the literature. To do so, we obtain upper bounds for the value of $S_2(N)$ for each network. Using this value in Theorem 1 will yield the desired running time.

Grid: Schnorr and Shamir [9] have shown that it is possible to sort N^2 keys in a N^2-node 2-dimensional grid in $O(N)$ time steps. This value of $S_2(N)$ implies that our algorithm will take $O(r^2N)$ time steps to sort N^r keys in a 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.

Mesh connected trees (MCT): This network was introduced in [3] and extensively studied in [2]. It is obtained as the product of complete binary trees. Due to Corollary 1 we can sort in the N^r-node r-dimensional mesh connected trees in $O(r^2N)$ time steps. If r is bounded, we again have $O(N)$ as the running time. This running time is optimal when r is fixed, because the bisection width of r-dimensional MCT is $O(N^{r-1})$ as shown in [2], and in the worst case we may need to move $O(N^r)$ values across the bisection of the network.

Hypercube: From the above analysis given for grids and the fact that the hypercube is a special case of grid with $N = 2$ fixed, it follows that the time to sort in the hypercube with our algorithm is $O(r^2)$. This running time is same as the running time of the well-known Batcher's algorithm for hypercubes. In fact, Batcher's algorithm is a special case of the proposed algorithm.

Petersen Cube: Petersen cube is the r-dimensional product of Petersen graphs. Product graphs obtained from the Petersen graph are studied in [7]. 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 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.

Product of de Bruijn and shuffle-exchange networks: To sort in their two-dimensional instances we can use the embeddings of their factor networks presented in [3] which have small constant dilation and congestion. In particular, N^2-node shuffle-exchange network can be embedded onto the two dimensional $N \times N$ product of shuffle-exchange networks with dilation 4 and congestion 2. Also N^2-node de Bruijn network can be embedded onto the two dimensional $N \times N$ product of de Bruijn networks with dilation 2 and congestion 2. Sorting $n = N^2$ keys in shuffle-exchange or de Bruijn networks requires in $O(\log^2 n)$ time by the Batcher's algorithm. Thus, we can sort on the two-dimensional product of shuffle-exchange or de Bruijn networks by emulation of N^2-node shuffle-exchange or de Bruijn networks 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 $O(\log^2 N)$.

References

[1] K. Batcher, "Sorting Networks and their Applications," in *Proceedings of the AFIPS Spring Joint Computing Conference*, vol. 32, pp. 307–314, 1968.

[2] K. Efe and A. Fernández, "Computational Properties of Mesh Connected Trees: Versatile Architecture for Parallel Computation," in *Proceedings of the 1994 International Conference on Parallel Processing*, vol. I, (St. Charles, IL), pp. 72–76, CRC Press Inc., Aug. 1994.

[3] K. Efe and A. Fernández, "Products of Networks with Logarithmic Diameter and Fixed Degree," *IEEE Transactions on Parallel and Distributed Systems*, 1994. Accepted for publication. Also available as Technical Report 93-8-1, CACS, University of SW. Louisiana, Lafayette, LA, Feb. 1993.

[4] A. Fernández, N. Eleser, and K. Efe, "Generalized Algorithm for Parallel Sorting on Product Networks," Tech. Rep. 95-1-1, CACS, University of SW. Louisiana, Lafayette, LA, Sept. 1995.

[5] D. Nassimi and S. Sahni, "Bitonic Sort on a Mesh-Connected Parallel Computer," *IEEE Transactions on Computers*, vol. C-27, pp. 2–7, Jan. 1979.

[6] D. Nath, S. N. Maheshwari, and P. C. P. Bhatt, "Efficient VLSI Networks for Parallel Processing Based on Orthogonal Trees," *IEEE Transactions on Computers*, vol. C-32, pp. 569–581, June 1983.

[7] S. R. Öhring and S. K. Das, "The Folded Petersen Cube Networks: New Competitors for the Hypercube," in *Proceedings of the 5th IEEE Symposium on Parallel and Distributed Computing*, pp. 582–589, Dec. 1993.

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

[9] C. P. Schnorr and A. Shamir, "An Optimal Sorting Algorithm for Mesh Connected Computers," in *Proceedings of the 18th Annual ACM Symposium on Theory of Computing*, (Berkeley, CA), pp. 255–263, May 1986.

[10] H. Stone, "Parallel Processing with the Perfect Shuffle," *IEEE Transactions on Computers*, vol. C-20, pp. 153–161, Feb. 1971.

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