(3)

Sorting Networks Using k–Comparators

Y. B. Chiang
Department of Mathematics and Applied Mathematics
University of Cape Town

A thesis submitted in fulfilment
of the requirements for
the degree of PhD in Mathematics

6 June 2001

Abstract

Parallelism in computing has become the norm in extracting faster performance from computer systems, but while the idea of applying parallelism to achieve greater speeds is appealing, there are practical difficulties that need to be overcome before this can be done effectively. Hence in this work we investigate the possibility of modulate parallel sorting using k–comparators ($k > 2$). We formally introduce a merge paradigm that standardizes the sorting algorithms, then we construct efficient sorting algorithms that have good packaging property.

After we construct various efficient merge sorting algorithms on the parallel interconnection networks, following the paradigm, we continue to solve one particular problem in parallel connecting networks: the rearrangeability of the double butterfly networks of size N. We are able to find elegant control algorithms that route any signal from the input terminals to the output terminals of the double butterfly networks, for $N = 8$, 16 respectively (we also extend this approach for $N = 32$ informally).

In essence, this work explores some fundamental aspects of parallel interconnecting networks, where we introduce new ideas and new approaches to solve problems of sorting on highly parallel interconnecting networks and the rearrangeability problem of the double butterfly networks.

Contents

Acknowledgements

First and foremost, I must thank my supervisor, Prof. R. I. Becker, for his exemplary guidance during the preparation of this dissertation. I am truly grateful for his time, support and invaluable advice. My deepest gratitude for all the contributions and invaluable guidance that he has given me, to assist me in finish my thesis.

I would like to thank the Foundation for Research and Development for the funding towards my study. I would like to thank Head of Department Prof. Hahn and Department of Mathematics and Applied Mathematics for employed me as a tutor and then a temporary lecturer, which trains me to be a better mathematician. I would like to thank Dr. R. J. Lindebaum for his time on word-proofing my rough draft and numerous positive comments towards this write-up. I would also like to thank Prof. Brink and Prof. Gilmour for the encouragement that they have gave me.

Finally, I would like to thank my parents. I am grateful for their support and encouragement.

Chapter 1

Preliminaries

Multistage interconnection networks have long been studied for use in telephone services and multiprocessor systems. Much of literature on interconnection networks from the 1960's deals with telephone switching, as in Benes [13] [14] [15]. Even as early as 1968, there were studies done on two-dimensional mesh-type interconnection networks - see for example Kautz, Levitt and Waksman [48].

As the technology advanced, it became commercially viable to manufacture more powerful and much faster processors using mass production. Manufacturers were able to connect vast number of processor chips together and place numerous processor components on a single chip. Since then the research work done on interconnection networks has shifted to an emphasis on the analysis of multiprocessor systems.

As systems with thousands of processors became commercially viable, it became necessary to address many new aspects of interconnection networks, such as performance evaluations, routing algorithms, communication schemes between processors and the robustness of an interconnection network.

The important theme throughout this work is that of parallel sorting. This is because the interconnection network plays a key role in implementing fast parallel sorting algorithms. Before we formally introduce the techniques and the associated terminology, we now briefly touch on the motivational aspect of each chapter in this work.

In the literature, oblivious sorting algorithms appear in two guises: there are sorting comparator networks of the type found in Knuth [49], and then there are a number of sorting techniques dealing with interconnection networks, such as those found in Leighton [58]. These latter algorithms are generally designed to be efficient on networks with a specific type of interconnection, such as two-dimensional planar interconnection networks of general-purpose processors. These algorithms are usually oblivious, and proceed by sorting subparts of the interconnection network in various ways. These subparts are taken to be either one-dimensional portions, such as rows or columns, or if recursion is applied, square

sub-blocks of the interconnection network. Sometimes operations are performed and these preserve the structure of the interconnection network, for example, transposes are taken and the columns or rows are shuffled. This format for the algorithms ensures efficiency and ease of control.

The sorting networks of the type found in Knuth [49] are networks constructed from comparators, which are dedicated processors that are designed to perform sorting operation only. These algorithms are generally recursive in nature, such as Batcher's networks. In the chapters to follow, we will show that it is advantageous to think of these networks as sorting on a rectangular grid. The algorithms also proceed by sorting subparts of the grid, for example, there is no requirement to restrict the piece sorted recursively to being a grid of the same shape. There is the freedom to choose unusual shapes for the subparts sorted. In Chapter 4, we will present four such sorting algorithms, which merge subparts that are not square sub-blocks of the grid.

The algorithms used for constructing sorting networks are generally more efficient than those used for sorting on interconnection networks. However, both systems of sorting algorithms often can be shown to use the same principles, particularly if both are viewed as sorting on a grid. The grid representation's set of interconnections will not be predefined, but can be given in a way making the sorting algorithm efficient. It will be seen that a number of sorting networks, such as the odd-even network and the bitonic network of Batcher [8], the balanced network of Dowd et al [34] and variety of the networks produced in Becker, Nassimi and Perl [9], are related to sorting on grids.

The sorting networks referred to above are analogous to grids with two columns of height $\frac{N}{2}$. Generally, the types of sorting network that can be modelled by the grid representation are ones based on recursive mergers. It will be seen that there is a direct generalization of this relationship to that between grids with q columns each of height $\frac{N}{q}$, and sorting networks made up of k–comparators (k is a multiple of q) which are q–way r–mergers (merging r sorted sequence). In this way we will be able to build up a class of sorting networks using k–comparators, based on recursive q–way r–mergers, which is analogous to the known sorting networks of this type using 2–comparators. A move towards solving this problem was made by Parker and Parberry [75], and Becker and Litman [10]. Note: In some of the publications, the term "q–way" implies there are q sorted vectors that need to be merged. However, we will be following the terminology used in Nakatani et al [64] and Liszka and Batcher [62] to name an algorithm, where the term "q–way" is used to describe the number of columns in the grid A (derived from the input vector V, see Figure 3.1). The term "r–merge" implies we are merging groups of r sorted subvectors in parallel.

In Chapter 3 and Chapter 4 of this thesis, we will investigate certain topics in parallel sorting networks. In particularly, we will be concerned with parallel merge sorts. The usual merge sorting algorithms in the literature use 2–comparators that merge two vectors. However, in these two chapters, we will introduce new methods that merge r vectors using k–comparators, where r and k can be greater than 2. We will also indicate, when appropriate, some of the sorting algorithms on planar interconnection networks that are related to the algorithms we construct.

There are four reasons for this study:

1 r–Mergers: There has been renewed interest in merging $r \geq 2$ vectors with recent papers by Wen [102] and by Hayashi, Nakano and Olariu [42], after the subject was originally discussed sequentially in Knuth [49] (see those papers for motivation). Those papers give optimal algorithms and the algorithms are fairly complex. It is desirable to have conceptually simple algorithms which bear roughly the relation of Batcher's networks to more complex merging of two vectors.

2 k–Comparator Networks ($k > 2$): This question was first proposed by Knuth [49] (question 44 on page 243) in the early 1970's, then Parker and Parberry [75] (and others like Ajtai, Komlos, and Szemeredi [3]) have asked whether sorting can be based on k–comparator networks, if such k–comparators could be very efficiently constructed. In this thesis, complexities of parallel sorting algorithms are measured in terms of the number of comparators used and the number of parallel operations performed (the depth of a network). Our aim is to construct efficient k–comparator networks for k reasonably small, when this is possible.

3. Packaging of VLSI circuits: The number of inputs that a VLSI chip can receive at any given time is still much smaller than the size of data sets that need to be sorted. This is true despite the drastic improvements in VLSI circuits. In order to sort large sets of data, we need to pack many chips together. This motivates the need to better understand the q–way r–merge sorting networks and it would be advantageous to package them efficiently using standard components. If one is to regard a 2–comparator as the standard component, the packaging may be complicated and inefficient. If we can design around k–comparators ($k > 2$), then the packaging may be more efficient.

4 To be able to view some of the two-dimensional sorting algorithms, which sort long narrow rectangular mesh-type interconnection networks, under one uniform framework.

In Chapter 3 and Chapter 4, we shall approach the above four aspects in a fairly unified way. This unification is achieved by viewing them as sorting on a grid. We will investigate the following problem: Can one find generalizations of the popular 2–comparator mergers (Batcher's odd-even and bitonic [8], balanced [34], and the networks of Becker, Nassimi and Perl [9]) to k–comparator networks which will also be r–mergers and have good packaging properties, i.e. will be good VLSI algorithms. In Chapter 3, we will generalize those two most used 2–comparator networks: Batcher's [8] odd-even merge network and bitonic merge network. Then in Chapter 4, we will generalize the more recent version of 2–comparator networks: the balanced network of Dowd et al [34] and the collection of networks of Becker, Nassimi and Perl [9].

The above 2–comparator networks provide a clear and easy sorting method using 2–comparators, and are therefore used even when there are more efficient procedures. Hence the study of the underlying

paradigm of those algorithms will assist in extending the paradigm to k–comparators with r–merging properties. In Nakatani et al [64] and Liszka and Batcher [62], the above two algorithms of Batcher are generalized into q–way 2–mergers, and we will generalize those results even further into q–way r–mergers. We will construct analogues of the above two algorithms (from Nakatani et al [64] and from Liszka and Batcher [62]) for the k–comparator case, and then we will compare their depth and the number of comparators used for merging and for sorting.

In Chapter 4, we will present four merge grid algorithms. With the exception of the bichain merge grid algorithm (with bitransformer) that uses five levels of k–comparators when implemented on a k–comparator network of size $k \times k^{\frac{1}{2}}$. The remaining three algorithms will each use four levels of k–comparators when implemented on a k–comparator network of the same size (i.e. $k \times k^{\frac{1}{2}}$). These four new algorithms can be described by the following paradigm: Each algorithm takes the input N–vector and sorts its m–chains, which are the columns of the corresponding grid. The set of multiple generalized chains is then sorted and finally the resulting N–vector is sorted by a transformer appropriate for the specified type of multiple generalized chains.

Note: the bichain merge algorithm with skewtransformer uses four levels of k–comparators, which remedies the extra level used in the bichain sorting algorithm with bitransformer (when implemented on a k–comparator network of size $k \times k^{\frac{1}{2}}$).

In Chapter 5 of this thesis, we deal with the permutation properties of the double butterfly network. One of the fundamental problems in the interconnection network theory is to design rearrangeable networks with as few switches as possible. Characterizations and properties of various permutation networks have long been studied, such as Waksman [101], Opferman and Tsao-Wu [70] [71], Ramanujam [83], Yalamanchill and Aggaewal [106], Szymanski and Hamacher [94], and the two-dimensional permutation networks in the work of Oruc's [72]. The need to study permutation properties of a network stems from the communication requirements of parallel algorithms.

At the beginning of this chapter, we indicated that the pervasive theme throughout this thesis revolves around parallel sorting, which deals with comparison networks. Although the aspect of permutation networks deals with switching networks, this does not change what we have stated, because our approach incorporates some known properties of the Omega network and the bitonic comparison networks to reduce the complexity of the problem. We will prove that the double butterfly network is rearrangeable for $N \leq 32$ and provide a routing algorithm for any required permutation.

The following is a summary of the subsequent chapters and sources of the topics in each chapter:

Chapter 2: introduces the terminology used and background material.

Chapter 3: introduces the unified formulation of the grid representation, which is applicable for both the line representation and the mesh-type interconnection representation. In the effort to demonstrate

the beneficial aspects of the dual representation, we review some of the past algorithms and convert those algorithms into our framework. This method enables the standardization of the existing algorithms, and provides a better understanding of the concept behind each sorting technique. In the process of standardizing the formulation of merge algorithms, we construct two efficient q–way r–merge algorithms that use $(r-2)q$–comparators to merge r sorted vectors, where both algorithms have time upper bounds that are reasonably tight under the standard mesh-type interconnection networks (any processor from such a network has at most four connections to its immediate neighbours only). Chapter 3 starts with algorithms from various past papers, where we standardize and modify each algorithm under our new framework. The complete q–way r–merge algorithm is a new sorting algorithm. The most important contribution of this chapter is the introduction of the merge paradigm that can be generalized to describe most of the q–way r–merge algorithms. Furthermore, we were able to construct more efficient new algorithms (in terms of the depth of networks and the number of k–comparators required).

Chapter 4: presents a new class of parallel merge algorithms, which is more elegant and systematic. In fact, the k–comparator networks constructed from this class of merge algorithms do not need any rewiring. Hence the algorithms from this class are more conducive to merge schemes. The work done in Chapter 4 is original, except the m–cochain algorithm is from Becker and Litman [10]. This new class of merge algorithms has excellent packaging property and can be easily implemented to sort in three-dimensional interconnection networks.

Chapter 5: introduces new concepts in solving the problem of the rearrangeability of the double butterfly networks. Here we treat the rearrangeability problem of the double butterfly networks, for $N \leq 16$, where we construct an algorithm which produces the required connection in a double butterfly network for any given pair of input vector and output vector. We also discuss how to extend this algorithm for $N \geq 32$.

Chapter 6: contains conclusion and remarks for future works.

Chapter 2

Ingredients of Sorting Networks, Notation and Terminology

To facilitate a better understanding of this thesis, in this introductory chapter, we will introduce concepts that are fundamental to sorting. Most of the terminology used in this chapter is from Cormen, Leiserson and Rivest [31] and Knuth [49].

The chapter is organized as follows: In section 1, we introduce the basic definitions for comparison networks and parallel sorting networks. We also introduce the framework of the line representation. In section 2, we examine the zero-one principle, which is fundamental to the study of comparison sorting networks. We provide a generalized proof for this principle. In section 3, we examine two parallel networks: the butterfly network and the Omega network. In section 4, we introduce the planar interconnection networks and the mesh representation. Then we examine a few of the different indexing schemes on an interconnection network. In section 5, we introduce the grid representation of a parallel network. In section 6, we conclude by giving a list of references related to the study of parallel interconnection networks.

2.1 Introduction to Comparison Networks and Sorting Networks

Sorting is one of the most extensively studied problems in computer science. The ability to sort quickly is important in a wide range of practical applications. A reduction in the time spent on the task of sorting allows the freed computer resources to be used for other applications.

One can design efficient parallel sorting networks from comparators and wires: a comparison network is comprised solely of *wires* and *comparators*. A **comparator** (Figure 2.1(a)) is a device with two inputs,

x and y, and two outputs, $\bar{x}$ and $\bar{y}$, that performs the following function:

$$\begin{aligned} \bar{x} &= \min(x, y), \\ \bar{y} &= \max(x, y). \end{aligned}$$

A well-developed theory exists for sorting and merging networks using comparators with two inputs and two outputs; in the later sections of this chapter, we will give a list of papers reviewed. However, as the technology evolves, it become possible to fit a lot more circuits on a single chip and so occupy less area of a circuit board (than what was permitted by earlier technology). Hence it is reasonable to consider physical components which can sort inputs and outputs of size k, $k \geq 2$, and we can use such chips as building blocks for more general sorting networks. We will call such a building block a *k-comparator*, so that the usual comparators are the 2-comparators. There are numerous algorithms that can be considered in this framework. Two such algorithms are the column sort of Leighton [56] and the shear sort in Schnorr and Shamir [89].

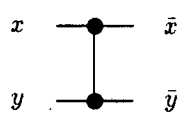


Figure 2.1: A comparator with inputs x and y and outputs $\bar{x}$ and $\bar{y}$, drawn as a single vertical line, using Knuth's line representation.

The notation we adopt is to denote a sequence of k elements as $\vec{x} = (x_i)_{i=0}^{k-1} = (x_0, x_1, \ldots, x_{k-1})$ and let $\bar{\vec{x}} = (\bar{x}_i)_{i=0}^{k-1} = (\bar{x}_0, \bar{x}_1, \ldots, \bar{x}_{k-1})$ be the sorted sequence corresponding to $\vec{x}$. Furthermore, given a sequence $(x_i)_{i=0}^{k-1}$, the *rank* of an element $x_j \in (x_i)_{i=0}^{k-1}$ is the index of x_j in $(\bar{x}_i)_{i=0}^{k-1}$. So the rank of an element in a sequence of distinct elements is the number of elements in this sequence that are less than it. For example, given a sequence $(3, 19, 10, 2)$, then $(2, 3, 10, 19)$ is the sorted sequence. Hence the rank of 2 in this sequence is 0, the rank of 3 is 1, the rank of 10 is 2 and the rank of 19 is 3.

Sometimes we use the data element contained in the wire to label the wire, when it is convenient to do so. Hence a **k-comparator** has $x_0, x_1, \ldots, x_{k-1}$ as input lines and $\bar{x}_0, \bar{x}_1, \ldots, \bar{x}_{k-1}$ as output lines. See Figure 2.2 for the Knuth's line representation of a k-comparator.

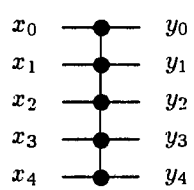


Figure 2.2: k-comparator where $k = 5$.

The second hardware component of the comparison network is the **wire**. A *wire* transmits a value from place to place in the comparison network. Wires can connect the output of one comparator to the input of another comparator. Otherwise they are either the network input wires or the network output wires.

Before we give a formal description of the comparison network, we want to clarify the usage of the term "comparison network" in this thesis: In the past literature, comparison networks refer to any network that is constructed from 2–comparators and wires. However, here we use the term "comparison networks" in a boarder sense. Comparison networks refer to any network which is constructed from k–comparators and wires. In certain instances, when we need to distinguish between a network that uses 2–comparators and a network that uses k–comparators, then we shall call the network a 2–comparator network and a k–comparator network respectively.

In order to develop the theory of sorting, it is more convenient to represent the comparison networks in slightly different way, where we use the representation of Knuth [49]. A comparison network of size n is a set of comparators interconnected by wires, where there are n input terminals at the left of the network and n output terminals at the right of the network. We draw a comparison network on n inputs as a collection of n horizontal lines with comparators stretched vertically. Note that a line from an input terminal to an output terminal does not represent a single wire, but rather a sequence of distinct wires connecting various comparators. The main requirement for interconnecting comparators is that the graph of interconnections must be acyclic: if we trace a path from the output of a given comparator to the input of another comparator or an input terminal or an output terminal, the path we trace must never cycle back on itself and go through the same comparator twice. We draw a comparison network in Figure 2.3 to illustrate the idea, where network inputs on the left and network outputs on the right; data moves through the network from left to right. A k–comparator only processes the data elements from the lines that are jointed to the vertical wires with small circles.

One further assumption needed for the comparison network: each comparator produces its output values only when both of its input values are available to it. Since some of the disjoint comparisons are simultaneously executed, it is convenient for us to introduce a new notion: the *depth* of a wire, the *level* of a comparator or a switch, and a *stage* of a parallel network (a parallel network is either a comparison network or a switching network, where we will give a formal definition of switching networks in Chapter 5).

Under the assumption that each comparator takes unit time, we can define the *running time* of a comparison network as the time it takes for all the output wires to receive their values once the input wires receive theirs.

In Cormen, Leiserson and Rivest [31], the **depth** of a wire is recursively define as follows: an input wire of a comparison network has depth 0. Now, if a comparator has two input wires with depths d_x and

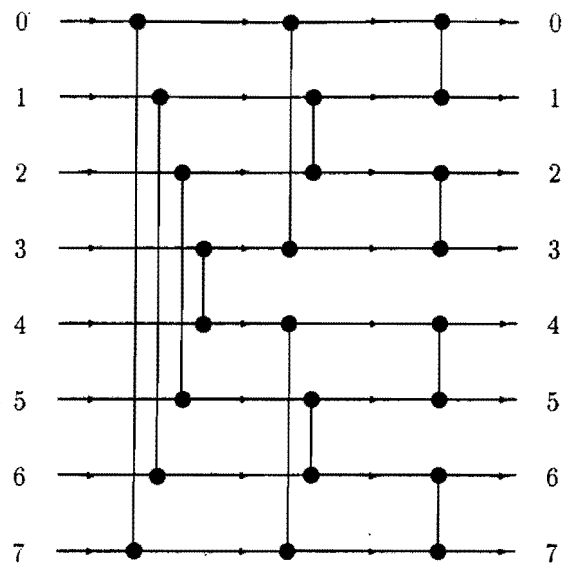


Figure 2.3: A comparison network, with 2–comparators.

d_y, then its output wires have depth $\max\{d_x, d_y\} + 1$. Because there are no cycles of comparators in a comparison network, the depth of a wire is well defined. Next the **level** of a comparator or a switch in a parallel network is defined to be the maximum depth of its input wires. The collection of comparators from the same level forms a **stage** in the comparison network.

Lastly, a sorting network takes any n–vector as input and produce a monotonically increasing n–vector. It should be clear that not all the comparison networks are sorting networks. Henceforth, to distinguish the comparison sorting networks from other comparison networks, the comparison sorting networks will be called the k–comparator sorting networks.

2.2 The Zero-One Principle

In this section we introduce the $[0-1]$ **principle**, which says that if a comparison network correctly sorts any input sequence drawn from the set $\{0, 1\}$, then it will correctly sort any input sequence with arbitrary real numbers. The $[0-1]$ **principle** is very useful, since it allows one to focus on checking if the sorting network is correct for any input sequence consisting solely of 0's and 1's. Once the sorting network is constructed and it is proven that it can sort all sequences consisting solely of 0's and 1's, we can then appeal to the $[0-1]$ **principle** to show that it properly sorts sequences of arbitrary values.

The $[0-1]$ **principle** has been proven often in the past literature. We found that the approach used to prove the $[0-1]$ **principle** in Cormen, Leiserson and Rivest [31], is best suited for proving the generalized version, where the $[0-1]$ **principle** is proven to be correct for any k–comparator network

($k \leq 2$), not just for the 2-comparator network.

Claim: If f is a monotonically increasing function, then a single k–comparator with inputs $f(x_i)$, $0 \leq i \leq k-1$ and $k \geq 2$, produces $f(\bar{x}_i)$ at the i^{th} output terminal.

Proof: The proof of this claim follows immediately from the inequalities $\bar{x}_0 \leq \bar{x}_1 \leq \ldots \leq \bar{x}_{k-1}$ which imply $f(\bar{x}_0) \leq f(\bar{x}_1) \leq \ldots \leq f(\bar{x}_{k-1})$. ∎

Lemma 2.2.1 *If a comparison network transforms the input sequence $(x_i)_{i=0}^{n-1}$ into the output sequence $(y_i)_{i=0}^{n-1}$, then for any monotonically increasing function f, the network transforms the input sequence $(f(x_i))_{i=0}^{n-1}$ into the output sequence $(f(y_i))_{i=0}^{n-1}$.*

Proof: We will prove this lemma by induction on the depth of each wire in a standard comparison network: Let a wire be at depth 0 (an input wire of the network). Let the input values be x_j when the input sequence $(x_i)_{i=0}^{n-1}$ is applied to the network. Then the statement is trivially true, since the wire assumes the value $f(x_j)$ when the input sequence $(x_i)_{i=0}^{n-1}$ is applied.

Assume the statement is true for any wire which has depth less than d, where $d \geq 0$. Now suppose we have a wire at depth d. Then by definition it is an output wire of a k–comparator in level $d-1$. Thus the input wires to this k–comparator are at a depth strictly less than d, by the inductive hypothesis, if the input wires to the k–comparator carry values from the input sequence $(x_i)_{i=0}^{n-1}$, then they carry corresponding elements from input sequence $(f(x_i))_{i=0}^{n-1}$, when f is applied.

Next, from the result obtained from the Claim above, the k–comparator with inputs $f(x_i)$, $0 \leq i \leq k-1$ and $k \geq 2$, produce $f(\bar{x}_i)$ at the i^{th} output terminal. Since the k–comparators in level $(d-1)$ are mutually disjointed, and the wire is arbitrarily chosen, the statement is true for any wire of depth d. Hence the lemma is proved. ∎

In the next theorem we shall prove the $[0-1]$ **principle**.

Theorem 2.2.2 *If a comparison network with n inputs sorts any possible sequences of 0's and 1's correctly, then it sorts any sequences of arbitrary numbers correctly.*

Proof: Suppose for the purpose of contradiction that the network sorts all $0-1$ sequences, but there exists a sequence of arbitrary numbers that the network does not correctly sort. That is, there exists an input sequence $x_0, x_1, \ldots, x_{n-1}$ containing elements x_j and x_i, where $x_j > x_i$. Furthermore, the network places x_j before x_i in the output sequence. Now define a function f as follow:

$$f(x) = \begin{cases} 0 & \text{if } x \leq x_i \\ 1 & \text{if } x > x_i \end{cases}$$

The function f is clearly monotonic increasing.

Since the network places x_j before x_i in the output sequence when $(x_0, x_1, \ldots, x_{n-1})$ is the input sequence, it follows from Lemma 2.2.1 that it places $f(x_j)$ before $f(x_i)$ in the output sequence when $(f(x_0), f(x_1), \ldots, f(x_{n-1}))$ is the input sequence. But since $f(x_j) = 1$ and $f(x_i) = 0$, we obtain the contradiction since the network fails to sort the $0-1$ sequence $(f(x_0), f(x_1), \ldots, f(x_{n-1}))$ correctly. ∎

2.3 Interconnection Schemes

Efficient construction and functioning of a multiprocessor system depends crucially on the type of interconnection used in a network. Here we shall not deal with the relative efficiency of the different types of interconnection networks (Raghunathan and Saran [82] give a comparison between the Shuffle/Exchange network and the butterfly network, Leighton [58] gives an in-depth comparison between various interconnection schemes). We focus on two types of interconnections: The Shuffle/Exchange network and the butterfly network.

We introduce these two particular networks because: firstly, there are many other variations of the butterfly network, and secondly, these two parallel networks are very popular, and there is extensive literature dealing with these two interconnection schemes. The materials from this section are corresponding to the work that will be presented in Chapter 5.

In what follows, we describe the butterfly network and the Shuffle/Exchange network:

- The n–level butterfly network has 2^n input terminals and 2^n output terminals. There are n levels of comparators in this network. At each level i, $0 \leq i \leq n-1$, there are 2^{n-1} 2–comparators. If we use the Knuth's line representation to describe the butterfly network, and we label the input lines by n–bit binary numbers, then a 2–comparator in the i^{th} level of the network will connect input lines which differ in precisely the $(n-i)^{th}$ rightmost bit. See Figure 2.4 for an illustration of the 3–level butterfly network.

Before we introduce the Shuffle/Exchange network, we need to introduce the perfect shuffle operation. To describe the perfect shuffle operation (or the shuffle operation for short), we will use the analogy used in Stone [93]: Given an N–vector, where N is divisible by 2, the shuffle operation of this N–vector is a permutation of the elements that is identical to a perfect shuffle of a deck of cards. First, we divide the N elements into two equal parts (the first $\frac{N}{2}$ elements and the last $\frac{N}{2}$ elements), then we interleave the elements of the first half with the elements of the second half. That is alternate inserting the elements from the two different subsets as shown in Figure 2.5(a). In terms of the connecting 2–comparators, the i^{th} output terminal is connected to the $((2i + \lfloor \frac{2i}{N} \rfloor) \bmod N)^{th}$ input terminal of the next level, where $0 \leq i \leq N-1$.

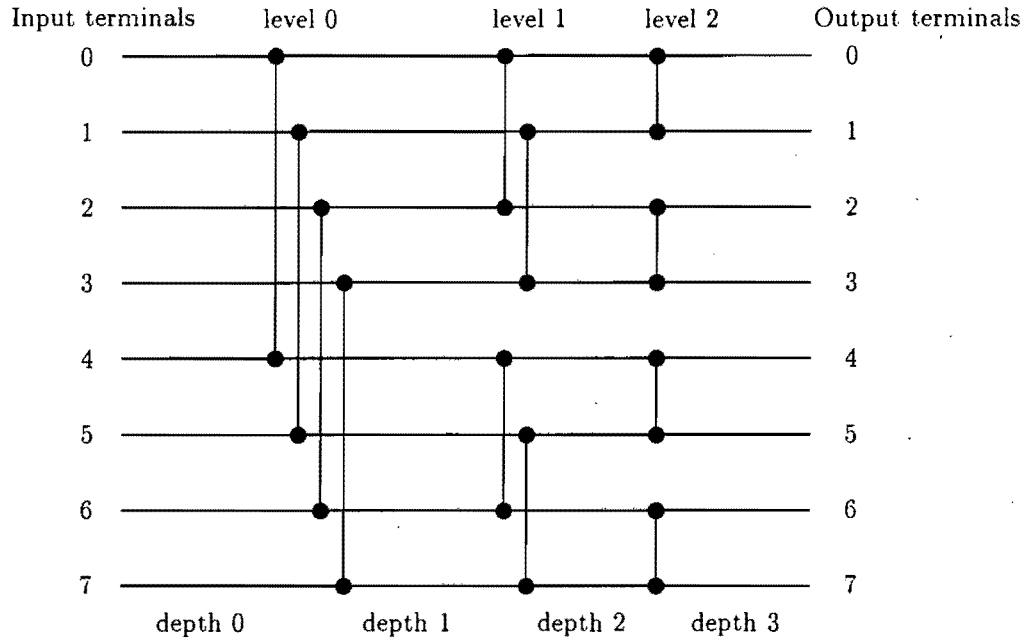


Figure 2.4: The 3-level butterfly network, where $N = 8$.

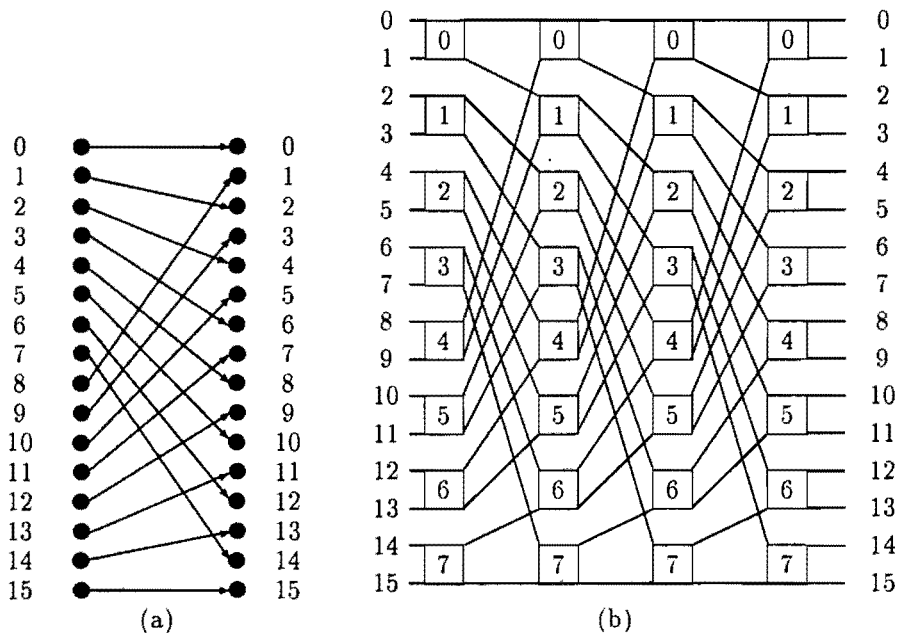


Figure 2.5: (a) The perfect shuffle of an 16-vector and (b) An Omega network for $N = 16$.

Another view of the shuffle is related to the binary representation of the indices of the elements of the vector: The i^{th} element of N-vector is shuffled to the position i', where i' is the number obtained by cyclically rotating the bits in the binary representation of i one bit position to the left.

Here is the formal definition of the Omega networks:

- The Omega network (It is sometime called the Shuffle/Exchange network) of size 2^n is constructed from n identical stages in series. See Figure 2.5(b) for an illustration of the Omega network.

Although the butterfly network and the Omega network need to use different network representations, they are shown to be isomorphic to each other. We refer reader to Leighton [58] for the in-depth discussion and Parker [74] for various Shuffle/Exchange type switching networks.

Depending on the connection scheme used to connect comparator modules, a parallel network will have different characteristic properties, such as **reachability**, **connectivity** and **rearrangeability**.

In a network, the output terminal w is *reachable* from the input terminal v, if there is a connection from v to w. A network with N inputs and N outputs is said to be *connected* when for any pair of input terminal and output terminal, there exists a path which connects this input terminal to the corresponding output terminal. This implies, for any pair of output terminals w and input terminal v, w is reachable from v. A network with N inputs and N outputs is said to be *rearrangeable*, if for any one-to-one mapping π of the inputs to the outputs, we can construct edge-disjoint paths in the network connecting the i^{th} input to the $\pi(i)^{th}$ output for $0 \leq i \leq N-1$. Note that if a network is rearrangeable then the network is connected, but the converse is false.

2.4 Two-Dimensional Interconnection Networks: the Mesh Representation

Thus far, we have focused most of our attention on the notation for one-dimensional interconnection networks. However, higher dimensional interconnection networks are also possible. We are especially interested in the two-dimensional interconnection networks: the mesh representation. The mesh representation is a very attractive and practical architecture for parallel processing. This is because of the simple, modulo interconnecting pattern, which makes it easy to construct and program. Furthermore, the mesh structure is amenable to VLSI implementation: the advancement in VLSI technology implies it is possible to connect a large number of processors into a network, where each processor contains numerous processing chips. The processors are connected together by wires in the network. Hence two-dimensional interconnection networks are better suited for the representation of VLSI structures.

Mechanically modifying algorithms that were designed for ordinary computers for the VLSI structure

has largely proven a failure, since it is difficult to make use of the enormous parallelism of VLSI technique. However, in the next two chapters on sorting we shall show that the usage of the k–comparators may be the solution to this problem, where we regard the k–comparators as an extension of the 2–comparators.

In the mesh representation, where the elements of the interconnection network are organized in an $m \times n$ mesh-type processor array, the interconnection network can either be a square mesh or a rectangular mesh (a discussion on whether the square interconnection network or the rectangular interconnection network is optimal can be found in Bar-Noy and Releg [7]).

There are many different ways of indexing the elements of a mesh-type interconnection network, we will introduce four schemes on the square interconnection network. However, these schemes are not restricted to work on the square interconnection network alone, they can also apply to the rectangular interconnection network. We use Figure 2.6 to illustrate the indexing schemes.

I. Row-major indexing (Column-major indexing).

II. Snake-like Row-major indexing (Snake-like Column-major indexing).

III. Reversed Row-major index.

IV. Inverse Row-major index.

0	1	2	3
4	5	6	7
8	9	10	11
12	13	14	15

(a)

0	1	2	3
7	6	5	4
8	9	10	11
15	14	13	12

(b)

0	4	8	12
1	5	9	13
2	6	10	14
3	7	11	15

(c)

3	2	1	0
7	6	5	4
11	10	9	8
15	14	13	12

(d)

15	14	13	12
11	10	9	8
7	6	5	4
3	2	1	0

(e)

Figure 2.6: We use a 4×4 mesh-type interconnection network to illustrate the different indexing scheme: (a) Row-major index. (b) Snake-like Row-major index. (c) Column-major index. (d) Reversed Row-major index. (e) Inverse Row-major index.

2.5 The Grid Representation

The last representation we want to introduce is the grid representation. The grid representation of a parallel network is the most vital concept introduced in this research work. The introduction of grid structure enables us to interpret sorting operations described in terms of k–comparator networks into operations that are easier to visualize on a two-dimensional array.

The dual notions of using the grid representation and incorporating the line representation of the k–comparators will become important in representing the sorting algorithms in the next two chapters

of this thesis. Representing sorting networks using these two representations simultaneously will give a much better understanding of the underlying technique of the sorting algorithms.

In the grid representation, the processor elements are arranged into a square array or a rectangular array, where any indexing scheme introduced for the interconnection network can also applied to the elements of a grid. However, in the chapters to follow, we shall use the row major indexing scheme mostly.

The major difference between an interconnection network and a grid is the following: in an interconnection network, any element in the array has predefined connections to the elements that are adjacent to it, whereas in a grid array there are no connections assigned and we are free to sort any subpart of a grid. The fact that elements of a grid have no predefined connection makes it a useful designing structure. In the next chapter, we shall demonstrate how one can use a $p \times q$ grid structure A (a mesh-type grid) and a $(p.q)$–vector structure V (a sequential array) interchangeably.

Henceforth in this thesis, the term "interconnection network" is used to refer to any network that has fixed predefined interconnections, such as a mesh-type interconnection network or a planar interconnection network. The term "grid" is used to refer to any rectangular or square network that has no predefined interconnections. The term "k–comparator network" is to refer to any parallel network presented in terms of the representation adapted from Knuth's [49] line representation for networks of k–comparators. Networks that are from any one of the above three representations will be referred to as "parallel networks" collectively.

2.6 Background

One further aspect of sorting networks we have not pointed out yet, is that a sorting network can be used for routing. A routing algorithm needs to send appropriate data, in the one-to-one manner, to the designated terminal at the right time. Any sorting algorithm can be used to route data. The basic idea is to assign rank to each element of the input data. We rank the elements in the following way, that the element needs to be sent to the i^{th} terminal will be assign rank i. Hence, if the input data is sorted, the required data is sent to the designated terminal. We will give a formal discussion on this aspect in Chapter 5.

There are many aspects of interconnection networks that are not covered in this work. However, we present a list of publications that involve different aspects of the interconnection networks. Collections of various papers on different aspects of the interconnection networks can be found in Scherson and Youssef [88], Varma and Raghavendra [100]. There are survey papers such as: Feng [37], Broomell and Heath [23] on interconnection networks, Estivill-Costro and Wood [35] on adoptive sorting algorithms. There are papers on randomized sorting algorithms such as: Andersson et al [6], Plaxton [78], Leighton and

Plaxton [60], Kaufmann and Sibegn [47]. There are papers on the performance analysis of switching networks such as: Agrawal [2], Hen and Igarashi [43], Kahale et al [46], Kunde [52], Thompson [96], and Yoon, Lee and Liu [110]. There are papers on fault tolerant switching networks such as: Das and Dattagupta [33], Karlin, Nelson and Tamaki [41], Leighton, Ma and Plaxton [59], Leighton and Maggs [57], Parhami and Hung [73], Rudolph [84], Yao and Yao [107], Yen, Bastani and Leiss [109] and Varma and Raghavendra [99]. There are papers on modeling interconnection networks such as: Agrawal [1], Akers and Krishnamurthy [5], Bhatt et al [18], Carlsson et al [25], Li and Cheng [61], Saad and Schultz [85], and Scherson [87]. Furthermore, the time complexity lower bound found by Beigel and Gill [12].

Chapter 3

Standardized Sorting Algorithms

With the advent of VLSI technology, the integration of a large number of processing elements on a single chip is possible. Hence there is a need for algorithms exploiting the potentially high degree of parallelism in networks of such processing elements.

These are some properties a "good" VLSI algorithm should have:

1. It can be implemented using a few types of simple processors.

2. Its data and control flow are simple and regular, so the processors can be connected by a network with local and regular interconnections.

3. It uses extensive pipelining and parallel processing.

Parallel sorting algorithms and their efficient implementations on VLSI chips have been studied intensively. One of the earliest algorithms for sorting on rectangular mesh-type interconnection networks was published by Thompson and Kung [95]; their two algorithms use unit-distance routing steps and 2-comparators. Subsequent algorithms were published by Nassimi and Sahni [66] and Kumar and Hirschberg [50].

In the realm of logic design, there are different possible schemes that one can choose from when implementing a parallel sorting algorithm. It is not our purpose to carry out a comparative evaluation of the different schemes and the corresponding sorting algorithms. Indeed, the choice of which scheme to follow depends highly on the adopted model of the computer on which the algorithm is to be implemented. By applying the grid representation to represent a comparator network, we provide a comprehensive framework for representing a comparator network. Furthermore, such a homogeneous presentation will provide comparative evaluation of comparison networks from various schemes.

As we mentioned in Chapter 2, the grid representation can be regarded as sorting using the line repres-

entation and the mesh representation (the interconnection network type of representation) concurrently. This ability to consider sorting algorithms in a dual interchangeable manner can be beneficial for finding better and simpler merge algorithms.

One of the representations that can be modelled by the grid is any algorithm that uses the line representation, which is adapted from Knuth's [49] line representation for networks of k–comparators. In this representation, horizontal lines represent inputs and vertical lines with dots at certain intersections represent k–comparators. This implies that our algorithms will use input vectors with a sequential indexing scheme. *The first rule of our scheme is that the network using the line representation will be restricted to using k–comparators that sort in the same direction only.* The k–comparator sorts in increasing order from the top; we will call such k–comparators the *standard* k–comparators. None of the k–comparators in our line representation will sort in the opposite direction (sort in decreasing order from the top); we call such k–comparators the *reverse* k–comparators. However, if an algorithm does need to sort in the reverse direction, we can always use the standard k–comparators and rewire the connection.

The second representation that can be modelled by the grid are those algorithms that use the mesh representation. Although, the mesh-type algorithms tend to be most efficient on interconnection networks with snake-like row major indexing scheme, the grid representation will not use this indexing scheme. The reason is that the grid representation is done in conjunction with the line representation. The network using the line representation can be converted to use the row major indexing scheme or the column major indexing scheme in the grid, where we regard the input N–vectors as $m \times n$ arrays. The choice of using the row major indexing scheme or the column major indexing scheme enables us to use standard k–comparators having the same direction of data flow (the reverse direction is not allowed). This restriction simplifies the global control of the processors. It also standardizes the existing algorithms, such that we are able to have a better understanding of algorithms that used different schemes. The standardizing scheme allows us to achieve a better understanding, because the scheme puts less emphasis on the implementation aspects of each algorithm and focuses on the aspects of the sorting technique. We feel this is justified, because the VLSI technology will be able to construct such a chip for the algorithm, if the sorting technique is sound and efficient. Hence *the second rule of our scheme is that the network is restricted to use row major indexing or column major indexing only, in the grid representation.*

Under the scheme discussed above, we were able to convert a number of algorithms that are implemented to sort on the two-dimensional interconnection networks: Shear sort and the merge algorithm of Scherson and Sen [86], revsort and the merge algorithm of Schnorr and Shamir [89], and the merge algorithms of Lang et al [53]. However, we should mention that not every sorting algorithm on the interconnection network can be easily converted into our new scheme. For example, the algorithms in Kumar and Hirschberg [50], Nassimi and Sahni [66], Thompson and Kung [95] will be too complex to be converted, since those algorithms do not have good VLSI packaging property, which implies it is difficult to preserve the underlying properties of such algorithms using k–comparators.

To summarize, the grid representation of the algorithm can be regarded either as sorting on a mesh-type interconnection network using either the column major or row major indexing order, or as sorting, in line representation, using the standard k-comparators, which is analogous to the 2-comparator network. Furthermore, the restriction of using two similar indexing schemes eliminates the difference arising from algorithms that use different indexing schemes, and gives a better understanding of the underlying sorting technique.

One last motivation is the aspect of packaging. The partitioning of the system into physical sub-systems is a cardinal task when building large scale digital systems. It is desirable to have few different types of sub-systems, especially when the sub-systems are manufactured by mass production techniques, and when the sub-systems are not general-purpose ones. For a given physical size of a sub-system, it is usually preferable to reduce the total number of sub-systems in a system, rather than the total amount of logic in the system. There are two sorting modes (in Cole and Siegel [30]): Perimeter sorters, which have their input/output ports located along the boundary of the sub-system and the dense sorters, which can have input/output ports located anywhere within the sub-system. However, in both cases, the input/output links have to cross the sub-system's boundary, while the logic is within the package. Therefore, in many cases, the input/output capability of the sub-system is the dominant factor in the system partitioning.

It is usual, when designing sorting networks, to neglect the fact that when used in practice, the networks will be packaged. It is desirable to exhibit how the topology of the network depends not only on the number n of keys, but also on the I/O capacity of the intended packages. For a discussion of packaging the Butterfly and Batcher's networks, see Litman [63]. We present here practical sorting networks that are better than the Batcher's networks with respect to packaging. Packaging is used to partition a network into components (e.g., chips) such that each component has at most k inputs and at most k outputs with the goal to minimize the number of components.

In an adequate implementation of the system partitioning, we are able to solve the problem of sorting a set of data, which is too large to be entirely contained in the main memory. In such a case the data to be sorted is split into segments, which can be kept in a storage device and sorted separately. The sorted segments are then combined with various strategies, depending on the nature of the memory device and the algorithms used. A similar problem arises in a VLSI environment, when VLSI chips are of fixed size, and the data set is larger than the capacity of the chip.

This problem is relevant because of the following issues:

1. It can be expected that the size of most VLSI sorting devices will be limited by the technological constraints, thus it is impractical to assume the applicability to unbounded data sets, unless appropriate strategies are devised.

2. One needs to consider the effect on the cost, when one is to manufacture a large circuit. The cost of a large circuit varies exponentially with the area (refer to Ullman [97] for an in-depth discussion): When fabricating a chip, there is the opportunity for an imperfection, such as a minute piece of dust, to introduce a flaw somewhere in the circuit. A circuit with a flaw can be assumed to be useless (unless some technology to repair or bypass the flow can be implemented, such as the correction network for n–comparators introduced in Schimmler and Starke [90]).

   To elucidate the second point above we used an example from Ullman [97]: The largest circuits that it is feasible to fabricate have a very low probability of being produced unflawed; the yield is very low. If we double the area of such a chip, the probability of both halves being unflawed is the square of that probability. In general, if we multiply the area by c, the probability of finding flaw in the chip is increased by the power of c . Since the cost of producing a good chip is inversely proportional to the yield, the cost per chip would grow as c grows past 1. Of course, if the area of the chip were small to begin with, so the yield was close to 1, the cost would not be a very sensitive function of the area at all.

3. We need to consider the speed of circuits. Often there is a trade-off between the area needed and the time spent performing a function.

With the above three issues in mind, we propose that merge algorithms, implemented with fixed size k–comparators, can be good candidates to alleviate the above issues satisfactorily. Furthermore, the highly parallel structure of the merge algorithm means that it is possible to achieve a satisfactory trade-off between the area needed and the time spent performing a function. In this chapter and the following chapter, we will demonstrate how a class of such merge algorithms can be constructed.

In the literature there are host of merge algorithms: Preparata's parallel-sorting schemes [79], Nakatani et al introduce the simple k–way Bitonic sort [64], Bilardi and Preparata's more advance structure of Bitonic sort [19] and a VLSI sorting network [20], Hagerup and Rüb's optimal algorithm on the EREW PRAM [40], Wen [102] and Hayashi and et al [42]. This resurgent of intense interest on the subject of r–mergers, which can be contributed to the advancement of the VLSI architecture and the ability of the process elements to execute instructions efficiently in parallel. The goal is to find faster parallel merging algorithms in optimal time, for some practical models. The most used model in the literature is the parallel random access machine (PRAM). There are three categories of PRAM: the exclusive-read and exclusive-write (EREW), concurrent-read and exclusive-write (CREW) and concurrent-read and concurrent-write (CRCW). Hagerup and Rüb [40] described an optimal EREW PRAM algorithm for merging two sorted sequences of total length n. Then, more recently, there are the papers of Cypher and Plaxton [32], Wen [102] and Hayashi, Nakano and Olariu [42], which formulated optimal PRAM algorithms for merging $r \geq 2$ sorted vectors. Cypher and Plaxton [32] introduced a deterministic sorting algorithm of $O(\log_2 n(\log_2 \log_2 n)^2)$. For example, if there are n elements and organized as 2^a sorted subvectors of length 2^b (assume $n = 2^{a+b}$), then the time complexity is $O(a(\log_2 a)^2)$. A shorter version

of this algorithm can be found in Leighton [58] also. Wen [102] presented an optimal CREW PRAM algorithm which takes $O(\frac{n \log_2 k}{p} + \log_2 n)$ time (p is the number of processes used and n is the total size of the input lists). Hayashi, Nakano and Olariu [42] presented optimal PRAM algorithms for all three categories. The algorithms presented in the above four publications followed a similar merge paradigm: A sample set is selected from the elements of the input vector, where the selection scheme is based on the ranks of the elements in the input vector, see Cole [28]. Next, the input vector is partitioned into blocks, with respect to the elements of the sample set, then a parallel merge sort is applied, see Cole [29]. The characteristic of the algorithms derived from this paradigm is that they are close to be optimal, with respect to the model selected. The reason for this is because the decomposition is based on the ranks of the elements in the vector, so the vector is partially sorted after the decomposition. Hence, if an efficient method of finding the sample set can be implemented, then the algorithm can sort the vector in lesser time. Furthermore, some of those algorithms are implemented in a pipelined fashion.

This chapter is organized as follows: In section one, we introduce the technique for presenting a grid (the grid representation) and some useful terminology. In section two, we review three well known merge algorithms, the odd-even merger, the bitonic sorting algorithm and the balanced sorting algorithm. For each merge algorithm, we introduce its implementation on a 2-comparator network, then we implement the merge algorithm on an $\frac{N}{2} \times 2$ grid. Using those three merge algorithms, we demonstrate how our merge paradigm is formulated, and how this merge paradigm can be used to unify numerous algorithms and standardize the formalism of merge algorithms. In section three, we describe the scheme of our merge paradigm in terms of operations on the grid. We want to emphasis that a grid array is different from an interconnection network: in an interconnection network, any element in the interconnection network has predefined connections to the elements that are adjacent to it, whereas in a grid there are no connections assigned and we are free to sort any subpart of a grid. We review some of the sorting algorithms that sort on rectangular mesh-type interconnection networks. We implement those merge procedures to sort efficiently on rectangular grids. Again, we use these algorithms to formulate the merge schemes and construct comparator networks. In section four, a class of merge algorithms is introduced. In subsection 4.1, we introduced four grid algorithms: The first two algorithms are from the family of q-way 2-merge algorithms that merge two sorted vectors on a $p \times q$ rectangular grid. They are generalization of the odd-even merge algorithm and the bitonic sorting algorithm presented in section two of this chapter. We describe the above two generalized algorithms in a format that is conducive to our paradigm. The last two merge algorithms are from the family of q-way r-merge algorithms. These two new algorithms are further generalization of the first two merge algorithms, where they merge $r > 2$ sorted vectors on a $p \times q$ rectangular grid. In subsection 4.2, we present the multi-way multi-merge algorithm, which uses the four grid algorithms as the underlying algorithm to construct a class of merge algorithms. We demonstrate how various k-comparator merge networks can be derived from the multi-way multi-merge algorithm. Then in subsection 4.3, we examine the time complexities of various k-comparator merge networks that are derived from the multi-way multi-merge algorithm. In section five, we formulate the

sorting algorithms using the multi-way multi-merge algorithm. Then we investigate the time complexities of the k–comparator sorting networks derived from the sorting algorithm. In section six, we investigate a variation of our 2–merge algorithm. From it we derive a different sorting algorithm, which has the same leading order term but has a better lower order term than the q–way 2–merge algorithm, in terms of the time delay. In section seven, we investigate possible constructions of k–comparators and the effect of different constructions on the time complexity. In section eight, we give a summary of the time complexity of various networks. In section nine, we discuss the application of the grid algorithms to the problem of sorting on rectangular and square interconnection networks.

3.1 Terminology

Let A be an m by n grid with elements v_{ij}. Now we are able to consider elements in the columns and the rows: let Row_i be the i^{th} row and Col_j be the j^{th} column where $0 \leq i \leq m-1$ and $0 \leq j \leq n-1$. Let V be an $(m.n)$–vector such that $V = \{v_{00}, v_{01}, \ldots, v_{0(n-1)}, v_{10}, \ldots, v_{1(n-1)}, \ldots v_{(m-1)(n-1)}\}$, where v_{ij} is an element at the i^{th} level of V, in steps of n. We call V the row major representation of A, hence we can regard V and A as representing the same data structure interchangeably. See Figure 3.1 for an illustration. Unless otherwise specified, we use the row major indexing scheme to index the elements of a grid. A vector of size N will be called an N–vector, where the size of a vector is the number of its components.

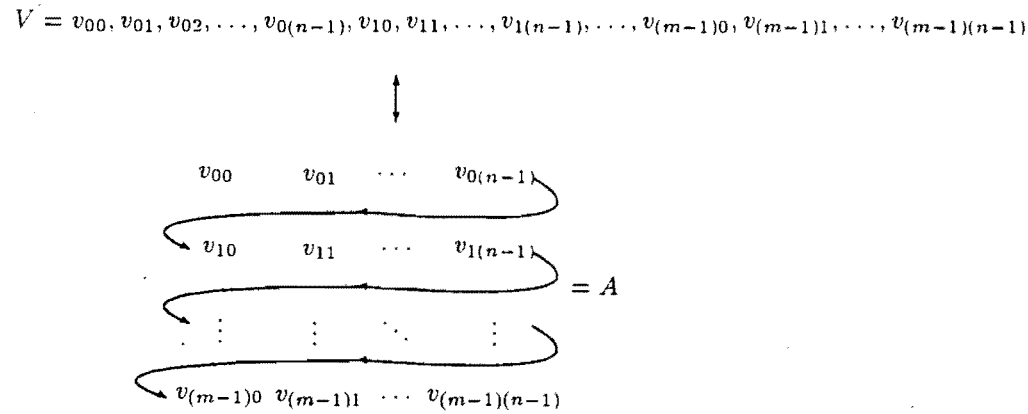


Figure 3.1: An example to illustrate how we regard V and A as representing the same data structure interchangeably.

One of the tools used intensively in the dual representation of the network is the usage of various types of decomposition chain. The reason why we use the decomposition chains is that they can be applied in both the line representation and the grid representation concurrently. One advantage of using the decomposition chain is that each type of decomposition chain can be constructed in a systematic fashion.

by partitioning the grid in a well-defined manner. Furthermore, the manner in which the decomposition chain is constructed is not restricted by the connection of a given array, hence they are well suited for the grid representation. The chain-decomposition indexes the elements sequentially, so they can be easily converted into a grid using either the row major indexing scheme or the column major indexing scheme. A given decomposition chain will be used to describe how the network is to be connected, so that efficient sorting can be achieved.

We define the first chain decomposition, which will be used extensively in our merge algorithms and some of the algorithms reviewed here (See Figure 3.2 for some examples of m–chains).

Definition 3.1 *The* m**–Chains**: *There are* n *such subvectors in* V, $(v_i, v_{i+n}, \ldots, v_{i+(n-1)n}, v_{i+n^2}, \ldots, v_{i+(m-1)n})$, *where* $i = 0, 1, \ldots, n-1$. *Elements belonging to the same chain are precisely those that have the same last digit in their* n*–ary expansion. An* m*–chain has* m *elements.*

We shall define other types of decomposition chains when they are needed. Before we start to review some of the past algorithms, we will introduce three types of transformers for V, which are sub-networks used in some of the sorting algorithms. Here are the three k–comparators networks, defined in terms of the Knuth's [49] line representation for network on V:

Definition 3.2

1 N*–*k**–Unitransformer**: *This network consists of one layer of* k*–comparators, where* $\frac{N}{k}$ k*–comparators, each applied to successive lines, starting with line 0.*

2 N*–*k**–Cotransformer**: *This network consists of two layers of* k*–comparators: the first layer has* $\frac{N}{k}$ k*–comparators, each applied to successive lines, starting with line 0; the second layer has* $\frac{N}{k} - 1$ k*–comparators, each applied to successive lines, starting from the* $(\lceil \frac{k}{2} \rceil)^{th}$ *line. Note: In the second layer, the first* $\frac{k}{2}$ *lines and the last* $\frac{k}{2}$ *lines are not operated on, since they are sorted already by the first* k*–comparator and the last* k*–comparator respectively, in the first layer of the cotransformer.*

3 N*–*k**–Bitransformer**: *This network consists of three layers; the first two layers are just the* N*–*k*–cotransformer, then a further layer is appended after it. The third layer has* $\frac{N}{k}$ k*–comparators, each applied to successive lines, starting with line 0 again.*

Observation: In terms of the grid, each m–comparator covers a subgrid of size m in the grid A. Sometimes, when we partition the grid in such a consecutive manner, we will call such subgrids *blocks* or *half shifted blocks* respectively. We will give a formal description of this in a later chapter.

We will use the $[0-1]$ **principle** to assist in the analysis of the complexity and the correctness

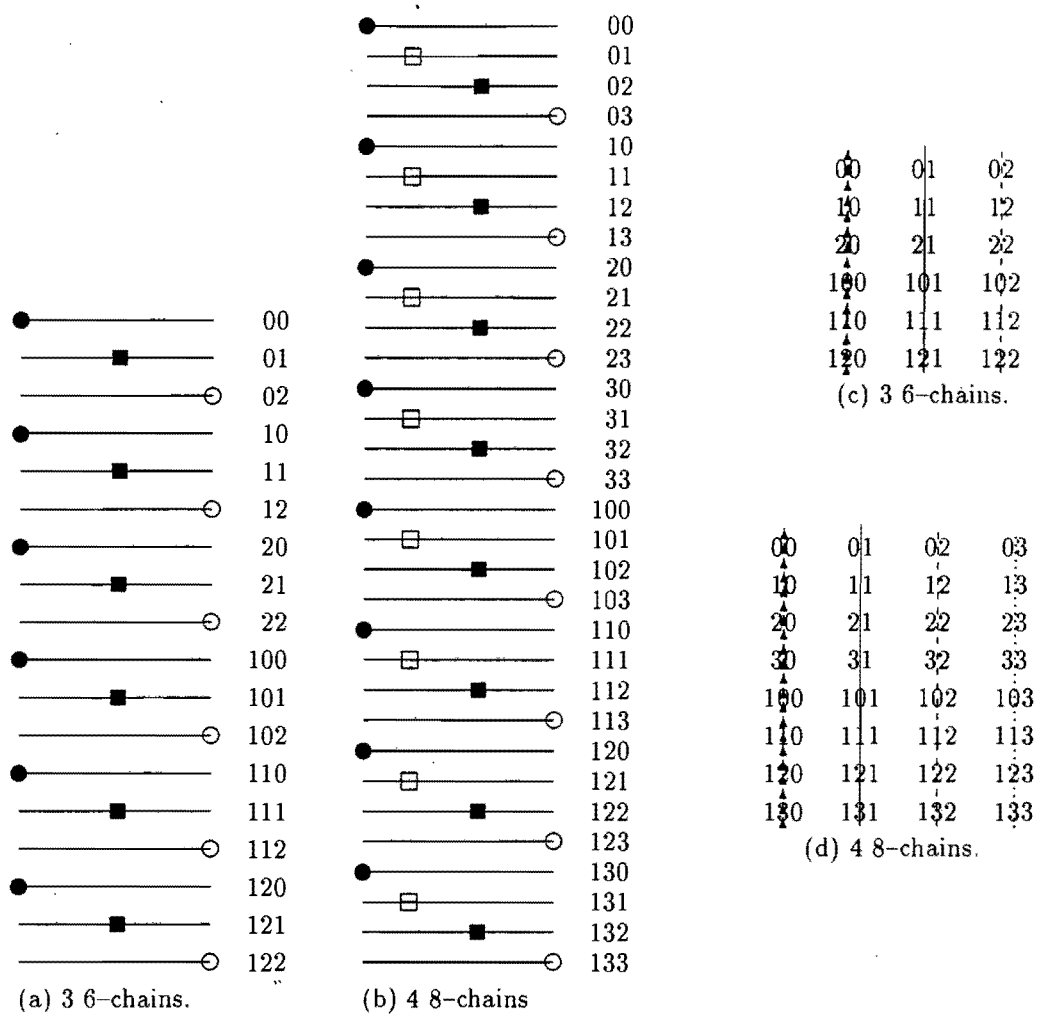
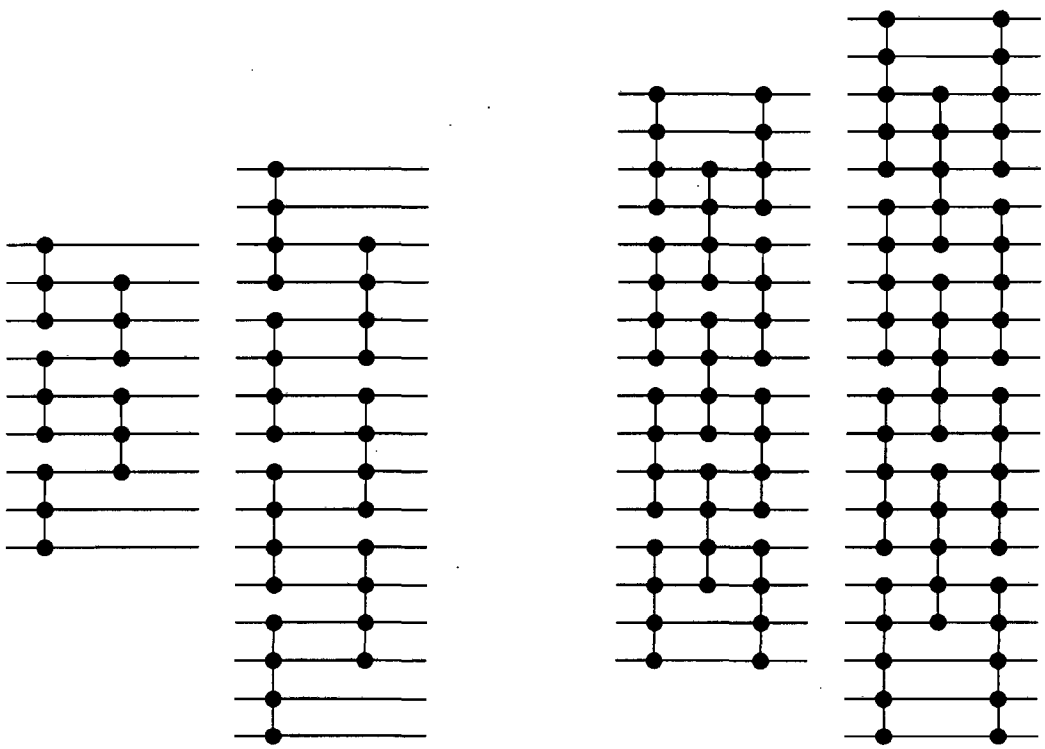


Figure 3.2: Examples of m–chains in both the line and the grid representations

of each algorithm. This principle is proved in Knuth [49] and we have proved the generalized version for k–comparators in Chapter 2. Furthermore, we will adopt the idea of "clean" and "dirty". A row (column) is refer to as being "clean", if it contains identical elements, viz., only 0's or only 1's whereas a row (column) consisting of both 0's and 1's will be called "dirty". We sometimes use all-1 and all-0 to distinguish clean rows (or columns) containing 1's and clean rows (or columns) containing 0's respectively.

Assume the elements of a grid or an N–vector are indexed in either row major or column major order. We sometimes need to refer to the movement of the elements in V. To facilitate this need, we will be following the norm: If we refer to the elements moving up the grid (or to the upper section of or to the top of V), then we are referring to moving the elements from positions of larger indexes to positions of smaller indexes. The opposite will be true if the elements are moving down the grid (or to the lower section of or to the bottom of V). If we did not specify which direction we are sorting a grid, then we



(a) Cotransformers (b) Bitransformers

Figure 3.3: Some examples of the cotransformers and the bitransformers.

are assuming the grid is sort in the increasing order from top to bottom.

3.2 The 2–Comparator Merge Sorting Algorithms

In this section, we will demonstrate how the dual representation is implemented. We illustrate this dual representation by means of three well known 2–comparator merge algorithms. We will examine and modify each of the following three algorithms: the Bitonic and Odd-Even networks of Batcher [8], and the balanced network of Dowd et al [34]. There are a number of other related networks constructed in Becker, Nassimi and Perl [9], which we will generalized in the next chapter. We will express each algorithm in our merge sorting paradigm, then we will generalize those popular 2–comparator mergers to k–comparator mergers, which will also be q–way mergers and have good packaging properties (to be good VLSI algorithms).

Nakatani et al [64] have already presented an extension of the bitonic sorting algorithm, and Liszka and Batcher [62] present an extension of the odd-even algorithm. Both extensions can be described by our merge paradigm. In the following sections, after we have introduce this new merge paradigm, we shall describe the above two algorithms using our merge paradigm, then we shall present a new algorithm, which is a generalized q–way r–merger using k–comparators.

To illustrate these ideas, we first show how the Bitonic, Balanced and Odd-Even merging networks lead to sorting algorithms on grids of size $\frac{N}{2}$ by 2. We will start with the Bitonic merge network and the definition of bitonic vectors:

Definition 3.3 *A vector V of real numbers is bitonic if*

1. *It is a concatenation of a monotonically increasing sequence U and a monotonically decreasing W, or if*

2. *V can be shifted cyclically so that condition (1) is satisfied.*

The Bitonic merger is shown in Figure 3.4(a) for $N = 8$. The two sorted sequences to be merged are placed on inputs 0 to $\frac{N}{2} - 1$, and $\frac{N}{2}$ to $N - 1$, the first is sorted in increasing order from the top, while the second in decreasing order. The network then produces a sorted sequence. Batcher's network consists of two merging networks of size $\frac{N}{2}$ (the solid and dotted networks) followed by a single parallel stage (shown in the box) which is called the transformer, as in Becker, Nassimi and Perl [9]. The corresponding algorithm on the grid is shown in Figure 3.4(b). The upper half of the grid consists of lines 0 to $\frac{N}{2} - 1$, and the lower half of the grid consists of lines $\frac{N}{2}$ to $N - 1$. Using this grid with row-major indexing, the algorithm is then:

The Bitonic Algorithm

1. Sort the upper half of the grid (the first block) and the lower half of the grid (the second block) in the directions: Sort the first block in increasing order and sort the second block in decreasing order. (This produces a bitonic vector).

2. Sort separately the two columns of $\frac{N}{2}$ elements connected by solid and dotted lines respectively. (These correspond to the solid and dotted parts of the merging network of Figure 3.4(a), to the left of the transformer. Mergers of size $\frac{N}{2}$ can be used to do these sorts since it can be shown that their inputs are made up of two halves sorted in the correct way).

3. Apply the transformer.

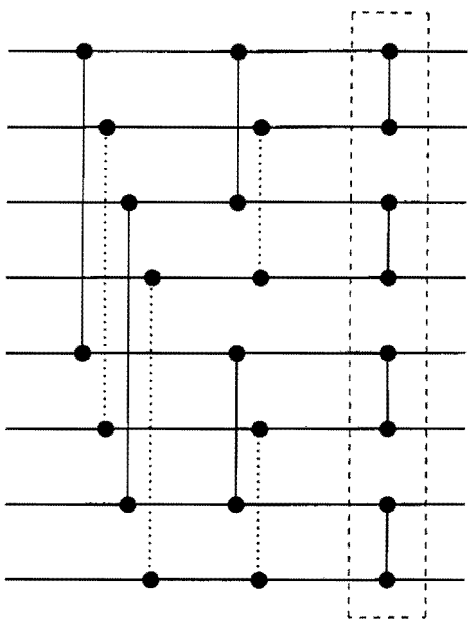
Now we make three observations:

i. This sorting procedure can be applied recursively to sort the two subgrids in Step 1. We then get the Batcher's Bitonic Sorting Network, with its corresponding sorting algorithm on the grid.

ii. The illustration in Figure 3.4(b) can be extended to $N = 16$ by adding a similar grid below the given one and joining the solid lines and the dotted lines in each half by a single solid or dotted line in each case. The extension to N an arbitrary power of 2 is then clear.

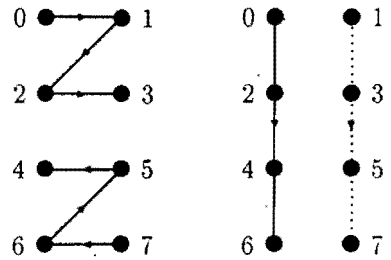
iii. In Step 2 of our grid representation in Figure 3.4, we apply a sub-network to sort each chain (in fact we used a 4–comparator), however, without the line representation we will not be able to see how this is implemented by the 2–comparators. We will skip on the detail of how the chains are sorted for now, after the formulation of the merge paradigm, we will see how this is achieved by applying the merge algorithms iteratively.

For the Balanced merging network, shown in Figure 3.5(a), the subvectors carrying the initial sorted sequences to be merged are lines $0, 2, 4, \ldots, \frac{N}{2} - 2$, and lines $1, 3, 5, \ldots, \frac{N}{2} - 1$, both sorted in increasing order from top down. Using the row-major indexing, these are the columns of the grid in Figure 3.5(b). An algorithm similar to the one presented above can be given to describe the Balanced merger, and is illustrated in Figure 3.5(b). However, there are a few modifications needed: First, the columns in Step 1 are sorted in the same direction (unlike the bitonic sort, where two subgrids are sorted in different directions). Second, the solid lines and dotted lines connect different sets of elements in Step 2. The solid-connected and dotted-connected elements are called *Cochains* in Perl [77], whereas for the Bitonic network, such sets of elements are called *Bichains*. Here is the algorithm:

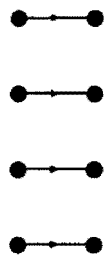


(a) Bitonic Network: $N = 8$.

Step 1 Sort each block in the direction indicated.



Step 2 Sort each column in the direction indicated.



Step 3 Apply Transformer.

(b) Bitonic Sort: $N = 8$.

Figure 3.4: Representing the Bitonic merger of size 8 using: (a) The line representation. (b) The grid representation.

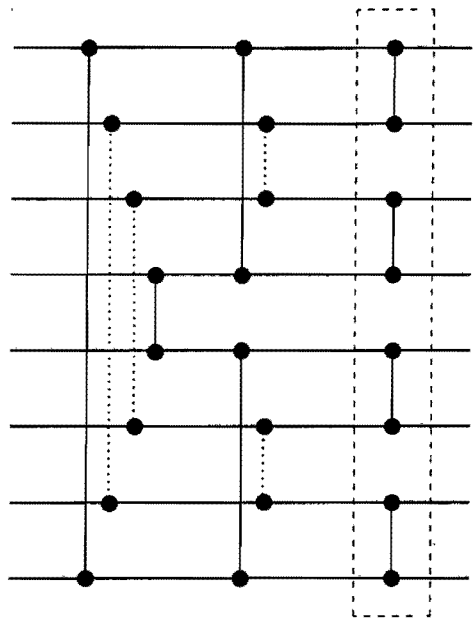
The Balanced Algorithm

1. Sort the columns in the downward directions. (This produces the sorted sequences to be merged).

2. Sort separately the two sets of $\frac{N}{2}$ elements connected by solid and dotted lines respectively. (These correspond to the solid and dotted parts of the merging network of Figure 3.5(a), to the left of the transformer. Mergers of size $\frac{N}{2}$ can be used to do these sorts since it can be shown that their inputs are made up of two halves sorted in the correct way).

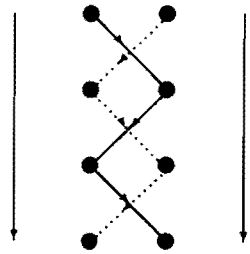
3. Apply the transformer.

For Batcher's odd-even merging network, shown in Figure 3.6(a), the sorted subgrids to be merged are in the same position as in the Bitonic network, but both are sorted in increasing order. Hence a similar algorithm to the one above can be applied. Again we use the row-major indexing on the grid. however, the algorithm needs few changes: First, the subgrids are sorted in increasing order, in Step 1. Second, the transformer in Step 3 is different from that of the Bitonic network. The sorting algorithm on the grid is illustrated in Figure 3.6(b). Here is the algorithm:

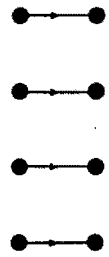


(a) Balanced Network: $N = 8$.

Step 1 Sort each column in the direction indicated.



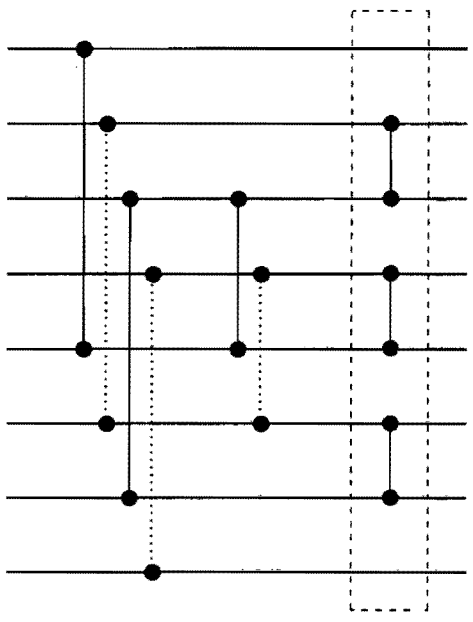
Step 2 Sort each chain downwards from top left.



Step 3 Apply Transformer.

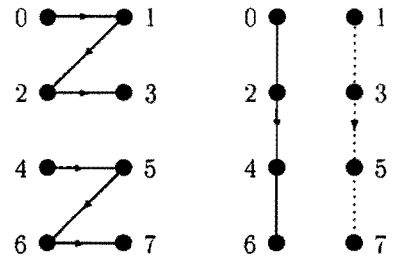
(b) Balanced Sort: $N = 8$.

Figure 3.5: Representing the Balanced merger of size 8 using: (a) The line representation. (b) The grid representation.

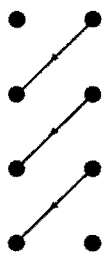


(a) Odd-Even Network: $N = 8$.

Step 1 Sort each block in the direction indicated.



Step 2 Sort each column in the direction indicated.



Step 3 Apply Transformer.

(b) Odd-Even Sort: $N = 8$.

Figure 3.6: Representing the odd-even merger of size 8 using: (a) The line representation. (b) The grid representation.

The Odd-Even Algorithm

1. Sort the subgrids in increasing order as indicated. (This produces the sorted sequences to be merged).

2. Sort separately the two sets of $\frac{N}{2}$ elements connected by solid and dotted lines respectively. (These correspond to the solid and dotted parts of the merging network of Figure 3.6(a), to the left of the transformer. Mergers of size $\frac{N}{2}$ can be used to do these sorts since it can be shown that their inputs are made up of two halves sorted in the correct way).

3. Apply the transformer.

Observation (i), (ii) and (iii) of the Bitonic algorithm above apply to both the Balanced and the Odd-Even networks. The above mergers are merging vectors using networks that uses $O(N \log^2 N)$ 2-comparators with $O(\log^2 N)$ levels of the 2-comparators.

This implementation of the dual representation on these three algorithms demonstrates the similarity between them and that they are following a similar paradigm. The aim of this prescribed scheme is to unify the presentation of the algorithms. This homogeneous framework will be valuable for the construction of the merge algorithms introduced in this chapter and the following chapter.

3.3 Sorting Algorithms on Rectangular Interconnection Networks

By examining the framework introduced in the previous section, by examples of Batcher's two merge algorithms and the balanced sort, we are now able to state the paradigm, informally: First, the grid array is decomposed into simple sub-parts and those sub-parts are then sorted. The sub-parts can be the rows or the columns or the sub-blocks (note: this process can be applied in an iterative manner into the sub-parts). Next sort some assigned decomposition chains of the grid. Finally apply a suitable sub-network, a transformer, which sorts the resulting vector. Furthermore, under the paradigm, **our sorting networks will be restricted to use one-size comparators only**. Henceforth, if we do not specify which indexing scheme the algorithm is using, we will **assume that the row major indexing scheme is applied**. Now we have an explicit scheme to implement merge algorithms on rectangular grids. In this section, we will test this merge paradigm on some past algorithms. We will demonstrate how those past algorithms that sort rectangular interconnection networks can be described by our merge paradigm.

There are numerous sorting algorithms on mesh-type interconnection networks. We are interested in

the algorithms from the following papers in particular: Leighton [56] gives two algorithms which use four stages of k–comparators where $k \geq 2(n-1)^2$ and $k \geq n(n-1)$ respectively. Parker and Parberry [75] introduce a modified version of Leighton's algorithms and they also gave a merge algorithm. Becker and Litman [10] introduce a new method which uses four stages of k–comparators where $k \geq n(n-1)$.

3.3.1 The Column Sort Algorithm

Now we will introduce the Column Sort Algorithm of Leighton [56]. This algorithm uses the column major indexing scheme on rectangular interconnection networks ($N = m \times n$ where $m \geq 2(n-1)^2$).

The Column Sort Algorithm

1. Sort all the columns downwards, independently.
2. Transpose the entries of the array. The entries are picked up column by column and then deposited row by row (always going from top to bottom in a column and from left to right in a row).
3. Sort all the columns downwards, independently.
4. Perform the inverse operation of Step 2.
5. Sort all the columns downwards, independently.
6. $\lfloor \frac{n}{2} \rfloor$–shift of the entries: Introduce one extra column at the beginning of the array, then we shift the entries of the old array, $\lfloor \frac{n}{2} \rfloor$ steps into the new array. So, we have $\lceil \frac{n}{2} \rceil$ number of dummy entries in the first column and $\lfloor \frac{n}{2} \rfloor$ number of dummy entries in the last column of the new array.
7. Sort all the columns downwards, independently.
8. Perform the inverse operation of Step 6.

The resulting array is a sorted column major indexing $m \times n$ rectangular array. We now describe the above column sort algorithm following our paradigm. First we define two different types of decomposition chain then we state the modified column sort algorithm:

Definition 3.4

1. *The* **left-right(n)–Chains**: *Assume that n is an even number, then the left-right(n)–Chains are n subvectors of V, where the i^{th} left-right(n)–Chain ($i = 0, 1, \ldots, n-1$) has v_{0i} as its first element, $v_{1(n-i-1)}$ as the second element, v_{2i} as its third element; then the process is continued in this alternating fashion, until the subvectors reach the last row (the $(m-1)^{th}$ row). There are m elements in each left-right(n)–chain. For example:*

$$
\begin{array}{c}
(v_{00}, v_{1(n-1)}, v_{20}, v_{3(n-1)}, \ldots) \\
(v_{01}, v_{1(n-2)}, v_{21}, v_{3(n-2)}, \ldots) \\
\vdots \\
(v_{0i}, v_{1(n-i-1)}, v_{2i}, v_{3(n-i-1)}, \ldots) \\
\vdots \\
(v_{0(n-1)}, v_{10}, v_{2(n-1)}, v_{30}, \ldots).
\end{array}
$$

2. *The* **quotient(n)–Chains**: *This are the m subvectors of V, which contains those elements v_i whose indexes have the same value of $\lfloor \frac{i}{n} \rfloor$ (indices of elements of each subvector are in increasing order). There are n elements in each quotient(n)–chain.*

There are two points we want to make here: First, the *quotient*(n)–chains are, in fact, just the rows of an $(m \times n)$ grid when the row major indexing scheme is used. However, the *quotient*(m)–chains will be the columns of the $(m \times n)$ grid when the column major indexing scheme is used. See Figure 3.7 for some illustrations. Second, in Definition 3.4 we define the *left-right*(n)–chains for n even only, because the *left-right*(n)–chains will be complicated for n is odd, if the definition of the *left-right*(n)–chains is to satisfy the condition of m-g-chains defined in the next chapter. Since the *left-right*(n)–chains are used to demonstrate a point in the next chapter, we think it is justifiable that we only give a partial definition here.

Now, following our merge paradigm, Column sort can be rewritten as the following algorithm. This modified version is less complex than the original version. Furthermore, the modified version has a better time upper bound than the original version, which we will prove in Theorem 3.9.1.

The Modified Column Sort Algorithm

1. Sort all the *quotient*(m)–chains of V.
2. Sort all the m–chains of V.
3. Apply the N–m–cotransformer.

An in-depth analysis of this algorithm can be found in Leighton [56]. However, we want to re-examine

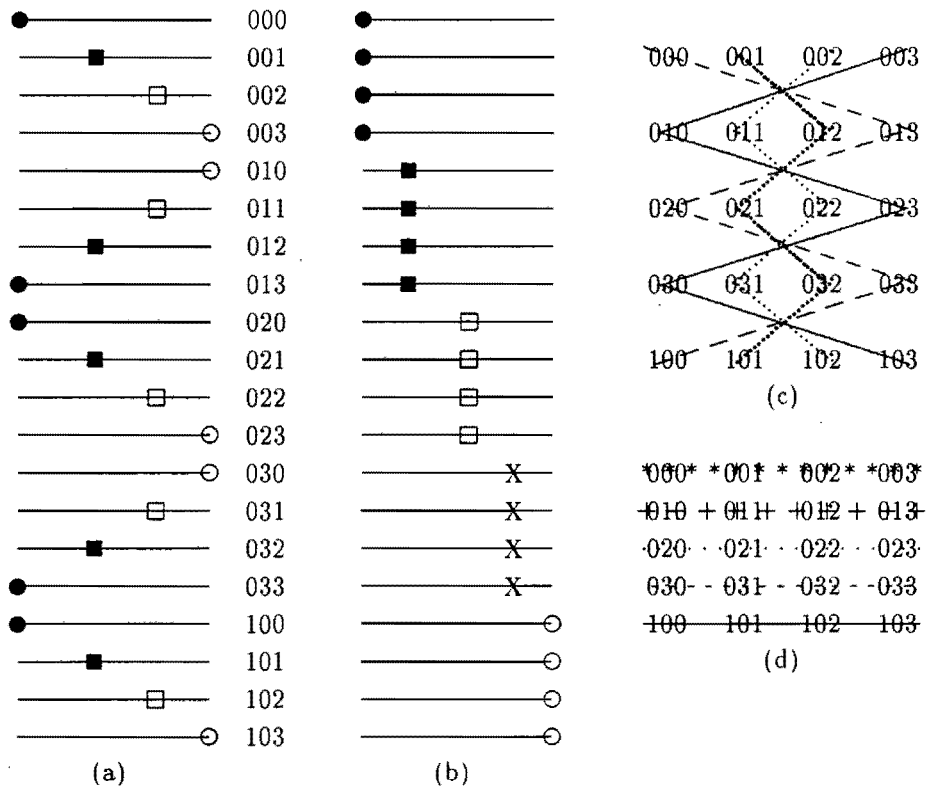


Figure 3.7: Some examples of (a) 4 *left-right*(4)-chains. (b) 5 *quotient*(4)-chains. (c) The 4 *left-right*(4)-chains in a 5×4 grid. (d) The 5 *quotient*(4)-chains in a 5×4 grid.

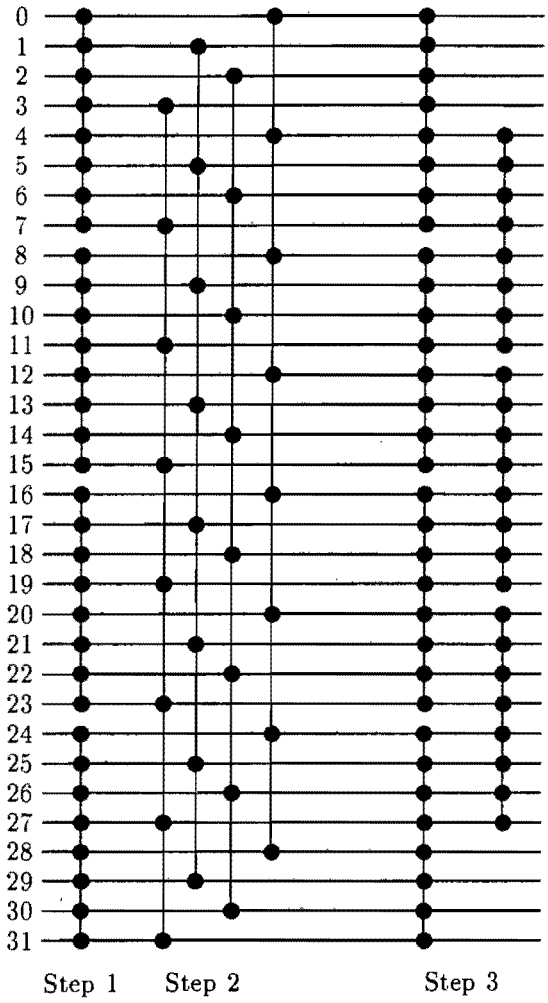


Figure 3.8: A line representation of the modified column sort, on a 32–vector.

this algorithm briefly using the row major indexing scheme. Using the row major indexing scheme, the algorithm can be described by the following three steps. Step 1: sort each block of size m. By "block" we mean, m successive lines in the line presentation, starting with line 0. A block corresponds to the first layer of m-comparators in the N-m-cotransformer. We will formally define a block in the following chapter. Step 2: sort the columns of V. Step 3: apply the N-m-cotransformer. See Figure 3.8 for an illustration of this algorithm.

The Diagonalizing Column Sort in Leighton [56] is not presented here, not because it can not be converted, but after converting the algorithm, we find that the required decomposition chains are complicated and not uniform (using the column major indexing scheme). However, the only difference between the Column sort and the Diagonalizing Column Sort is in Step 4, where the Diagonalizing algorithm uses the diagonalizing permutation instead of the transposition (from rows into columns) used in the Column sort. As a personal preference, we think it is much clearer to regard the process of sorting using the diagonalizing permutation on A as a process of sorting the skew blocks (sub-network) of A. We will introduce the skew blocks in the next chapter.

To summarize, under our merge paradigm using the row major indexing scheme, instead of the column major indexing scheme, on the $m \times n$ grids, the two algorithms by Leighton [56] are described by the steps: In Step 1 of the Column Sort, the algorithm sorts the blocks of size m (the Diagonalizing Column Sort sorts skew diagonal blocks respectively), then the algorithm sorts the m-chains, before the resulting array is input into the N-m-cotransformer. Parker and Parberry [75] introduce one such algorithm, where different size sorters are needed. The algorithm also uses the row major indexing scheme. We will introduce their algorithm in the next section.

3.3.2 The Skew Column Sort Algorithm

We will examine the modified column sort of Parker and Parberry [75] in this section. Since their algorithm needs to diagonalize the array, we will call the algorithm *the skew column sort*. As we have indicated in the introduction the skew column sort conforms to our framework of merge paradigm. The skew column sort and the series of sorting algorithms that we are going to introduce are similar in the following ways: firstly, the skew column sort uses the row major indexing scheme on rectangular interconnection networks of size m by n, the same as most of the algorithms introduce here. Secondly, the algorithm also applies a sub-network that is similar to the cotransformer, although their sub-network uses different size comparators. The skew column sort is the first algorithm we introduce, where the procedure needs to rewire the connections. We regard the operation of diagonalizing a interconnection network as rewiring the connections of the network. We will illustrate this point in the line representation of this algorithm.

Since the skew column sort algorithm conforms to the framework of our merge paradigm, there is no

need to present a modified version of this algorithm, like previous cases. We will introduce the original algorithm in the grid format and we will illustrate the line format by the example in Figure 3.9.

We introduce the skew column sort from Parker and Parberry [75], which is implemented using the mesh style representation. We will further interpret this algorithm using the line representation, as illustrated in Figure 3.9. We will not introduce different types of decomposition chains. It will be clear from Figure 3.9, that this is not the usual line representation scheme used before, as the algorithm needs to diagonalize the interconnection network. We demonstrate this by rewiring the connections of the network at Step 3.

Before we introduce the skew column sort algorithm in the grid representation, we will describe the diagonalization procedure [75] of the elements of an $m \times n$ mesh, with row major indexing: Each element at position (i, j) is moved to position $(i + j, j)$ simultaneously. The mesh expands downwards by n rows. The positions vacated in the upper right-hand corner (there are $\frac{1}{2}n(n-1)$ such positions) are filled up by "small" values (such as $-\infty$ or 0). The positions vacated in the lower left-hand corner (there are $\frac{1}{2}n(n-1)$ such positions plus one extra row) are filled up by "large" values (such as ∞ or 1).

The Skew Column Sort Algorithm

1. Partition the mesh into $\frac{m}{n}$ by n rectangles and sort each into row major order.
2. Sort all the columns downwards, independently.
3. Diagonalize the mesh as described above.
4. Partition the mesh into n by n squares and sort each into row major order.
5. Merge each half square with its mate in the adjacent square, that is, merge the last $\frac{1}{2}(n^2-3n+2)$ values of square i with the first $\frac{1}{2}(n^2-3n+2)$ values of square $i+1$, $0 \leq i < \frac{m}{n}$.

This algorithm is very similar in style to the algorithms that are implemented in the next chapter. However, this algorithm still sorts the blocks and it uses rewiring at Step 3 to get the desired connections. Furthermore, it uses different size comparators: In Step 1 and Step 2, it uses m-comparators. In Step 4, it uses n^2-comparators, and in Step 5, it uses (n^2-3n+2)-comparators (although, the algorithm can use n^2-comparators here also). In Figure 3.9, we illustrated the line representation of the skew column sort, using a network of size 32 to demonstrate this.

Next, Parker and Parberry [75] use the skew column sort to construct the merge algorithm. First they make the following assumptions: $N = n^p$, $m = n^{p-1}$, n is a integer square and p is an integer:

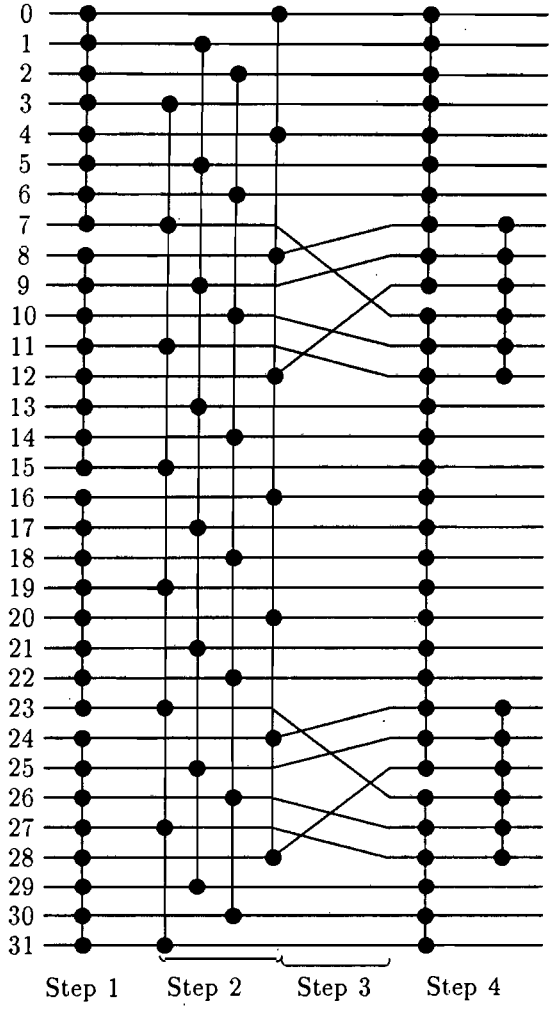


Figure 3.9: In Step 1, we sort all the *quotient*(m)-chains of V. In Step 2, we sort all the m-chains of V. In Step 3, we rewire the connections, hence rearrange the elements of V. In Step 4, we apply a different type of the cotransformer (we could have used the N-n^2-cotransformer here also).

The Construction of an (n, m)–Merger

1. Lay out n sequences of m sorted values in row major order.
2. Recursively merge each column using $(n, \frac{m}{n})$–Mergers.
3. Diagonalize the mesh as described above.
4. Use one layer of n^2–comparators to sort the n by n squares into row major order.
5. Use one layer of $(2, \frac{n^2}{2})$–Mergers to merge each half square with its mate in the adjacent square.

This merger algorithm is shown to use $4\log_{n^2} N - 4\log_{n^2} N + 1$ layers (which is equal to $(\log_n(N)-1)^2$) of n^2–comparators in Parker and Parberry's paper [75].

3.4 Generalization of the Bitonic Algorithm and the Odd-Even Merge Algorithm

In this section, we will present the generalized bitonic algorithm of Nakatani et al [64] and the generalized odd-even algorithm of Liszka and Batcher [62], in a format suitable for our merge paradigm. Basically, we can regard these two algorithms as sorting using k–comparators, where two sorted $\frac{N}{2}$–vectors are merged to form a sorted N–vector. Then we will present a further generalized version of the Batcher's two merge algorithms, where $r \geq 2$ sorted $\frac{N}{r}$–vectors are merged by layers of k–comparators.

We will be following the terminology used in Nakatani et al [64] and Liszka and Batcher [62] in naming an algorithm. Hence q–way r–merge will describe the following: the term "q–way" is used to describe the number of columns in the grid A (derived from V, see Figure 3.1). The term "r–merge" implies we are merging r sorted subvectors in parallel. In contrast, the modulo merge sorting algorithm of Liszka and Batcher [62] uses the term "module of size q" which is equivalent to q–way.

To avoid confusion, we will start with an outline of this section: Every algorithm in this section is presented in two different formulations under the dual representation introduced in Chapter 2. To start with, we introduce four different q–way r–merge algorithms, using the grid representation. The grid representation describes how an algorithm proceeds in sorting numbers arranged in a rectangular grid. Such an algorithm will be called a grid algorithm.

Grid algorithms, in general, have to sort subparts of the grid that are obvious to visualize (such

as columns, rows or consecutive subgrids of a certain size). Under the grid representation, no implicit instruction is given of how those subparts are sorted. However, a grid algorithm does provide an excellent mental visualization of the algorithm, and it gives a clear conception of the sorting algorithm. We will prove the correctness of the grid algorithms, after they have been introduced in this subsection.

After we have finished dealing with the grid algorithms, we will introduce a class of generalized merge algorithms, which is a recursive implementation of these four different grid algorithms, which can be easily implemented by sorting networks using k–comparators. This class of merge algorithms is constructed from the multi-way multi-merge procedure, which uses the grid algorithms as the underlying operations. It can be shown that the correctness of the merge algorithm follows immediately from the correctness of the underlying grid algorithm.

From the construction of the k–comparator network, one is able to calculate the time complexity of the corresponding merge algorithm (depending on which underlying grid algorithm is used). In the implementation of the merge algorithm by a network of k–comparators, the obvious aim is to minimize the following criteria: the depth of the network, the size of the comparator used and the total number of comparators used. For every merge algorithm presented in this chapter, we will let $T_k(N)$ denote the depth of the merger of size N using k–comparators and let $C_k(N)$ denote the number of k–comparators of the merge. We want to emphasize once again that our merger networks use the same size k–comparators only.

3.4.1 The Grid Representation of Four Different Merge Algorithms

Four different merge algorithms are introduced here, where two of these four merge algorithms belong to the family of q–way 2–merge algorithms and the other two merge algorithms belong to the family of q–way r–merge algorithms ($r > 2$). The case of $r = 2$ has network representation of depth one fewer than the case $r > 2$. However, all four algorithms are generalizations of Batcher's merge algorithms: the bitonic merge algorithm and the odd-even merge algorithm. Every algorithm will be introduced as a series of elementary operations on a rectangular $p \times q$ grid and we will prove the correctness of each algorithm.

A Generalization of Batcher's Merge Algorithms

We start with the family of 2–merge algorithms. In particular, we investigate the q–way bitonic merge algorithm and the modulo merge algorithm. Nakatani et al [64] introduce the q–way bitonic merge, which is a generalization of Batcher's bitonic merging network. This algorithm is based on the multi-way decomposition of a bitonic vector, instead of the 2–way decomposition of Batcher's bitonic merge. We will present a further generalization of Batcher's odd-even merge in a later section, where we will discuss

how this method is extended to $r > 2$ sorted input vectors.

The q–way bitonic merge algorithm merges two sorted sequences of length $\frac{N}{2}$. For simplicity, we assume that N and q are even integers.

The q–Way 2–Merge Bitonic Algorithm (Grid version)

1 Place elements of V into a $p \times q$ matrix A (using the row major indexing scheme).

2. Sort the columns of A.

3. Sort the rows of A.

We shall leave the correctness proof of the q–way bitonic merge till after we have introduced the next algorithm. We prove the correctness of both algorithms together, since these two algorithms' operations are executed in a similar style under our merge paradigm.

Liszka and Batcher [62] have extended the widely used odd-even merge into a modulo merge sorting network. The modulo merge is a generalization of Batcher's odd-even merge. We will present a further generalization of Batcher's odd-even merge later, by merging $r > 2$ sorted input vectors.

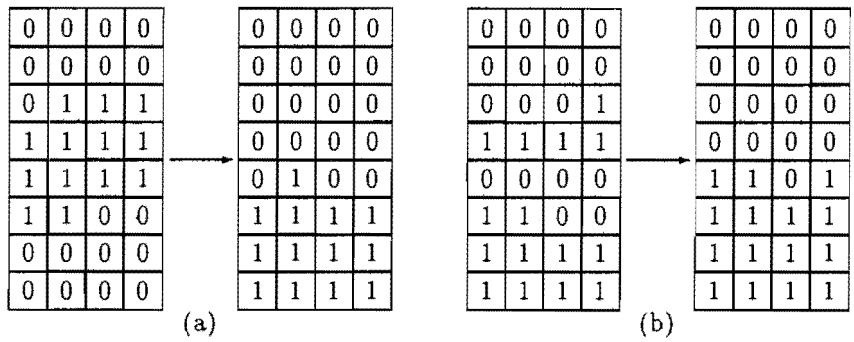


Figure 3.10: Example of a bitonic array and an odd-even array placed into an 8×4 grid respectively, then followed by a column sort.

We will look at the modulo merge network of size N, with q as the selected parameter for the merge algorithm. This network can be regarded as a q–comparators network, with rewiring procedures and special switches. The method described by Liszka and Batcher [62] in terms of generalized comparators is complicated (for a detailed discussion, see [62]), but it is fairly simple to describe its operation on a grid, which we will do below. We will describe a variant of this algorithm using simple generalized

comparators in the later section. Before we describe its workings, we introduce new terminology in the next definition:

Definition 3.5 *Given any vector V of real numbers that is a concatenation of 2 sorted sequences of equal length, then*

1 *The vector V is* **regular-bitonic**, *if it is a concatenation of two equal length sequences, where the first sorted subvector of V is in increasing order, the second sorted subvector of V is in decreasing order. (This just a bitonic vector V with two equal size subvectors, see Definition 3.3.)*

2 *The vector V is* **bi-increasing** *(i.e.* **2–increasing***), if it is a concatenation of two equal length sequences, where both subvectors are monotonically increasing.*

The modulo merge algorithm merges two sorted sequences of length $\frac{N}{2}$. Again for simplicity, we assume that N and q are even integers.

The q–Way 2–Merge Modulo Algorithm (Grid version)

1 Place elements of V into a $p \times q$ matrix A (using row major indexing scheme).

2 Reverse the last $\frac{p}{2}$ rows of A.

3 Sort the columns of A.

4 Sort the rows of A.

Lemma 3.4.1 *If the input vector is either regular-bitonic or bi-increasing, then after the columns are sorted, in either the q–way 2–merge bitonic algorithm or the q–way 2–merge modulo algorithm respectively, then :*

1 *there is at most one dirty row in the resulting grid.*

2 *the rows of the resulting grid are bitonic.*

Proof: It is sufficient to show the lemma is true for any 0–1 vector V, where V is a concatenation of a pair of increasing sequences or a pair of an increasing sequence and a decreasing sequence. Now we place V into the grid A when V is a regular-bitonic array, else if V is a bi-increasing array, we then place V into the grid A with the last $\frac{p}{2}$ rows reversed. Then A will have at most two dirty rows, and in this case, the first increasing (starts with 0) and the second decreasing (starts with 1).

When we sort the columns of A, the all-1 rows will sink to the bottom of the grid and the all-0 rows will rise to the top of the grid (See Figure 3.10 for the illustration). Sorting the two dirty rows, one increasing and one decreasing, results in at least one clean row. So at the end, there is at most one dirty row (See Figure 3.11 for the illustration). From the Figure 3.11, it is clear that the dirty row is bitonic. So all rows of the resulting grid are bitonic. ∎

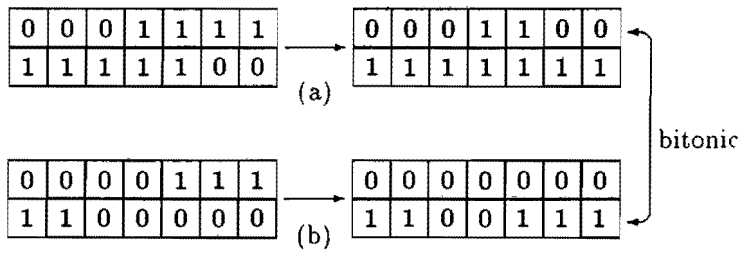


Figure 3.11: An illustration of the two possible cases of how the pair of dirty rows is combined.

Theorem 3.4.2 *If the input vector is either regular-bitonic or bi-increasing, then the q-way 2-merge bitonic algorithm and the q-way 2-merge modulo algorithm, respectively, will result in a grid A sorted in row major order and the corresponding vector V will be sorted.*

Proof: The correctness follows immediately from Lemma 3.4.1, since step 3 (step 4) sorts each row and there is only one dirty row after step 2 (step 3) in case of the q-way 2-merge bitonic algorithm (the q-way 2-merge modulo algorithm) respectively. ∎

It is quite obvious that any row of size q in A can be regarded as a block of q successive lines in the line representation (see Definition 3.2). Hence it is not difficult for one to see that if one uses one of these two q-way 2-merge grid algorithms to merge a pair of $\frac{N}{2}$-vectors, then the row sort at the end of each grid algorithm can be implemented by an N-q-unitransformer, where each row is sorted by a q-comparator.

A Further Generalization of Batcher's Merge Algorithms

Two new algorithms from the family of r-merge algorithms are introduced here. These two algorithms generalize the bitonic merge and the odd-even merge of two vectors into mergers of r vectors of equal size, respectively. We will give a brief description of the further generalized algorithms, with the following grid parameters: Let $N = pq$, and we will require that r is divisible by q, and p is divisible by $\frac{r}{q}$. In the following two algorithms, we let $k = k^*q$, where $k^* = r - 2$ if r is even and $k^* = r - 1$ if r is odd. Again, we are using the row major indexing scheme. We give a description of the workings of the algorithm for merging r sorted sequences of each length $\frac{N}{r}$. The sorted sequences are placed in a particular manner into

vector V, which in terms is placed into a $p \times q$ rectangular grid. We also assume that $\frac{N}{r} \geq q$ throughout.

Definition 3.6 *Given any vector V of real numbers that is a concatenation of $r \geq 2$ sorted sequences of equal length, then*

1 *The vector V is* **regular-r–tonic**, *if the first sorted subvector of V is in increasing order, the second sorted subvector of V is in decreasing order and the pattern is repeated for the remaining $r-2$ sorted subvectors.*

2 *The vector V is* **r–increasing**, *if the sorted subvectors of V are all in monotonically increasing order.*

For simplicity, we suppose that r is even in the next two algorithms:

The q–Way r–Merge r–tonic Algorithm (Grid version)

1. Place elements of V into a $p \times q$ matrix A (using row major indexing scheme).
2. Sort the columns of the corresponding $p \times q$ matrix A.
3. Sort the rows in groups of k^*, starting from Row_0.
4. Sort the rows in groups of k^*, starting from $\text{Row}_{\frac{k^*}{2}}$.

The q–Way r–Merge Modulo Algorithm (Grid version)

1. Place elements of V into a $p \times q$ matrix A (using row major indexing scheme).
2. Skip the first $\frac{p}{r}$ rows and reverse the next $\frac{p}{r}$ rows. We will repeat this process to each alternate subsequent $\frac{2p}{r}$ rows in the $p \times q$ matrix A, until there are less than $\frac{p}{r}$ rows that have not yet been processed.
3. Sort the columns of A.
4. Sort the rows in groups of k^*, starting from Row_0.
5. Sort the rows in groups of k^*, starting from $\text{Row}_{\frac{k^*}{2}}$.

Before we prove the correctness of these two q–way r–merge grid algorithms, we will make a similar observation as the one we made for the q–way 2–merge grid algorithms on page 46, which demonstrates how the last two steps of any one of these two q–way r–merge grid algorithms can be implemented on a k–comparator network: the last two steps of a q–way r–merge grid algorithm of size N can be implemented by the N–k^*q–cotransformer, where the first layer of the k–comparators and the second layer of the k–comparators of the N–k–cotransformer are corresponding to the second last step and the last step of the grid algorithm respectively. Now we shall prove the correctness of both algorithms together, since those two r–merge algorithms' operations are executed in a similar style under our merge paradigm.

Theorem 3.4.3 *If the input grid A is either regular-r-tonic or r-increasing, then after the columns of the resulting A are sorted in the respective algorithm, there will be at most $\lceil \frac{r}{2} \rceil$ successive dirty rows in the grid.*

Proof: For the case when $r = 2$ see Lemma 3.4.1. Hence it is true for $r = 2$.

For the case when $r > 2$, there will be at most r dirty rows in A. Since $\frac{N}{r} \geq q$, all the positions where two of the subvectors join lie on different rows. In the case of both algorithms, the dirty rows will therefore be alternately increasing and decreasing. Sort the columns by first moving the all–1 rows to the bottom of the grid and the all–0 rows to the top of the grid. There will then be at most r adjacent dirty rows. Now sort the r dirty rows in pairs, leaving the final dirty row untouched if r is odd. Since the rows are alternative increasing and decreasing, it follows as in Figure 3.11 there are at most $\lceil \frac{r}{2} \rceil$ dirty rows after these columns are sorted. Now move the newly created all–1 rows and all–0 rows to the top and bottom of the grid respectively, leaving at most $\lceil \frac{r}{2} \rceil$ successive dirty rows in all. ∎

Theorem 3.4.4 *If the input vector is either regular-r-tonic or r-increasing, then the q-way r-merge r-tonic algorithm and the q-way r-merge modulo algorithm, respectively, will result in a grid A sorted in row major order and the corresponding vector V will be sorted.*

Proof: From Theorem 3.4.3, we know the resulting grid A will have at most $\lceil \frac{r}{2} \rceil$ adjacent dirty rows. We claim that those $\lceil \frac{r}{2} \rceil$ dirty rows are contained in a group of k^* rows described in the last two steps of the q–way r–merge grid algorithms. For simplicity, we shall label the way we grouping k^* rows of A in the second last step of the algorithm as the first decomposition of the grid and in the last step of the algorithm as the second decomposition of the grid. If this claim is true, then the resulting grid will definitely be sorted. We prove this for the case where r is an even number and $k^* = (r - 2)$. A similar argument applies to the case when r is odd.

Suppose those successive dirty rows (at most $\frac{r}{2}$) are not contained in a group of k^* rows from the first decomposition. Since the first decomposition partition the grid A into blocks of k^* rows where $k^* > \lceil \frac{r}{2} \rceil$,

hence $\lceil \frac{r}{2} \rceil$ successive rows will contain in at most two groups of k^* rows. Thus there exists a group of k^* rows from the first decomposition that contains at least half of the dirty rows - in the extreme case where we have $\lceil \frac{r}{2} \rceil$ adjacent dirty rows, there are at least $\frac{r}{4}$ rows. Now those $\frac{r}{2}$ dirty rows are not able to extend by less than 1 row outside this group of k^* rows, so there are maximum of $\frac{r}{2} - 1$ rows in this group of k^* rows.

Any group of k^* rows in the first decomposition will overlap two groups of k^* rows in the second decomposition (except the first group and the last group), by $\frac{r}{2} - 1$ rows of A. Hence those $\frac{r}{2}$ dirty rows are contained in a group of k^* rows in the second decomposition. The claim is proved and hence the theorem is also proved. ∎

One further observation: For this family of r–mergers, the last two steps of each algorithm can be replaced by three steps, where we sort groups of rows that have $\frac{r}{2}$ rows in each group. The correctness when using these three steps can be found in the next chapter, under the bichain sorting algorithm. Assume that r is even then three steps can be described as follows:

i. Sort the rows in groups of $\frac{r}{2}$, starting from Row_0.

ii. Sort the rows in groups of $\frac{r}{2}$, starting from $\text{Row}_{\frac{r}{2}}$.

iii. Sort the rows in groups of $\frac{r}{2}$, starting from Row_0.

These three steps can be implemented by the N–$\frac{rq}{2}$–bitransformer given in Definition 3.2.

3.4.2 The Class of Merge Algorithms

We have dealt with grid algorithms up to the present. We will now use the grid algorithms to construct merging and sorting networks using k–comparators. These new merging networks will produce either a class of 2–merge networks or a class of r–merge networks ($r > 2$), depending on the number of vectors that are been merged at each recursion. We need to express the algorithms in terms of what happens to the original vector v, rather than what happens when it is arranged in a grid. We therefore introduce the following functions with vector inputs and outputs, which emulates the executions of the grid algorithms. We start with some of the general procedures, which will be common operations in the construction of the class of merge algorithms. In what follows r represents the number of vectors to be merged. Furthermore, we use the variable **m-g-chains** to denote either the m–chains, the m–bichains or any other type of multiple generalized chains (see Definition 4.2):

- **Divide(In:** v, N, q, **m-g-chains; Out:** $\{c_i : i = 0, 1, \ldots, q-1\}$): This procedure takes as input an N–vector v and gives as output its q $\frac{N}{q}$–**m-g-chains** of the type indicated by variable **m-g-chains**, where c_0 is the m-g-chain containing the first component of v.

- **Unite(In:** $\{c_i : i = 0.1\ldots,q-1\}$, $\frac{N}{q}$, **m-g-chains; Out:** v): The procedure performs the inverse operation to **Divide**. It takes as input q ($\frac{N}{q}$)–vectors, and combines them using the type indicated by variable **m-g-chains**, and outputs v, an N–vector. c_0 will be the m-g-chain of v which contains the first component of v.

- **Transformer(In:** v, N, k, q, **m-g-chain; Out:** w): This procedure applies to input v the N–k–transformer associated with ($\frac{N}{q}$)–m-g-chains of the type indicated by variable **m-g-chain** and outputs the resulting N–vector, w. It is used for the case $r > 2$.

- **Multi-way Multi-merge(In:** v, N, r, q, b; **Out:** w): This is an r–merger that takes as input an N–vector v and outputs an N–vector w.

Remark: this procedure takes a Boolean variable b and if $b = 1$ then the procedure applies the permutation routing to the elements of v (see the permutation procedure **ModuloPerm** below), else it skips the permutation routing. Now, if v is either an r–increasing N–vector with $b = 1$ or an r–tonic N–vector with $b = 0$, then this procedure will output a sorted w (see Theorem 3.4.5). Note that the Multi-way Multi-merge(v, N, r, q, b) corresponds to the q–Way r–merge grid algorithms.

For the case $r = 2$, we use the following (**Unitransformer**) procedure in place of the **Transformer**. The network corresponding to the Unitransformer has lower depth than the network corresponding to the Transformer; the network corresponding to the Transformer will use at least 2 layers of k–comparators, but the network corresponding to the Unitransformer uses 1 layer of k–comparators. In particular, as we have stated at the end of the grid algorithms subsection, the Transformer procedure in this section will need to be implemented by the N–k–cotransformer (since the q–way r–merge grid algorithms are the underlying merge algorithms) and the Unitransformer procedure will be implemented by the N–k–unitransformer (since the q–way 2–merge grid algorithms are the underlying merge algorithms). Here is the procedure:

- **Unitransformer(In:** v, N, k, q, **m-g-chain; Out:** w): This procedure applies to input v one layer of k–comparators that associates with ($\frac{N}{q}$)–m-g-chain of the type indicated by variable **m-g-chain** and outputs the resulting N–vector, w.

Furthermore, the procedure for the q–Way r–Merge r–tonic grid algorithm is different from that of the q–Way r–Merge Modulo grid algorithm, where in the latter algorithm, a permutation of elements of the input vector is needed. Here is the permutation procedure:

- **ModuloPerm(In:** v, N, k, q; **Out:** w): This procedure applies to input v and outputs the N–vector w that results from permuting elements of v in the following manner: Suppose for each element v_i in v, there is an ordered pair (x, y) associated with v_i, where $x = \lfloor \frac{ri}{N} \rfloor$ and y is the q–ary representation of the index i. Now if x is even then let $w_i = v_i$. Else let $w_{i'} = v_i$, where i and i' differ only in the least significant bit - if we denote the least significant bits of i and i' by

q_0^i and $q_0^{i'}$ respectively, then $q_0^{i'} = q - 1 - q_0^i$.

Note: If a given vector v is r–increasing, then applying the ModuloPerm procedure to v is an emulation of the execution of the modulo grid algorithm on the matrix A: The elements of the first subvector are placed in A using row major indexing, and the elements of the second subvector are placed in A using reverse row-major indexing. Repeat this process for the subsequent subvectors in this manner, where alternating schemes are used to place elements in A. See page 18 for more details of various indexing schemes.

Next we will state the merge algorithm formally and formulate it in the most general form. Given r sorted $\frac{N}{r}$–subvectors, such that $N = pq = rq^s$ for some integer s. Then the merge algorithm will output a sorted N–vector. This merge algorithm can be implemented by a k–comparator network, where $k = k^*q$ for some k^*. (We will show after the merge algorithm, that the value of k^* depends on value of r: if $r = 2$ then $k^* = 1$, else if r is even then $k^* = r-2$, else when r is odd, $k^* = r-1$.) The algorithm goes as follows:

Procedure Multi-way Multi-merge(In: v, N, r, q, b; **Out:** w)

If size of the vector is less or equal k **then** apply a k–comparator;
Else

Begin

1 **If** $b = 1$ **then ModuloPerm**(v, N, k, q; w) **else** $w \leftarrow v$;

2 **Divide**(w, N, q, p–**chains**; $\{c_i : i = 0, 1, \ldots, q-1\}$);

3 **multi-way multi-merge**(c_0, p, r, q, b; z_0), ..., **multi-way multi-merge**(c_{q-1}, p, r, q, b; z_{q-1});

4 **Unite**($\{z_i : i = 0, 1, \ldots, q-1\}$, p, p–**chains**; w');

5 **If** $r = 2$ **then**

5.1 **Unitransformer**(w', N, k, q, p–**chains**; w)

Else

5.2 **Transformer**(w', N, k, q, p–**chains**; w)

End

In the above procedure, if the multi-way multi-merge algorithm takes the initial input value assigned to the parameters r and b, then the same values of r and b respectively are used though-out the recursion.

For example, if the procedure starts with $r = 2$ and $b = 0$ (a q–way 2–merge modulo grid algorithm), then the inner recursion of the procedure will continue using the q–way 2–merge modulo grid algorithm also. However, the merge algorithm can be implemented in a different manner, such that an r–merger ($r > 2$) can be constructed from r'–mergers where $r' < r$. With a few modifications, the above algorithm will be able to implement mixed combinations of r–mergers with different values of r at any iteration. For simplicity, we will leave the algorithm as it is stated above. However, we will discuss the issue of combining mergers from different families in the later section of this chapter, after the sorting algorithm is introduced. We illustrate some merge networks derived from the multi-way multi-merge procedure in the following figures (Figure 3.12, Figure 3.13 and Figure 3.14).

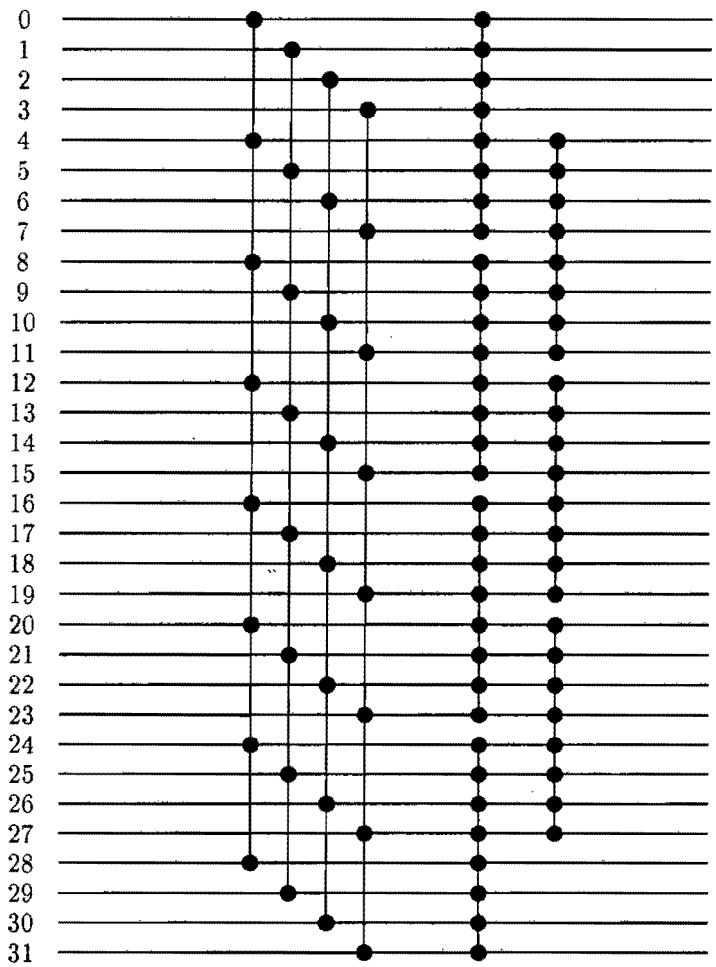


Figure 3.12: An example of the line representation of a network of size 32 constructed from the multi-way multi-merge procedure, where the underlying grid algorithm used is the 8–way 4–merge 4–tonic grid algorithm.

Next, we will prove the correctness of the multi-way multi-merge algorithm. This is a recursive algorithm. To implement a recursive algorithm correctly, we will need to show that the algorithm is

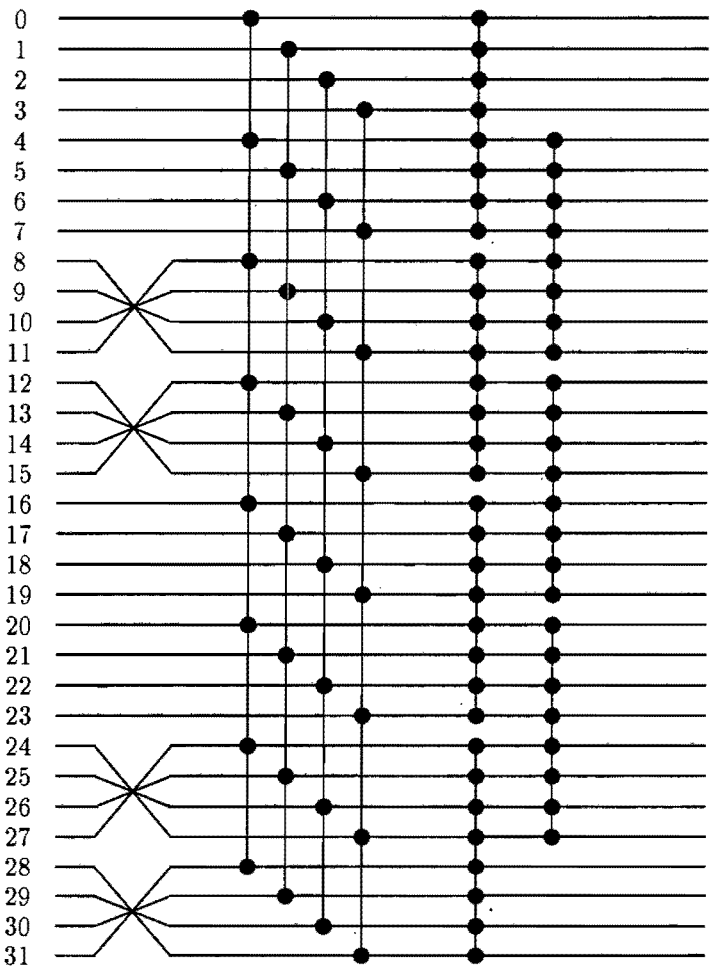


Figure 3.13: An example of the line representation of a network of size 32 constructed from the multi-way multi-merge procedure, where the underlying grid algorithm used is the 8–way 4–merge modulo grid algorithm.

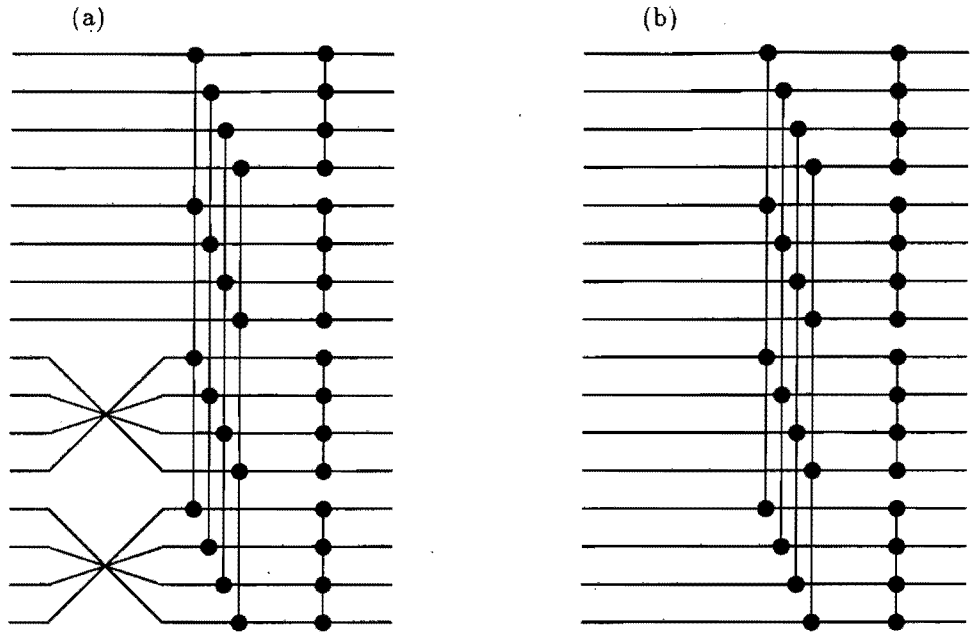


Figure 3.14: Here are two examples of the line representation of networks constructed from the multi-way multi-merge procedure (following different underlying grid algorithm): (a) A 4–comparator network that used the 4–way 2–merge modulo grid algorithm as the underlying merge algorithm. (b) A 4–comparator network that used the 4–way 2–merge bitonic grid algorithm as the underlying merge algorithm.

recursively well-defined. Here are some observations that will assist in the proof of the correctness theorem:

1 Step 1 of the multi-way multi-merge algorithm is equivalent to applying either the first two steps of the q–way r–merge modulo grid algorithm or the first step of the q–way r–merge r–tonic grid algorithm, depending on b is equal to 1 or 0 respectively.

2 Step 2 to Step 4 of the multi-way multi-merge algorithm will produce a vector that is equivalent to placing V into the grid A, after the column sort of the underlying grid algorithm is applied. See Figure 3.15 for an illustration.

3 Step 5.1 of the multi-way multi-merge algorithm is equivalent to the last step of the q–way 2–merge grid algorithm.

4 Step 5.2 of the multi-way multi-merge algorithm is equivalent to the last two steps of the q–way r–merge grid algorithm ($r > 2$).

Theorem 3.4.5 : (The Correctness Theorem) *The multi-way multi-merge procedure sorts any input N-vector V that is either a regular-r-tonic vector or an r-increasing vector.*

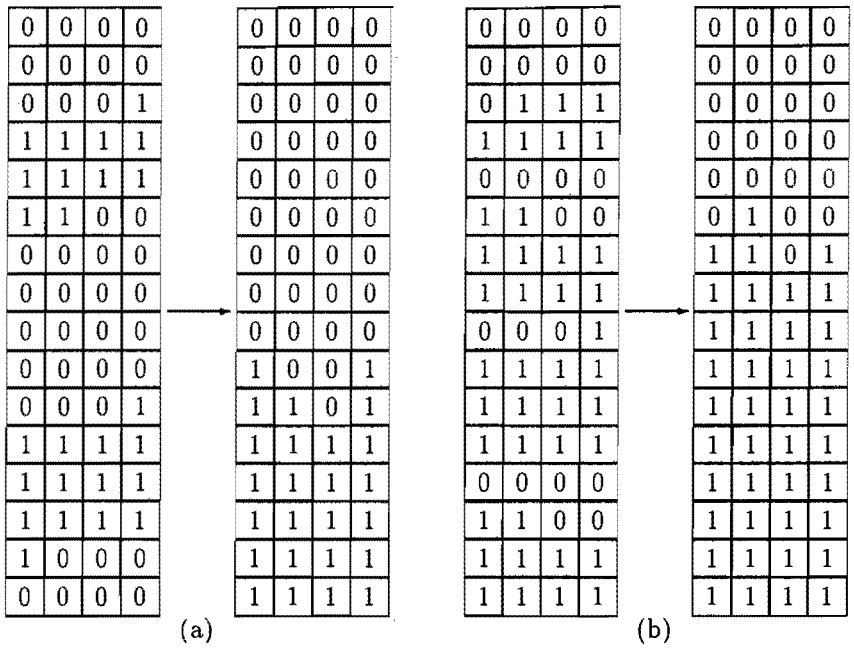


Figure 3.15: The grid representation of the 4–way 4–mergers on a 16×4 grid.

Proof: The multi-way multi-merge procedure clearly sorts, if the length of the input vector is less than k.

Assume that the procedure sorts any input vector of length less than $\frac{N}{q}$ which is either regular-r–tonic or r–increasing. Then it follows from the observations made before the start of this proof that it is sufficient to show that if the input vector V is either regular-r–tonic or r–increasing, then the columns of A are also regular-r–tonic or r–increasing respectively, where A is derived from placing elements of V (using row major indexing scheme) into the $p \times q$ grid A, in the manner described by the steps of the underlying grid algorithm. We will prove that the columns of A are also regular-r–tonic (r–increasing respectively) when the input vector is regular-r–tonic (r–increasing respectively) in Lemma 3.4.6 (Lemma 3.4.7 respectively). Hence by inductive hypothesis, the procedure correctly sorts each column of A (each column has length equal to $\frac{N}{q}$).

Then it follows immediately from Theorem 3.4.3 that there exists an N–k–cotransformer which sorts the resulting N–vector. By the correctness of the underlying grid algorithms (Theorem 3.4.4), we can conclude that the resulting vector is sorted. Hence we have proven the correctness of the multi-way multi-merge algorithm inductively. ∎

Next we will show, in terms of the operation described in the underlying grid algorithm, that if the input N–vector is either regular-r–tonic or r–increasing, then the columns of A are also regular-r–tonic or r–increasing respectively.

Lemma 3.4.6 *If the input vector is regular-r-tonic, then after Step 1 of the q-way r-merge r-tonic grid algorithm, the column vectors of A are regular-r-tonic.*

Proof: We shall regard a regular-r-tonic vector of size N as a concatenation of r equal size subvectors.

If $r = 2$, then after Step 1 of the q-way 2-merge bitonic grid algorithm, the columns of A are regular-bitonic. One just needs to make the following observation: the 1^{st} sorted subvector is placed in the upper half of V and the 2^{nd} subvector is placed in the lower half of V. It follows immediately, when the 1^{st} sorted subvector is transferred to the upper half of A, the columns of A will each contains increasing ordered subvector in its upper half.

Next, in the case of the q-way 2-merge bitonic grid algorithm, the opposite will be true for the lower half of A, where each column will contain a decreasing ordered subvector in its lower half. Hence the columns of A after Step 1 are regular-bitonic.

Clearly adding a third subvector preserves the condition that the column is regular-r-tonic by similar reasoning. We can continue with this procedure until the subvector is exhausted. ∎

Lemma 3.4.7 *If the input vector is r-increasing, then after Step 2 of the q-way r-merge modulo grid algorithm, the column vectors of A are r-increasing.*

Proof: If there are two sorted input vectors are sorted, then after Step 2 of the q-way 2-merge modulo grid algorithm, the columns of A are bi-increasing. An argument similar to the proof of Lemma 3.4.6 can be used, but in the case of the q-way 2-merge modulo grid algorithm, the one given for the upper half of A can be applied to the lower half of A also. Reversing each row in the lower half of A does not effect the fact that the subvector in the lower half of each column is sorted in increasing ordered. Hence the columns of A after Step 2 are bi-increasing.

Clearly adding a third subvector preserves the condition that the column is r-increasing by similar reasoning. We can continue with this procedure until the subvector is exhausted. ∎

Remark: We know that if $r = 2$ the multi-way multi-merge algorithm will need one layer of k-comparators, where $k = q$, so that size of comparator is the size of the row of A. If $r > 2$, then we need the N-(k^*q)-cotransformer, where $k^* = r - 2$ if r is even and $k^* = r - 1$ if r is odd (see Theorem 3.4.4). This has two layers of k-comparators.

3.4.3 Time Complexity of the Multi-way Multi-merge Network

In this section, we will calculate the time complexities of various merge networks derived from the multi-way multi-merge procedure. Henceforth, we will call the network base on the multi-way multi-merge

algorithm with parameters q and r, **the q–way r–merge network**. We will refer to the collection of q–way r–merge networks as **the multi-way multi-merge networks**.

We will be comparing the time complexity of various networks in this section. The time complexity is expressed in terms of N, q and k, where $k \geq k^* q$ and k^* depends on r. Recalled that the term "q–way" is used to describe the number of columns in the grid A that derived from the input vector. The term "r–merge" implies we are merging r sorted subvectors in parallel. When we apply the multi-way multi-merge algorithm, we can restrict the size of comparators to be of equal size k in each complexity calculated, hence we will try and express the time complexity in terms of k. For any k–comparator network, we let $T_k(N)$ denote the depth of the merger of size N and $C_k(N)$ denote the number of k–comparators used by the merger.

Time Complexity of the q–Way 2–Merge Network

We will start with the time complexity of the multi-way multi-merge procedure using the parameters $q = k$ and $r = 2$. We will calculate the depth of the k–comparator network from this family of the q–way r–merge networks. Furthermore, we will also calculate the number of k–comparators used in such a network:

$$\begin{aligned} T_k(N) &= T_k(\frac{N}{q}) + 1, \ T_k(k) = 1 \\ C_k(N) &= qC_k(\frac{N}{q}) + (\frac{N}{k}), \ C_k(k) = 1. \end{aligned}$$

When $r = 2$ the multi-way multi-merge algorithm uses the unitransformer (one layer of k–comparators) to sort the rows of A. Hence we require $q = k$ when we apply the multi-way multi-merge algorithm, in the case $r = 2$. We have the following:

$$\begin{aligned} T_k(N) &= \lceil \log_q \frac{N}{k} \rceil + 1 \\ &= \lceil \log_q N \rceil \\ &= \lceil \log_k N \rceil \qquad (3.1) \\ C_k(N) &= qC_k(\frac{N}{q}) + \lceil \frac{N}{k} \rceil \\ &= qC_k(\frac{N}{q^2}) + q\lceil \frac{N}{qk} \rceil + \lceil \frac{N}{k} \rceil \\ &= \sum_{i=0}^{\lceil \log_q \frac{N}{k} \rceil} q^i \lceil \frac{N}{q^i k} \rceil \\ &= \sum_{i=0}^{\lceil \log_k \frac{N}{k} \rceil} k^i \lceil \frac{N}{k^{i+1}} \rceil \qquad (3.2) \end{aligned}$$

For example, if N is a power of k, then equation 3.1 and equation 3.2 reduces to the following:

$$T_k(N) = \log_k N \tag{3.3}$$

$$C_k(N) = \frac{N}{k}\log_k N \tag{3.4}$$

Time Complexity of the q–Way r–Merge Network ($r > 2$)

Next, we will investigate the time complexity of the multi-way multi-merge procedure using the parameters q and $r > 2$. We will calculate the time complexity of the q–way r–merge network, using the same notation used above. Here is the time complexity and the number of k–comparators used in the corresponding network:

$$T_k(N) = T_k(\frac{N}{q}) + 2,\ T_k(k) = 1$$

$$C_k(N) = qC_k(\frac{N}{q}) + (2\frac{N}{k} - 1),\ C_k(k) = 1.$$

so that

$$T_k(N) = 2\lceil \log_q \frac{N}{k} \rceil + 1 \tag{3.5}$$

$$\begin{aligned} C_k(N) &= qC_k(\frac{N}{q}) + (2\lceil\frac{N}{k}\rceil - 1) \\ &= q^2C_k(\frac{N}{q^2}) + q(2\lceil\frac{N}{qk}\rceil - 1) + (2\lceil\frac{N}{k}\rceil - 1) \\ &= \sum_{i=0}^{\lceil \log_q \frac{N}{k}\rceil} q^i(2\lceil\frac{N}{q^ik}\rceil - 1) \\ &= (\sum_{i=0}^{\lceil \log_q \frac{N}{k}\rceil} 2q^i\lceil\frac{N}{q^ik}\rceil) - \frac{q^{\lceil \log_q \frac{N}{k}\rceil + 1} - 1}{q-1} \end{aligned} \tag{3.6}$$

For example, to simplify the expressions, if $N = q^s k$ for some positive integer s, then equation 3.5 and equation 3.6 reduces to the following:

$$\begin{aligned} T_k(N) &= 2\log_q \frac{N}{k} + 1 \\ &= 2(\log_q k)(\log_k \frac{N}{k}) + 1 \\ &= 2\log_q N - O(\log_q k) \\ &= 2\log_q k \log_k N - O(\log_q k) \end{aligned} \tag{3.7}$$

$$\begin{aligned} C_k(N) &= (\sum_{i=0}^{\log_q \frac{N}{k}} 2\frac{N}{k}) - \frac{q\frac{N}{k} - 1}{q-1} \\ &= 2\frac{N}{k}(\log_q \frac{N}{k} + 1) - \frac{qN - k}{k(q-1)} \end{aligned}$$

$$
\begin{aligned}
&= 2\frac{N}{k}\log_q \frac{N}{k} + O(\frac{N}{k}) \\
&= 2\frac{N}{k}\log_q k \log_k \frac{N}{k} + O(\frac{N}{k}) \\
&= 2\frac{N}{k}\log_q k \log_k N + O(\frac{N}{k}\log_q k). \qquad (3.8)
\end{aligned}
$$

Time Complexity of r–merge Network Constructed From the q–way 2–merge Algorithms

We have calculated the time complexities of the q–way r–merge networks derived from the multi-way multi-merge procedure for $r \leq 2$. An alternative to the multi-way multi-merge networks is to use the q–way 2–merge to merge groups of 2 subvectors until the whole vector is sorted. This could prove to be more efficient, since the q–way 2–merge network uses one fewer layer of comparators than the q–way r–merge network ($r > 2$), at each recursion. It makes sense for us to compare the time complexity of the q–way r–merge network of size N to the time complexity of an r–merge network that uses the q–way 2–merge algorithms to merge r $\frac{N}{r}$–vectors (by merging only two vectors at a time). The next procedure does exactly this: it uses the 2–merge algorithms to merge r $\frac{N}{r}$–vectors.

Here we define a permutation procedure and the procedure that implements the q–way 2–merge algorithm to merge r sorted $\frac{N}{r}$–vectors (for simplicity, we assume r is a power of 2.):

- **Bi-append(In:** $\{c_i : i = 0, 1, \ldots, r-1\}$, $\frac{N}{r}$, b; **Out:** w): This procedure takes as input r $\frac{N}{r}$–vectors and a Boolean variable b. If $b = 0$ then $\frac{N}{r}$–vectors c_i's are concatenated, with every odd-indexed c_i reversed. If $b = 1$ then $\frac{N}{r}$–vectors c_i's are simply concatenated.

- **Multi-2–merge(In:** $\{c_i : i = 0, 1, \ldots, r-1\}$, q, $\frac{N}{r}$; **Out:** w): Construct an r–merge implemented from the q–way 2–merge algorithm, with r input $\frac{N}{r}$–vectors c_i's, where $i = 0, 1, \ldots, r-1$, and output an N–vector w. If c_i's are sorted vectors, then w is sorted.

Procedure Multi-2–merge(In: $\{u_i : i = 0, 1, \ldots, r-1\}$, q, $\frac{N}{r}$; **Out:** w)

1 Pair off the u_i's into pairs $\{u_0, u_1\}, \{u_2, u_3\}, \ldots$.

2 For each pair of $\{u_i, u_{i+1}\}$ ($i = 0, 1, \ldots, \frac{r}{2}-1$), choose $b = 0$ or 1 and apply the **Bi-append(**$\{u_i, u_{i+1}\}$, $\frac{N}{r}$, b; z**)** permutation procedure then follow by the **multi-way multi-merge(**z, $2\frac{N}{r}$, **2**, q, b; w_i**)** procedure.

3 Let the $\frac{r}{2}$ output vectors of sized $(2\frac{N}{r})$ be w_i's where $i = 0, 1, \ldots \frac{r}{2} - 1$. Apply **multi-2–merge(**$\{w_i : i = 0, 1, \ldots, \frac{r}{2} - 1\}$, q, $2\frac{N}{r}$; w**)**.

Using the time complexity of the q–way 2–merge network, we are able to calculate the time complexity

for the network that uses the multi-2-merge algorithm. Using equation 3.1, we have the following:

$$\begin{aligned}
T_k(r,N) &= T_k(2\frac{N}{r}) + T_k(4\frac{N}{r}) + T_k(8\frac{N}{r}) + \ldots + T_k(N) \\
&= \lceil \log_k \frac{2N}{r} \rceil + \lceil \log_k \frac{4N}{r} \rceil + \ldots + \lceil \log_k N \rceil \\
&= \sum_{i=1}^{\lceil \log_2 r \rceil} \lceil \log_k \frac{2^i N}{r} \rceil \\
&= \sum_{i=1}^{\lceil \log_2 r \rceil} \lceil \log_k N - \log_k \frac{r}{2^i} \rceil \\
&\quad (N \textbf{ is a power of } k) \\
&= \lceil \log_2 r \rceil \log_k N - (\sum_{i=1}^{\lceil \log_2 r \rceil} \lceil \log_k \frac{r}{2^i} \rceil) \qquad (3.9)
\end{aligned}$$

Similarly, we can use equation 3.2 to calculate the number of k–comparators used by the network constructed from the multi-2-merge algorithm, however, we shall present the value in terms of the number of k–comparators of the k–way 2–merge networks only:

$$C_k(r,N) = \lceil \frac{r}{2} \rceil C_k(2\frac{N}{r}) + \lceil \frac{r}{4} \rceil C_k(4\frac{N}{r}) + \ldots + C_k(N)$$

Now we can compare the time complexities of the q–way r–merge network for $r > 2$ and the network derived from the multi-2-merge procedure also for $r > 2$, where we compare the depth of each k–comparator networks, where each network uses the same size k–comparators to merge r $\frac{N}{r}$–subvectors.

From equation 3.7, we know that the q–way r–merge network of size N derived from the multi-way multi-merge algorithm will have a depth of $2\log_q \frac{N}{k} + 1$. For simplicity, we only consider the case when $r > 2$ is an even number. This implies that the network derived from the multi-way multi-merge algorithm will use comparators of size $k = (r-2)q$. So the time complexity of the multi-way multi-merge network is:

$$\textbf{Equation 3.7} = (2\log_q(r-2) + 2)\log_k N - O(\log_q k). \qquad (3.10)$$

The corresponding network derived from the multi-2–merge algorithm, using the same sized k–comparators as the above networks, will have the following time complexity (note: $q = k$ for network derived from the family of 2–mergers):

$$\begin{aligned}
\textbf{Equation 3.9} &= \lceil \log_2 r \rceil \log_k N - (\sum_{i=1}^{\lceil \log_2 r \rceil} \lceil \log_k \frac{r}{2^i} \rceil) \\
&\quad (k = (r-2)q \textbf{ for any } r \geq 4,\ q \geq 3)
\end{aligned}$$

$$= \lceil \log_2 r \rceil \log_k N - \lceil \log_2 r \rceil$$
$$= \lceil \log_2 q \log_q r \rceil \log_k N - \lceil \log_2 r \rceil. \quad (3.11)$$

Comparing the coefficients of the leading term of Equation 3.10 and Equation 3.11 ($2\log_q(r-2)+2$ and $\lceil \log_2 q \log_q r \rceil$ respectively), we can conclude the following about these two time complexities: If $q \leq 4$, then the q–way r–merge network derived from the multi-2–merge algorithm has a better time complexity than the q–way r–merge network derived from the multi-way multi-merge algorithm. However, if $q > 4$ and r is large enough, then the converse is true. For example, in the case $q = 5$, when the number of vectors to be merged is greater than 21986, then the q–way r–merge network derived from the multi-way multi-merge algorithm is better than the q–way r–merge network derived from the multi-2–merge algorithm. And as q gets larger, the number of vectors r needed to be merged gets smaller for the converse to be true. In case of $q = 8$ or 16, we only need $r \geq 60$ and $r \geq 14$ respectively for the multi-way multi-merge network to perform better than the network that derived from the multi-2–merge algorithm.

The reason that as r increases, the multi-way multi-merge network performs better than the network derived from the multi-2–merge algorithm is because of the following: we assume that the input length N and parameter q remain fixed, when we compare the different time complexities. Hence for the multi-way multi-merge procedure ($r > 2$) to accommodate a larger number of vectors that need to be merged, the size of each k–comparator in the q–way r–merge network will need to be increased, but the depth of the network does not increase (if size of the k–comparator used is bigger, the depth might be reduced in some cases). However, in the case of multi-2–merge procedure, one extra iteration of the q–way 2–merge procedure will be needed to merge the extra pairs of subvectors, hence the depth of the corresponding k–way merge network will increase as a result.

3.5 The Sorting Network Using the Multi-Way Multi-Mergers And Its Time Complexity

In this section, we will construct k–comparator sorting networks of size N, which uses the multi-way multi-merge procedure in an iterative manner. Before we state the sorting algorithm formally, we briefly describe the algorithm: To sort an N–vector, we first decompose it into $\frac{N}{k}$ k–subvectors and sort those k–subvectors (assume that we are using k–comparators). Then we merge the k–subvectors in groups of $r \geq 2$ then form $\frac{N}{rk}$ sorted (rk)–subvectors, and continue to merge the result subvectors in groups, until we output a sorted N–vector. However, we would like the resulting k–comparator network derived from this sorting algorithm to use k–comparators of equal size k (for practical reasons). We now fix r, and we restrict the number of vectors to be merged in each group to be either r or 2.

Sorting Algorithm using multi-way multi-mergers

Input V (an N–vector).
Decompose V into $\frac{N}{k}$ k–subvectors.
Sort each k–subvector by a k–comparator.
$N^* \leftarrow k$.
While $N^* < N$ **Do**
Begin

1 Choose a value for r'; either $r' = r$ or $r' = 2$.

2 Arrange sorted N^*–vectors from the previous iteration into groups of r' consecutive vectors.

3 For each group of $\{u_i, u_{i+1}, \ldots, u_{i+r'-1}\}$, we choose $b = 0$ or 1 and apply the **Bi-append**$(\{u_j : j = i, i+1, \ldots, i+r'-1\}\ N^*,\ b;\ w)$ permutation procedure then follow by the **multi-way multi-merge**$(r'N^*,\ r',\ q,\ b;\ z)$ procedure.

4 $N^* \leftarrow r'N^*$.

End
Output the resulting N–vector.

The correctness of this sorting algorithm follows immediately from the correctness of the multi-way multi-merge procedure when the input vector has finite length N. The time complexity and the number of k–comparators used for this sorting network can be derived from the q–way r–merge network's. Furthermore, we extend the terminology from the previous section, **the k–comparator sorting network that is constructed using the q–way r–merge network will be called the q–way r–merge sorting network.** Figure 3.16, Figure 3.17 and Figure 3.18 illustrate some of the sorting networks.

From the above sorting algorithm, a class of sorting networks can be generated. It will be impractical to calculate the time complexities for every possible network that can be derived from this sorting algorithm. Hence we will be focusing on the time complexities of various extreme cases.

Suppose we were to consider the time complexity of the k–way r–merge sorting networks. Then we can derive the time complexity of the k–way 2–merge sorting network using the time complexity of the multi-way multi-merge networks. Let the depth of this sorting network of size N be given by $T_k^s(N)$ and the number of k–comparators used by $C_k^s(N)$.

$$T_k^s(N) = \left(\sum_{i=1}^{\lceil \log_2 \frac{N}{k} \rceil} T_k(2^i k)\right) + 1$$

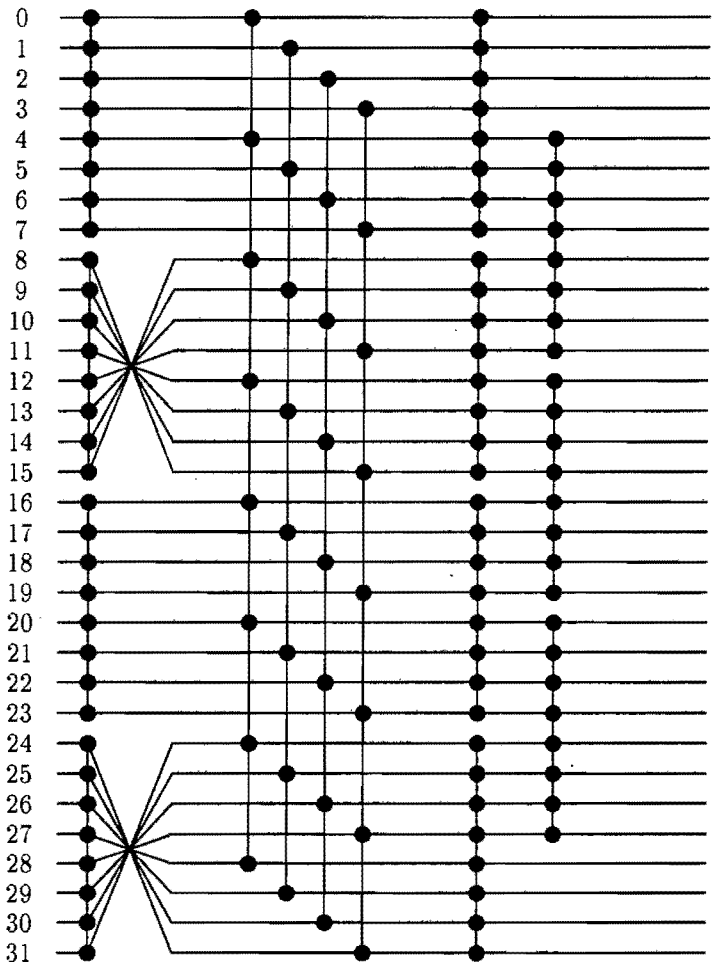


Figure 3.16: An example of the line representation of a sorting network of size 32 constructed from the multi-way multi-merge procedure, where the underlying grid algorithm used is the 8-way 4-merge 4-tonic grid algorithm.

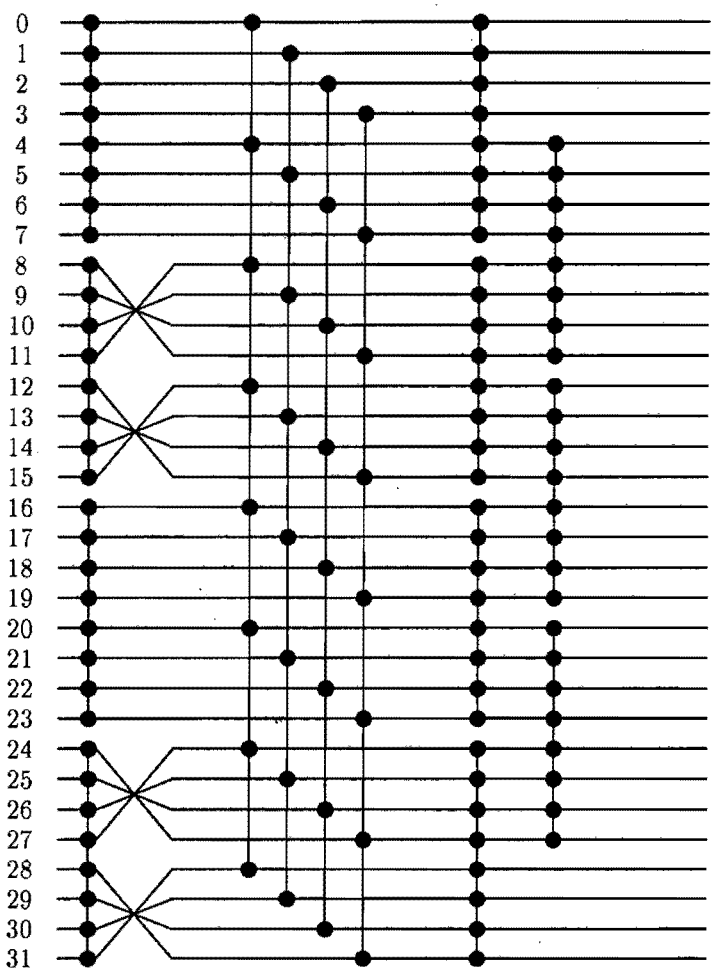


Figure 3.17: An example of the line representation of a sorting network of size 32 constructed from the multi-way multi-merge procedure, where the underlying grid algorithm used is the 8–way 4–merge modulo grid algorithm.

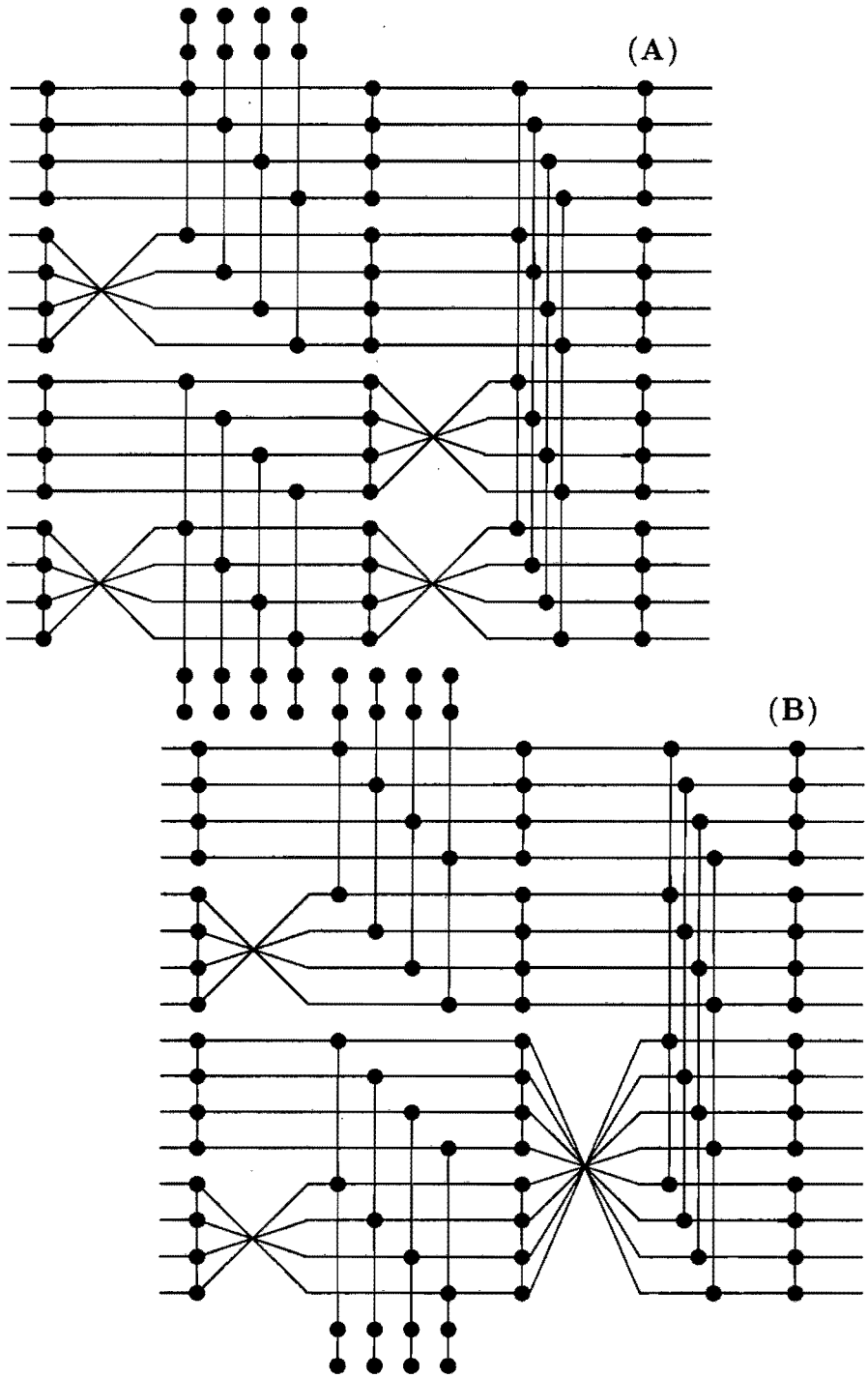


Figure 3.18: Two examples of the line representation of the sorting networks (observe that in both cases, the 4-comparators in the second layer are partially filled): (A) A 4-comparator sorting network that used the 4-way 2-merge modulo grid algorithm as the underlying merge algorithm. (B) A 4-comparator sorting network that used the 4-way 2-merge bitonic grid algorithm as the underlying merge algorithm.

$$= (\sum_{i=1}^{\lceil \log_2 \frac{N}{k} \rceil} \lceil \log_k (2^i k) \rceil) + 1$$
$$= (\sum_{i=1}^{\lceil \log_2 \frac{N}{k} \rceil} \lceil \log_k 2^i \rceil) + (\lceil \log_2 \frac{N}{k} \rceil + 1)$$

If N is a power of k and k is a power of 2, then

$$\begin{aligned} T_k^s(N) &= (\sum_{j=1}^{\log_k(N)-1} \log_2(k) j) + \log_2(k) \log_k \frac{N}{k} + 1 \\ &= \log_2(k) \frac{\log_k(N)(\log_k(N)-1)}{2} + \log_2(k) \log_k \frac{N}{k} + 1 \\ &= \frac{\log_2(k)}{2} \log_k(N) \log_k \frac{N}{k} + \log_2(k) \log_k \frac{N}{k} + 1 \\ &= \frac{\log_2(k)}{2} (\log_k \frac{N}{k})^2 + O(\log_2 \frac{N}{k}) \\ &= \frac{\log_2(k)}{2} (\log_k N)^2 + O(\log_2 \frac{N}{k}). \end{aligned} \tag{3.12}$$

and we have the following:

$$\begin{aligned} C_k^s(N) &= (\sum_{i=1}^{\lceil \log_2 \frac{N}{k} \rceil} \lceil \frac{N}{2^i k} \rceil C_k(2^i k) + \frac{N}{k} \\ &= (\sum_{i=1}^{\lceil \log_2 \frac{N}{k} \rceil} \lceil \frac{N}{2^i k} \rceil \lceil \log_k (2^i k) \rceil \lceil \frac{2^i k}{k} \rceil) + \frac{N}{k} \end{aligned}$$

If N is a power of k and k is a power of 2, then

$$\begin{aligned} C_k^s(N) &= (\sum_{i=1}^{\log_2 \frac{N}{k}} \frac{N}{k} \lceil \log_k (2^i k) \rceil) + \frac{N}{k} \\ &= (\sum_{i=1}^{\log_2 \frac{N}{k}} \frac{N}{k} (\lceil \log_k (2^i) \rceil + 1)) + \frac{N}{k} \\ &= (\frac{N}{k} \sum_{i=1}^{\log_2 \frac{N}{k}} \lceil \log_k (2^i) \rceil) + \frac{N}{k} \frac{(\log_2(\frac{N}{k}) + 1) \log_2 \frac{N}{k}}{2} + \frac{N}{k} \\ &= \frac{N}{k} (\log_2(k) \frac{\log_k(N)(\log_k(N)-1)}{2} + \frac{(\log_2(\frac{N}{k}) + 1) \log_2(\frac{N}{k})}{2} + 1) \\ &= \frac{N}{k} (\log_2(k) \frac{(\log_k N)^2}{2} + \frac{(\log_2 N)^2 + (\log_2 k)^2}{2} - O(\log_2 N \log_2 k)). \end{aligned}$$

Now, $C_k^s(N)$ and $\frac{N}{k} T_k^s(N)$ have the same leading term, however $C_k^s(N) > \frac{N}{k} T_k^s(N)$ in the lower order terms, this implies this network uses extra k-comparators. This is caused by the fact that when the algorithm merges pairs of small subvectors, many of the k-comparators are partially filled (the length of

input subvectors is less than k). Hence the k–way 2–merge sorting network is resource wasteful, unless different size comparators are used in the network or only the 2–comparators are used.

Suppose we were to consider the time complexity of the q–way r–merge sorting networks ($r > 2$), then we can derive the time complexity of the q–way r–merge sorting network using the time complexity of the multi-way multi-merge network where $r > 2$. The time complexity of the sorting network for the case $r > 2$ can be shown to be different to the case $r = 2$.

$$\begin{aligned} T_k^s(N) &= \sum_{i=0}^{\lceil \log_r \frac{N}{k} \rceil} T_k(r^i k) \\ &= \sum_{i=0}^{\lceil \log_r \frac{N}{k} \rceil} (2\lceil \log_q r^i \rceil + 1) \end{aligned} \tag{3.13}$$

The depth derived in equation 3.13 is a complicated expression and it is difficult to simplify, hence we shall make a few assumptions: we assume that $r = q$ and $N = q^s k$ for some positive integer s. Now we have the following result:

$$\begin{aligned} \textbf{Equation 3.13} &= \sum_{i=0}^{\log_q \frac{N}{k}} (2 \log_q q^i + 1) \\ &= \sum_{i=0}^{\log_q \frac{N}{k}} (2i + 1) \\ &= 2\frac{\log_q \frac{N}{k}(\log_q \frac{N}{k} + 1)}{2} + (\log_q \frac{N}{k} + 1) \\ &= (\log_q \frac{N}{k} + 1)^2 \\ &= (\log_q N)^2 - O(\log_q k \log_q N) \\ &= (\log_q k \log_k N)^2 - O(\log_q k \log_q N). \end{aligned} \tag{3.14}$$

and we have the following:

$$\begin{aligned} C_k^s(N) &= \sum_{i=0}^{\lceil \log_r \frac{N}{k} \rceil} \lceil \frac{N}{q^i k} \rceil C_k(q^i k) \\ &= \sum_{i=0}^{\log_q \frac{N}{k}} \frac{N}{q^i k}(2q^i(\log_q q^i + 1) - \frac{q^{i+1} - 1}{q - 1}) \\ &= \frac{N}{k}(\sum_{i=0}^{\log_q \frac{N}{k}} 2(i + 1) - \frac{q^{i+1} - 1}{q - 1}) \\ &\leq \frac{N}{k}(2\frac{\log_q \frac{N}{k}(\log_q \frac{N}{k} + 1)}{2} + O(\log_q \frac{N}{k})) \\ &= \frac{N}{k}(\log_q \frac{N}{k})^2 + O(\frac{N}{k} \log_q \frac{N}{k}) \end{aligned}$$

$$= \frac{N}{k}(\log_q k \log_k N)^2 + O(\frac{N}{k}\log_q \frac{N}{k}).$$

To compare the time complexities of equation 3.12 and equation 3.14, we will need to assume that the networks are using k–comparators of the same size (equal to k). Hence the following conditions need to be true: we made the assumption that $q = r$, when we derive the time complexity of the q–way r–merge sorting network. Furthermore, we assume $k = q$ is a power of 2 when we calculate the time complexity for the k–way 2–merge sorting network. Hence for the k–comparators to satisfy both restrictions, we will need that $k = k^*q = (r-2)r = 2^s$ for some positive integer s. But there are too few possible integer solutions for r and s that satisfy this restriction, which makes it impractical (for example $r = 4$ and $s = 3$ is one of the solution set). Hence in this section we will relax the restriction that $k = k^*q$, and instead we will only require $k > k^*q$ and k is a power of 2 as the size of k–comparators used in the q–way r–merge sorting network.

Now, if we were to compare the leading coefficients of equation 3.12 and equation 3.14 ($\frac{\log_2(k)}{2}$ and $(\log_q k)^2$ respectively), then when $q \geq 16$ we will have $\frac{\log_2(k)}{2} \geq (\log_q k)^2$, assuming we restrict the size of the comparator to be $(q-2)q \leq k \leq q^2$. So under the above conditions (which are reasonable restrictions), we see that the q–way r–merge sorting network is more efficient than the q–way 2–merge sorting network. However, if $r < 16$, then the k–way 2–merge sorting network will need less depth than the q–way r–merge sorting network of the same size (input/output of size N). Also the q–way r–merge sorting network still uses fewer comparators than the k–way 2–merge sorting network (if $r \leq 4$).

3.6 Combining the q–way r–merger and the q–way 2–merger to Sort

In the earlier section, we mentioned that the manner in which we implement the multi-way multi-merge procedure is not the only way to construct the q–way r–merge sorting networks. It is possible to combine the methods developed from the family of r–mergers and from the family of 2–mergers.

In the previous section, we introduced a class of sorting networks, which utilizes the multi-way multi-merge procedure in an iterative manner. We have also shown that, under the assumption that $r = q$, the sorting network derived from the multi-way multi-merge procedure with a fixed value for r (where $r \geq 16$) will be more efficient than the sorting network derived from the multi-way multi-merge procedure with $r = 2$. To complete the work investigated in this chapter, we will provide an algorithm that construct sorting network that requires less depth even when $r < 16$ than the depth of the corresponding k–way 2–merge sorting network.

In this example case, the time complexity will be expressed in terms of the parameters q, k and N used in the q–way r–merge network. Furthermore, we shall not confuse the parameter q used here with

the parameter q used in the q–way 2–merge type of network, since in the 2–merge network we require that $q = k$. But in the case of the q–way r–merge network we require that $q < k$. In particular, we will assume the set of parameters as the following: $k = q^2$ and $r = q$. As we have discussed in the previous section, we used slightly larger comparators than what is required. This is to facilitate the fact that the resulting network is to accommodate both the q–way r–merge networks and the k–way 2–merge networks. So instead of using $k = k^*q = (q-2)q$, in this case we used the size $k = q^2 > k^*q$ as the size of the k–comparators.

We already know that using the network derived from the multi-2–merge algorithm to merge $r = q$ sorted vectors requires more comparators than the q–way q–merge network used to merge the same set of sorted vectors. Hence we will be concerned with improving the depth of the network. We know that the time complexity of the q^2–way 2–merge algorithm for size N is $T_k(N) = \lceil \log_k N \rceil$. Hence the networks used to implement the multi-2–merge algorithm that merge r sorted subvectors have the following depth:

$$T_k(r, N) = (\lceil \log_k(2N) \rceil + \lceil \log_k(4N) \rceil + \ldots + \lceil \log_k(rN) \rceil)$$

if N is an even power of q, then

$$\begin{aligned} T_k(r, N) &= \sum_{i=1}^{\lceil \log_2 r \rceil} (\lceil \log_k 2^i \rceil + \log_k N) \\ &= (\sum_{i=1}^{\lceil \log_2 r \rceil} \lceil \log_k 2^i \rceil) + \lceil \log_2 r \rceil \log_k N \qquad (r \leq k) \\ &= \lceil \log_2 r \rceil + \lceil \log_2 r \rceil \log_k N \\ &= \lceil \log_2 r \rceil (1 + \log_k N). \end{aligned}$$

The following observation is of particular interest in the construction of the next sorting algorithm. Following from the calculation above, we observed that if N is an even power of q and we have r sorted (qN)–vectors, then as the calculation shown below, the depth required for the networks used in both cases are the same.

$$T_k(r, (qN)) = (\lceil \log_k(2qN) \rceil + \lceil \log_k(4qN) \rceil + \ldots + \lceil \log_k(rqN) \rceil)$$

if N is an even power of q, then

$$\begin{aligned} T_k(r, (qN)) &= \sum_{i=1}^{\lceil \log_2 r \rceil} (\lceil \log_k(2^i q) + \log_k N \rceil) \\ &= (\sum_{i=1}^{\lceil \log_2 r \rceil} \lceil \log_k(2^i q) \rceil) + \lceil \log_2 r \rceil \log_k N \qquad (rq \leq k) \\ &= \lceil \log_2 r \rceil + \lceil \log_2 r \rceil \log_k N \\ &= \lceil \log_2 r \rceil (1 + \log_k N). \end{aligned}$$

Immediately we have the following lemma.

Lemma 3.6.1 *Under the multi-2-merge algorithm with $k = q^2$ and $r = q$, the k-comparator network used to merge q sorted q^{2i}-vectors (where i is a positive integer) and the k-comparator network used to merge q sorted q^{2i+1}-vectors will have the same depth, which is equal to $\lceil \log_2 q \rceil (i+1)$.*

Proof: The proof of this lemma follows immediately from the observation made above. ∎

Under the assumption that N is a power of k, $q = r$ and $k = q^2$, then the leading term of the respective depths of the sorting networks derived from the multi-way multi-merge procedure with $r = 2$ and $r > 2$, are $\lceil \log_2 q \rceil \log_k N + O(\log_2(r))$ and $4 \log_k(N) + O(\log_2(r))$ respectively. It is clear from the above two depths, that if $r \geq 16$ the q-way r-merge sorting network will have less time delay than the k-way 2-merge sorting network. However, the opposite will be true if $2 < r < 16$. Next, we will demonstrate that it is possible to construct a sorting network that has less depth than the k-way 2-merge sorting network, when $r < 16$ (number of subvectors that needed to be merged).

Procedure Combined multi-way multi-merge(In: v, (rN), r, q, b; **Out:** w)

If size of the vector is less or equal k **then** apply a k-comparator; $(k = q^2)$
Else
Begin
If $b = 1$ **then** apply the **ModuloPerm**(v, (rN), k, q; z) **else** $z \leftarrow v$;
Divide(z, rN, q, m-**chains:** $\{c_i : i = 0, 1, \ldots, q-1\}$); $(N = p \times q)$
multi-2-merge(Divide(c_0, rp, r, *quotient*(p)-**chain;** $\{c_j^0 : j = 0, \ldots, r-1\}$), q, p; w_0).
$\ldots$,
multi-2-merge(Divide(c_{q-1}, rp, r, *quotient*(p)-**chain;** $\{c_j^{q-1} : j = 0, \ldots, r-1\}$), q, p; w_{q-1});
Unite($\{w_i : i = 0, 1, \ldots, q-1\}$, rp, m-**chains;** w');
Transformer(In: w', rN, k, q, m-**chains;** w).
End

Suppose that $k = q^2$, $r = q$ and that N is a power of k. Let $T_k^c(r, N)$ denote the depth of the network derived from the combined multi-way multi-merge procedure using k-comparators. Let $T_k^2(r, N)$ denote the depth of the multi-2-merger using k-comparators. Then we have the following:

$$\begin{aligned} T_k^c(q, N) &= 2 + T_k^2(q, \frac{N}{q}) \\ &= 2 + \lceil \log_2 q \rceil \log_k N. \qquad (k = q^2) \end{aligned}$$

Following Lemma 3.6.1, we realize that it is possible to improve the sorting network constructed from the q^2–way 2–merge networks in an iterative manner, and we can reduce the depth of the network by applying the combined multi-way multi-merge, when the size of input vector is an even power of q. Next, we present a faster sorting algorithm for $q = r \leq 16$:

The Combined Sorting Algorithm

Input V (an N–vector).
Decompose V into $\frac{N}{k}$ k–subvectors. $(k = q^2)$
Sort each k–subvector using a k–comparator.
For $i = 1$ to $\lceil \log_r \frac{N}{k} \rceil$ **Do**
If i is odd **then**

For each group of r sorted $(r^{i-1}k)$–vectors, $\{c_j^0 : j = 0, \ldots, r-1\}$, we choose $b = 0$ or 1 and apply the **Bi-append**$(\{c_j^0 : j = 0, \ldots, r-1\}, r^{i-1}k, b; z)$ then followed by the **combined multi-way multi-merge**$(z, r^i k, r, q, b; w)$ procedure.

else

For each group of r sorted $(r^{i-1}k)$–vectors, $\{c_j^0 : j = 0, \ldots, r-1\}$, we apply the **multi-2-merge**$(\{c_i : i = 0, 1, \ldots, r-1\}, q, r^i k; w)$ procedure.

Output the resulting N–vector.

The depth of the network derived from the combined algorithm is the follows (assume that $k = q^2$ and $q = r$):

$$\begin{aligned} T_k^s(N) &= 1 + T_k^c(q, q^2) + T_k^2(q, q^3) + T_k^c(q, q^4) + \ldots \qquad \text{(until we reach } N.) \\ &= 1 + (2 + T_k^2(q, q)) + T_k^2(q, q^3) + (2 + T_k^2(q, q^3)) + \ldots \end{aligned}$$

If N is a power of k, then

$$\begin{aligned} T_k^s(N) &= 3 + \lceil \log_2 q \rceil + \sum_{i=1}^{\log_k(N)-2} (2 + 2T_k^2(q, q^{2i})) + \lceil \log_2 q \rceil \log_k N \qquad \text{(from Lemma 3.6.1)} \\ &\quad \text{(let } K \text{ denote } \log_k(N) - 2) \\ &= 3 + \lceil \log_2 q \rceil + 2K + \lceil \log_2 q \rceil (\sum_{i=1}^{K} (2i + 2)) + \lceil \log_2 q \rceil \log_k N \\ &= 3 + \lceil \log_2 q \rceil + 2K + \lceil \log_2 q \rceil K(K+1) + \lceil \log_2 q \rceil 2K + \lceil \log_2 q \rceil \log_k N \\ &= \lceil \log_2 q \rceil (\log_k N)^2 - \frac{\lceil \log_2 q \rceil}{2} \log_k N + 2 \log_k N + \lceil \log_2 q \rceil - 1 \end{aligned}$$

Furthermore, if we assume k is a power of 2, then the above equation will be

$$T_k^s(N) = \frac{\log_2 k}{2}(\log_k N)^2 - \frac{\log_2 k}{4}\log_k N + 2\log_k N + \frac{\log_2 k}{2} - 1$$

Since the depth of the sorting network constructed from the k–way 2–merge networks is equal to $T_k^s(N) = 1+T_k^2(q,q^2)+T_k^2(q,q^3)+T_k^2(q,q^4)+\ldots$. Comparing this to the depth of the network constructed from the combined sorting algorithm, we find that the new combined algorithm reduces the depth by $\lceil\log_2 q\rceil \log_k N - 2\log_k N - \lceil\log_2 q\rceil + 2$ layers. One last observation: we can use one layer of k–comparators ($k = q^2$) to implement $T_k^2(q,q)$, instead of $\lceil\log_2 q\rceil$ layers of k–comparators used.

3.7 The Construction of k–Comparators

One aspect of the work that has not been investigated is how to construct a k–comparator. In previous sections, we bypassed this issue by assuming that it is possible to construct a k–comparator. Under the obvious definition, given by Olariu, Pinotti and Zheng [69], that a k–comparator is a sorting device capable of sorting k elements in constant time, our assertion that it is possible to construct the k–comparators is not unreasonable. It is noted by Olariu, Pinotti and Zheng [69] that a reconfigurable mesh of size $k \times k$ can be considered as a suitable candidate to function as a k–comparator, because the reconfigurable mesh can sort k elements in $O(1)$ time. There are numerous papers in the literature that formulate fast sorting algorithms on the reconfigurable meshes, for examples Jang and Prasanna [45], Nigam and Sahni [67].

The processors in the reconfigurable mesh of size $k \times k$ are connected by a bus system, whose configuration can be dynamically changed to suit computational needs. So the reconfigurable meshes are much more complex than the 2–comparator networks, and as Olariu, Pinotti and Zheng [69] have pointed out further: the k–comparator constructed from the reconfigurable mesh of size $k \times k$ is much more expensive than that constructed from the sorting network of size k using 2–comparators and its use should be avoided whenever possible. However, we postulate that the concern of the high cost of constructing the k–comparators will become less significant in the future, by observing the following:

1 Technology is advancing at such an astonishing pace, it is not difficult to conceive that it will be possible to find a cost efficient method of manufacturing the reconfigurable meshes in the near future. We shall quote from the paper of Olariu and Schwing [68] to demonstrate this point: ... *the reconfigurable bus system [can be] implemented using optical fibers as the underlying global bus system and using electrically controlled directional coupler switches for connecting or disconnecting two fibers. Due to these new developments, the reconfigurable mesh is likely to become commercially available in the near future.* Hence it is very likely that some new technological improvements will be discovered, so that the cost of manufacture the k–comparators

can be drastically reduced.

2 The reconfigurable mesh of size $k \times k$ is a suitable candidate to function as a k–comparator. However, there maybe other alternative approaches that are cheaper to construct, either by using different kinds of devices or by using different underlying architectural design. One such possibility is to implement the k–comparators using some kind of analogue processors. Hence further study on possible alternative approaches is needed.

Consequently, developing sorting algorithms to utilize the k–comparators may become of practical relevance, so it is important that we are able to find elegant and efficient sorting algorithms that utilize the k–comparators. Although we disagree with Olariu, Pinotti and Zheng [69] on the subject that it is more viable to use the 2–comparator networks of size k to emulate the k–comparators than using the k–comparators, we agree that it is of practical importance to investigate the effect on the time complexity of our sorting algorithms, if we were to replace the k–comparators with the 2–comparator networks of size k. Obviously, if k–comparators are used as the measurement for the time delay, then our generalized sorting algorithm will be more efficient when $r \geq 16$ and our combined sorting algorithm will be more efficient when $2 < r < 16$. However, if 2–comparators are used to implement the required k–comparators, then the outcome may be different. Suppose we use the Batcher's bitonic 2–comparator sorting network to implement a k–comparator. Then each k–comparator will be delayed by a further factor of $\frac{1}{2}(\log_2 k)(\log_2(k)+1)$ (since the bitonic 2–comparator sort used $\frac{1}{2}(\log_2 k)(\log_2(k)+1)$ layers of 2–comparators).

In the case of the algorithms of Nakatani et al [64] and Liszka and Batcher [62], the k–comparators in their algorithms will require less time delay, since the k–comparators are used to sort bitonic subvectors (the rows in the resulting grid, see Lemma 3.4.1). Hence the generalized 2–merge algorithms constructed will need k–comparators to perform k–merge operations, not as k–sorters (which was the case in the our generalized algorithm). Now, the bitonic merger of size k needs $\log_2 k$ layers of 2–comparators, hence the k–comparators used in their algorithm will only delay by a factor of $\log_2 k$.

3.8 Summary of the Time Complexity

In this section we will give a brief summary of the time complexity of various multi-way multi-merge network. We present in the next table a list of the time complexities of various k–comparator networks. The parameter N is the size of the input vector, q is the modulo of the network (in the grid representation, q is the width of the grid), r is the number of sorted subvectors that are merged and k is the size of the comparator used. We assume that N is a power of k and the k–comparator sorts in $O(1)$.

Under the assumption that $q = k$ and $r = 2$, the time complexity of a q–way 2–merge network has $\log_k N$ time delay and it needs $\frac{N}{k}\log_k N$ k–comparators.

The Time Complexity of Various k–Comparator Networks

Type of Network	**Time Complexity:** $T_k(N)$	**No. of k–Comparators:** $C_k(N)$
The k–way 2–merge network	$\log_k N$	$\frac{N}{k}\log_k N$
The q–way r–merge network	$2\log_q k \log_k N - O(\log_q k)$	$\frac{N}{k}(2\log_q k \log_k N + O(\log_q k))$
The r–merge network derived from the multi-2–merge procedure ($r > 2$)	$\lceil \log_2 r \rceil \log_k N - \lceil \log_2 r \rceil$	$\sum_{i=1}^{\lceil \log_2 r \rceil} \lceil \frac{r}{2^i} \rceil \frac{2^i N}{rk} \log_k \frac{2^i N}{r}$
The r–merge network derived from the combined procedure ($r > 2$)	$2 + \lceil \log_2 q \rceil \log_k N$	–
The k–way 2–merge sorting network	$\frac{\log_2 k}{2}(\log_k N)^2 + O(\log_2 \frac{N}{k})$	$\frac{N}{k}(\log_2 k \frac{(\log_k N)^2}{2} + O((\log_2 N)^2))$
The q–way r–merge sorting network	$(\log_q k \log_k N)^2 - O(\log_q k \log_q N)$	$\frac{N}{k}((\log_q k \log_k N)^2 + O(\log_q \frac{N}{k}))$
The network derived from the combined sorting algorithm (for $r > 2$)	$\lceil \frac{\log_2 k}{2} \rceil (\log_k N)^2 - O(\log_k N)$	–

Table 3.1: The time complexity of the k–Comparator networks. Entries with "–" indicate that we did not find the number of k–comparators used by those particular networks.

Under the assumption that $N = q^s k$ for some positive integer s, the time complexity of a q-way r-merge network has $2\log_q k \log_k N - O(\log_q k)$ time delay and it needs $2\frac{N}{k}\log_q k \log_k N + O(\frac{N}{k}\log_q k))$ k-comparators.

Under the assumption that N is a power of k, r is a power of 2 and $k \geq 2$, the time complexity of an r-merge network derived from the multi-2-merge procedure ($r > 2$) has $\lceil \log_2 \rceil \log_k N - \lceil \log_2 r \rceil$ time delay and it needs $\sum_{i=1}^{\lceil \log_2 r \rceil} \lceil \frac{r}{2^i} \rceil \frac{2^i N}{rk} \log_k \frac{2^i N}{r}$ k-comparators.

Under the assumption that N is a power of k, $r = q$ and $k = q^2$, the time complexity of an r-merge network derived from the combined procedure ($r > 2$) has $2 + \lceil \log_2 q \rceil \log_k N$ time delay.

Comparing the coefficients of the leading term of above three r-merge networks, we can conclude the following about these time complexities: The network derived from the multi-2-merge procedure and the r-merge network derived from the combined procedure have the same leading term. However, if N is a power of k, the r-merge network derived from the multi-2-merge procedure has smaller lower order term, and if N is not a power of k, then the r-merge network derived from the combined procedure has smaller lower term. If $q \leq 4$, then the q-way r-merge network derived from the multi-2-merge algorithm has a better time complexity than the q-way r-merge network derived from the multi-way multi-merge algorithm. If $q > 4$ and r is large enough, then the converse is true. As for the number of k-comparators used, the q-way r-merge network required fewer k-comparators than the other two r-merge networks.

Under the assumption that N is a power of k and r is a power of 2, the time complexity of a q-way 2-merge sorting network has $\frac{\log_2(k)}{2}(\log_k N)^2 + O(\log_2 \frac{N}{k})$ time delay and it needs $\frac{N}{k}(\log_2(k)\frac{(\log_k N)^2}{2} + \frac{(\log_2 N)^2 + (\log_2 k)^2}{2} - O(\log_2 N \log_2 k))$ k-comparators. We then observed that $C_k^s(N)$ and $\frac{N}{k}T_k^s(N)$ has the same leading term, however $C_k^s(N) > \frac{N}{k}T_k^s(N)$ in the lower order terms, this implies this network uses extra k-comparators. This is caused by the fact that when the algorithm merges pairs of small subvectors, many of the k-comparators are partially filled (see Figure 3.18).

Under the assumption that $N = q^s k$ for some positive integer s and $r = q$, the time complexity of a q-way r-merge network has $(\log_q k \log_k N)^2 - O(\log_q k \log_q N)$ time delay and it needs $\frac{N}{k}(\log_q k \log_k N)^2 + O(\frac{N}{k}\log_q \frac{N}{k})$ k-comparators.

If we restrict the size of the comparator to be $(q-2)q \leq k \leq q^2$, then when $q \geq 16$ the q-way r-merge sorting network is more efficient than the q-way 2-merge sorting network. However, if $r < 16$, then the k-way 2-merge sorting network will need less depth than the q-way r-merge sorting network of the same size (input/output of size N). Also the q-way r-merge sorting network still uses fewer comparators than the k-way 2- merge sorting network (if $r \leq 4$).

Under the assumption that N is a power of k, $r = q$ and $k = q^2$, the time complexity of an r-merge sorting network derived from the combined sorting procedure has $\lceil \frac{\log_2 k}{2} \rceil (\log_k N)^2 - O(\log_k N)$ time delay. This time complexity has the same leading term as the q-way 2-merge sorting network, however,

it has a smaller lower order term. Furthermore, it requires less k–comparators.

3.9 Sorting on Rectangular Mesh-Connected Networks

Certain of the grid algorithms can be used as a basis of sorting on rectangular planar interconnection networks. We will investigate the tightness of each algorithm with respect to the time lower bounds needed to sort on various models of interconnection network. Various time lower bounds for different models can be found in Schnorr and Shamir [89]. We will use the time lower bounds of various models provided by Schnorr and Shamir [89], where appropriate. If the time lower bounds needed to sort on a particular model that is not proved by Schnorr and Shamir [89], then we shall derive the appropriate time lower bound, by following the reasoning of Schnorr and Shamir [89] and without providing any formal proof. Henceforth, when we consider the time upper bound of an algorithm, we consider the constant of the leading term, but ignore the low order additive terms in the time complexity function (using the big O notation).

In the calculation of the time upper bound of each sorting algorithm, we will need to resort to an obvious comparison-exchange algorithm, the odd-even transposition sort (see [48] for an early reference). Here we give a brief description of the odd-even transposition sort. A detailed investigation of this algorithm and of networks can be found in Leighton [58] and the implementation of the odd-even transposition sort on a planar interconnection network can be found in Thompson and Kung [95]. A network that uses odd-even transpositions to sort N elements has N stages of 2–comparators and uses $\frac{1}{2}N(N-1)$ 2–comparators. The 2–comparators are arranged in a brick-like pattern: the elements in position i and $i+1$ of the N–vector are compared and possibly exchanged in stage t of the network, when $i+t$ is even ($0 \leq i < N-1$ and $0 \leq t \leq N-1$). Smaller values are moved to the top of the array during each exchange. Another observation, in most of the sorting algorithms on 2–D meshes, is that the sorted array produced from the odd-even transposition sort on a 2–D mesh has snake-like row major order (although, this depends highly on connections and method used). Hence, to convert sorted arrays into the row major index, we need to reverse the order of elements in each alternative row and this will take $N^{\frac{1}{3}}$ time cycles, if the row contain $N^{\frac{1}{3}}$ elements.

Theorem 3.9.1 *The time complexity of the original column sort of Leighton [56] and of the modified column sort on an* $N^{\frac{2}{3}} \times \frac{N^{\frac{1}{3}}}{2}$ *array is* $8N^{\frac{2}{3}} + O(N^{\frac{1}{3}})$ *and* $4N^{\frac{2}{3}} + O(N^{\frac{1}{3}})$ *respectively.*

Proof: We will prove the time complexity of the original column sort of Leighton [56] first, then we will prove the time complexity of the modified column sort. The procedure of the original column sort can be found on page 35. The procedure of the modified column sort can be found on page 36.

The original algorithm sorts the columns in Step 1, Step 3, Step 5 and Step 7, this can be done by

the odd-even transposition sort, which takes $N^{\frac{2}{3}} + \frac{N^{\frac{1}{3}}}{2}$ time cycles each time we sort the columns. The transpose operation in Step 2 and the inverse transpose operation will each takes $N^{\frac{2}{3}}$ time cycles. The half shift of Step 6 and Step 8 will each takes $\frac{N^{\frac{2}{3}}}{2}$ time cycles. Hence the total time delay is $8N^{\frac{2}{3}} + O(N^{\frac{1}{3}})$.

Next the modified algorithm sorts the blocks of size $N^{\frac{2}{3}}$ in Step 1. Again this can be done by the odd-even transposition sort, which takes $N^{\frac{2}{3}} + \frac{N^{\frac{1}{3}}}{2}$ time cycles. In Step 2 the columns are sorted, which takes $N^{\frac{2}{3}} + \frac{N^{\frac{1}{3}}}{2}$ time cycles. We are left with $2N^{\frac{1}{3}}(N^{\frac{1}{3}} - 1)$ unsorted numbers and this can be done by the odd-even transposition sort, which takes $2N^{\frac{2}{3}}$ time cycles. Hence the total time delay is $4N^{\frac{2}{3}} + O(N^{\frac{1}{3}})$. ∎

Next we will calculate the time upper bound for the new merge algorithm.

Theorem 3.9.2 *The time complexity of the $N^{\frac{1}{3}}$-way $N^{\frac{1}{3}}$-merge $N^{\frac{1}{3}}$-tonic algorithm and of the $N^{\frac{1}{3}}$-way $N^{\frac{1}{3}}$-merge modulo algorithm on an $N^{\frac{2}{3}} \times N^{\frac{1}{3}}$ array are both $2.5N^{\frac{2}{3}} + O(N^{\frac{1}{3}})$.*

Proof: We will prove the time complexity of the $N^{\frac{1}{3}}$-way $N^{\frac{1}{3}}$-merge $N^{\frac{1}{3}}$-tonic algorithm first, where we will refer back to the procedure of the $N^{\frac{1}{3}}$-way $N^{\frac{1}{3}}$-merge $N^{\frac{1}{3}}$-tonic algorithm described on page 47.

Step 1 of the $N^{\frac{1}{3}}$-way $N^{\frac{1}{3}}$-merge $N^{\frac{1}{3}}$-tonic algorithm can be regarded as sorting blocks of sub-arrays of size $N^{\frac{2}{3}}$ in an alternating manner (the input vector is regular-$N^{\frac{1}{3}}$-tonic). By this we mean that consecutive sub-arrays will be sorted in increasing order, then decreasing order. Since we are sorting blocks in parallel, we can apply the odd-even transposition sort, which takes $N^{\frac{2}{3}} + N^{\frac{1}{3}}$ time cycles. Next we sort the columns of size $N^{\frac{2}{3}}$, again this can be sorted by the odd-even transposition sort in $N^{\frac{2}{3}} + N^{\frac{1}{3}}$ time cycles. Then we need to sort the rows of size $N^{\frac{1}{3}}$ - this can be done in $N^{\frac{1}{3}}$ time cycles. From Theorem 3.4.3, we know that there are at most $\lceil \frac{N^{\frac{1}{3}}}{2} \rceil$ dirty rows in the resulting array, which can be sorted using the odd-even transposition sort in $\frac{N^{\frac{2}{3}}}{2} + N^{\frac{1}{3}}$ time cycles. Hence this algorithm will sort in $2.5N^{\frac{2}{3}} + O(N^{\frac{1}{3}})$ time cycles.

Similarly, the time complexity of the $N^{\frac{1}{3}}$-way $N^{\frac{1}{3}}$-merge modulo algorithm on page 47, can be shown to sort in $2.5N^{\frac{2}{3}} + O(N^{\frac{1}{3}})$. Since the only difference between these two algorithms is the initial process of transferring the sorted subvectors (the input vector is $N^{\frac{1}{3}}$-increasing). ∎

The time lower bound for sorting on an $N^{\frac{2}{3}} \times N^{\frac{1}{3}}$ array, where every processor has at most 4 connections to its immediate grid neighbours without the wraparound connections, is shown to be $2N^{\frac{2}{3}} + N^{\frac{1}{3}}$ in [89]. Our new generalized algorithm is $0.5N^{\frac{2}{3}}$ away from optimal, hence it is an efficient algorithm. Unlike the $3n$ algorithm introduced in Schnorr and Shamir [89], our algorithm does not have very large lower order term either. The modified column sort of Parker and Parberry [75] will not do as well as our algorithm, because the algorithm needs to diagonalize the array, which will introduce a time delay of at least $N^{\frac{2}{3}}$.

We conclude this chapter by investigating a recent paper: Sibeyn [91] has formulated various merge algorithms for the multiple instruction streams multiple data streams architecture (MIMD). These merge algorithms are designed to sort square interconnection networks in an iterative manner, where the scheme partitioning the large square interconnection network into number of square connected sub-networks and then those square blocks are sorted in parallel. The algorithm will output sorted row major index arrays.

The characteristic property of the processor unit in the MIMD model is that each processor unit is capable of storing a fixed number of data (which is called the queue of a processor unit). The merge algorithms of Sibeyn [91] take advantages of the fact that the processor units are able to queue data, to formulate their algorithms. In the algorithm scheme, any data from the sub-blocks around the boundary of the larger square interconnection network is shifted to the queue of the processor units in the centre. Those algorithms can be described basically as follows. After every sub-block is sorted in parallel: The data of the sorted blocks is moved following a prescribed scheme and then stored in the queue of the processor units in the centre of the square interconnection network, along the direction of the rows: the section of processor units that contains data is called the column bundle. Next, the columns of the column bundle are sorted in parallel. Therefore it is clear after the data is shifted, that the algorithm is still sorting a rectangular interconnection network, like our q–way r–merge algorithm. However, we want to stress that the similarity between Sibeyn's algorithm scheme and our q–way r–merge algorithm ends here. After step 2, our algorithm applied the transformer stage then completes one iteration of the merge sort, whereas Sibeyn's algorithm scheme does the following: the algorithm copies the t smallest packets of every processor unit to its upper neighbour and the t largest to its lower neighbour. Sections of width s of the rows are sorted, and then the algorithm throws away the ts smallest and largest packets. Some of the execution steps in the scheme of Sibeyn can be done in a pipeline fashion. Although our algorithm was not executed in a pipeline fashion, nevertheless our algorithm does not require the processor units to have the ability to perform any special task, except that the processor units must be able to function as the k–comparators.

The one of Sibeyn's algorithms that is related to our model is the "merge algorithm" in [91] merges m^2 sorted submeshes into row major order on an $n \times n$ interconnection network (assume that n is a power of 2). Then Sibeyn implements the merge sort in the following way: the sorting algorithm sorts the 2×2 submeshes and then partitions them into groups and merges the four 2×2 submeshes in each group to form the 4×4 sorted submesh. This process is continued for $2^i \times 2^i$ submeshes, until we sort the whole $n \times n$ interconnection network. Using the splitter route method (partially sorting and routing the elements of the mesh to the appropriate sections of the mesh), the sorting can be performed in $2.5n + O(n^{\frac{5}{6}})$. This time complexity has higher lower term time complexity than our q–way r–merge sorting on the rectangular interconnection networks.

The algorithm scheme of Sibeyn [91] is relevant to our q–way r–merge algorithm, because the underlying principle of the algorithm scheme is very similar to the one we used for the our q–way r–merge

algorithm. but the algorithms of Sibeyn [91] are constructed to be implemented on a MIMD architecture machine, whereas our q–way r–merge algorithm is constructed to be implemented on any machine that is capable of emulating the k–comparator network. It is clear from the reasons given above that these two merge schemes will be implemented in very different manners: The merge scheme of Sibeyn [91] is designed in the manner that is best suited for the square mesh interconnection networks with sorting processors which are capable of storing data, whereas our merge scheme is designed to sort on rectangular interconnection networks with any type of processor. Furthermore, as we have stated in the introduction of this chapter (see page 22), our merge scheme puts the emphasis on finding an elegant merge sorting paradigm/technique, which can be implemented on any underlying processing architecture. Since Sibeyn's scheme is designed with few types of processing models in mind, in particular the MIMD model. and the underlying architecture used by Sibeyn is so different to the one we used, the results derived in Sibeyn [91] in fact complement the work we have presented in this chapter.

Chapter 4

The Generalization of the Merge Algorithm and Sorting Networks

In this chapter, we will investigate an efficient implementation of sorting networks using the technique of decomposition chains. The common measures of efficiency of a sorting network are its depth and size. In this respect, the Ajtai, Komlos and Szemeredi's sorting networks [4], whose depth is $O(\log n)$, are the best networks for very large n. They are not usable for practical values of n due to the large constant associated with the O notation. So, the best practical networks are still the Batcher's sorting networks [8] invented in the 1960's.

In the previous chapter, we introduce four merge algorithms, which are generalizations of the bitonic sorting algorithm and the odd-even merge algorithm. Those merge algorithms sort a combination of different decomposition chains with rewiring, and then an appropriate transformer is applied (with respect to the corresponding chain). In this chapter we will generalize this procedure and get a class of algorithms (in a similar style to that in Becker, Nassimi and Perl [9]), where they will need no rewiring when implemented on the k–comparator networks. Since the algorithms in this chapter do not need rewiring, they are more readily converted to sort on higher dimension interconnection networks (as demonstrated in the last section of this chapter).

The networks in this chapter can be regarded as the generalization of the balanced network of Dowd et al [34] and the collection of networks constructed in Becker, Nassimi and Perl [9].

The algorithms in the previous chapter need rewiring when implemented on the k–comparator networks, since some procedures in those algorithms need to produce output with sections of its elements in decreasing order, but the k elements outputted by any k–comparator will be sorted in increasing order, thus we will need to re-route the data elements. Rewiring of the connections from a set of outputs of k–comparators to the next set of inputs of the next level of k–comparators will achieve the required

permutation, when sections of the elements of the output need to be in the reverse order.

Another difference between the class of merge algorithms from the previous chapter and the class of merge algorithms from this chapter is that different types of decomposition chains are used in them. In the previous chapter, we show that if the input N–vector is either r–increasing or r–tonic, then the multi-way multi-merge algorithm will output a sorted N–vector. Now we can express this in terms of procedures on the decomposition chain: if the input N–vector V has r sorted $quotient(\frac{N}{r})$–chains, and the q $\frac{N}{q}$–chains of V are then sorted, then applying the N–k–cotransformer will output a sorted N–vector. A similar scheme of constructing merge algorithms is used in this chapter also, however, we require the input N–vector V has q sorted $\frac{N}{q}$–chains, and the q $\frac{N}{q}$–multiple generalized chains of V are then sorted, then applying an appropriate N–k–transformer will output a sorted N–vector. The different types of $\frac{N}{q}$–m-g-chain used in the scheme are the p–cochain, the p–bichain and alt-q–bichain.

The first algorithm that we investigated is the cochain merge algorithm of Becker and Litman [10], it is a generalization of the balanced sorting algorithm introduced on page 31. We then proceed to generalize the g-chain periodic sorting algorithm of Becker, Nassimi and Perl [9], where we introduce three new sorting algorithms for rectangular grids, from which we will construct a class of sorting algorithms. Furthermore, we will show that the merge algorithms use fixed size comparators, the q^2–comparators, to sort any N–vector, where N is a power greater than 2 of q.

To start with, we explore different ways of partitioning a $p \times q$ grid (where $p = q^{s-1}$ for some $s \geq 2$) into subvectors of equal size, where the size of a vector is the number of its components. We will number the components of an N–vector from 0 to $N-1$, and call these component numbers the indices of the vector. We express the indices in their q–ary expansion.

We will concentrate on a few types of decomposition chains using the multiple generalized chain decomposition, which is defined in Definition 4.2. In the following sections of this chapter, we will present four different merge algorithms. With the exception of the bichain merge algorithm (using the bitransformer) which uses five levels of k–comparators when sorting a $k \times k^{\frac{1}{2}}$ rectangular array, the other three algorithms use four levels of k–comparators. Note: a different bichain merge algorithm (using the skewtransformer) uses four levels of k–comparators and this removes the extra level used in the bichain merge. These four algorithms can be described by the following paradigm: each algorithm takes the input N–vector and sorts its p–chains (columns of A). Next, the algorithm proceeds to sort a set of multiple generalized chains, then the resulting N–vector is sorted by an appropriate transformer (with respect to the specified type of multiple generalized chain). We want to emphasis here following this paradigm that not every set of the multiple generalized chain will result in a sorting network. We will illustrate this in the last section of this chapter by constructing one such decomposition chain that is not suited for this paradigm.

We will prove the correctness of the merge algorithm using a decomposition tree. An in-depth invest-

igation into the merge algorithm using the p–cochains and the corresponding network can be found in the paper of Becker and Litman [10]. We will incorporate their result into our generalized merge algorithm. We then construct a sorting algorithm, using the family of merge algorithms.

In Bilardi and Preparata [20], a generalization of a known class of parallel sorting algorithms is presented, together with a new interconnection to execute them. Our generalization is different from their method, but we can implement our class of algorithms using similar interconnecting of trees. Furthermore, they based the merge algorithm on pair-wise merging, unlike our new merge algorithm which can be implemented using any sorting algorithm that efficiently sorts q^2–vectors.

In the previous chapter, we have put a lot of effort into distinguishing the grid algorithm approach and the k–comparator network approach. In this chapter, we will reap from the knowledge that we have gained from the previous chapter and put less emphasis on distinguishing those two approaches of our paradigm. In particular, for each algorithm constructed, the results will be shown in terms of procedures using the underlying grid algorithm or the k–comparator network interchangeably. The choice of which approaches to take will depend on which method gives the most concise and clear proof.

This chapter is organized in the following way: In section 1, we define *multiple generalized chains* and introduce some of the properties of this family of decomposition chains. From section 2 to section 5, we present the grid merge algorithms that sort $p \times q$ grids. In sections 6 and 7, we introduce the merge algorithm and the sorting algorithm in terms of the k–comparator networks. In section 2, we investigate the p–cochain merge algorithm of Becker and Litman [10], we describe the algorithm in terms of three operations on a $p \times q$ grid. This merge scheme will be used in all the algorithms that will be introduced: first a set of decomposition chains is sorted (such as the cochain, the bicahin and the alt-q–bichain), followed by a "transformer" stage that consists of two steps of operations (it can be regarded as a k–comparator sub-network, hence we call it the transformer stage). If the p–chain (the columns) of the grid are sorted, then the algorithm sorts the grid into row major ordering. In section 3, we introduce the p–bichain merge algorithm. We state the p–bichain merge algorithm in terms of four operations on a $p \times q$ grid. The p–bichain merge algorithm sorts any $p \times q$ grid, if the columns are sorted initially. In section 4, we introduce a new type of *multiple generalized chain* that is a variation of the p–bichain, the alt-q–bichain. The aim is to construct a merge algorithm that uses fewer operations than the p–bichain merge algorithm. In fact, the alt-q–bichain merge algorithm uses three operations on a $p \times q$ grid. In section 5, we investigate an alternative method of improving the complexity of the p–bichain merge algorithm: since the extra operation needed in the p–bichain merge algorithm occurred in the "transformer" stage of the algorithm, we will now modify the transformer stage by introducing a new type of transformer. Hence this modified p–bichain merge algorithm will sort using three grid operations. In section 6, we formulate the G-Merge algorithm in terms of k–comparator networks, where we use the p–cochain merge algorithm, the alt-q–bichain merge algorithm and the modified p–bichain merge algorithm as the underlying algorithms. We then analyze the time complexity of the family of merge networks

constructed from the G-Merge algorithm. In section 7, we introduce the class of sorting algorithms. We formulate the sorting algorithm by applying the G-Merge algorithm iteratively. In section 8, we discuss and clarify some of the issues that we have not covered - such as the time complexity of implementing these algorithms on 2-dimensional interconnection networks, and a discussion on implementing the algorithms on 3-dimensional interconnection networks.

4.1 Multiple Generalized Chains

In this section we will introduce a family of decomposition chains, which we use to partition N-vectors. In this chapter, we assume that all N-vectors are of size $N = q^s$ for some positive integers q and s (Note: the corresponding grid array A will have dimension $p \times q$, where $p = q^{s-1}$).

In the previous chapter, we defined the p-chains of V (see Definition 3.1), however, the same definition can be recursively defined on each p-chain. So we have the next definition (see Figure 4.1 for an illustration of this recursive definition of p-chains):

Definition 4.1 *The* (q,t)-**Chains**, *where* $1 \leq t \leq s-2$: *The* q^{s-t} *subvectors of the* V, *where the* q^t-*subvectors are just the* (q,t)-*chains of the* $(q,t+1)$-*chains of* V. *Thus the elements belonging to the same chain are precisely those that have the same last* $s-t$ *digits in their* q-*ary expansion.*

$v_0\ v_1\ v_2\ v_3\ v_4\ v_5\ v_6\ v_7\ v_8\ v_9\ v_{10}\ v_{11}\ v_{12}\ v_{13}\ v_{14}\ v_{15}\ v_{16}\ v_{17}\ v_{18}\ v_{19}\ v_{20}\ v_{21}\ v_{22}\ v_{23}\ v_{24}\ v_{25}\ v_{26}$

A 3^3-vector.

$v_0\ v_3\ v_6\ v_9\ v_{12}\ v_{15}\ v_{18}\ v_{21}\ v_{24}$
$v_1\ v_4\ v_7\ v_{10}\ v_{13}\ v_{16}\ v_{19}\ v_{22}\ v_{25}$
$v_2\ v_5\ v_8\ v_{11}\ v_{14}\ v_{17}\ v_{20}\ v_{23}\ v_{26}$

3 $(3,2)$-chains of the 3^3-vector.

$v_0\ v_9\ v_{18}$ $v_3\ v_{12}\ v_{21}$ $v_6\ v_{15}\ v_{24}$: 3 $(3,1)$-chains from the 1^{st} $(3,2)$-chain.
$v_1\ v_{10}\ v_{19}$ $v_4\ v_{13}\ v_{22}$ $v_7\ v_{16}\ v_{25}$: 3 $(3,1)$-chains from the 2^{nd} $(3,2)$-chain.
$v_2\ v_{11}\ v_{20}$ $v_5\ v_{14}\ v_{23}$ $v_8\ v_{17}\ v_{26}$: 3 $(3,1)$-chains from the 3^{rd} $(3,2)$-chain.

9 $(3,1)$-chains of the 3^3-vector.

Figure 4.1: An example to illustrate Definition 4.1, on a 27-vector.

It is clear from Definition 4.1, that under this recursive definition, the p-chains of an N-vector defined in Definition 3.1 are just the set of $(\frac{N}{p}, \log_{\frac{N}{p}}(p))$-chains. However, using the recursive notation to denote

set of p–chains can be cumbersome, so to simplify our representation, sometime we will continue to use the notation defined in Definition 3.1 in the case where there is no ambiguity.

Using this recursive definition of the p–chains in Definition 4.1, we are able to define a family of decompositions, called the *multiple generalized chains* on V. We continue to use the terminology of the generalized chains defined in Becker, Nassimi and Perl [9], which gives two subvectors only, compared to our case of q subvectors. For each given type of the m-g-chains decomposition, one is able to specify the components as being those components of the original vector having a certain relationship among the bits of q–ary representation of their indices. In other words, the partition is according to some systematic rules that can be applied to an arbitrary vector of size q^s.

Definition 4.2 *A set of q decomposition chains of a q^s-vector V is called the m-g-chains of V, if the set of the $(q, s-2)$-chains generated from those q q^{s-1}-subvectors are exactly the set of q^2 $(q, s-2)$-chains of V. An m-g-chain of V has q^{s-1} elements.*

The multiple generalized chain decomposition partitions the $p \times q$ grid into groups of p–subvectors, where the $(q, s-2)$–chains of those p–subvectors are the $(q, s-2)$–chains of the $p \times q$ grid. In fact, there are $O(\frac{q^2!}{(q^2-q)!q!}) = O((\frac{q^2}{q}))$ different ways to partition V into groups of q $(q, s-2)$–chains. Any such a group of subvectors that equally divides a vector will be called a set of multiple generalized chains. So intuitively the m-g-chains decomposition of V is just a scheme, which partitions V into groups of q $(q, s-2)$–chains. Now we will define two types of m-g-chains of V:

Definition 4.3

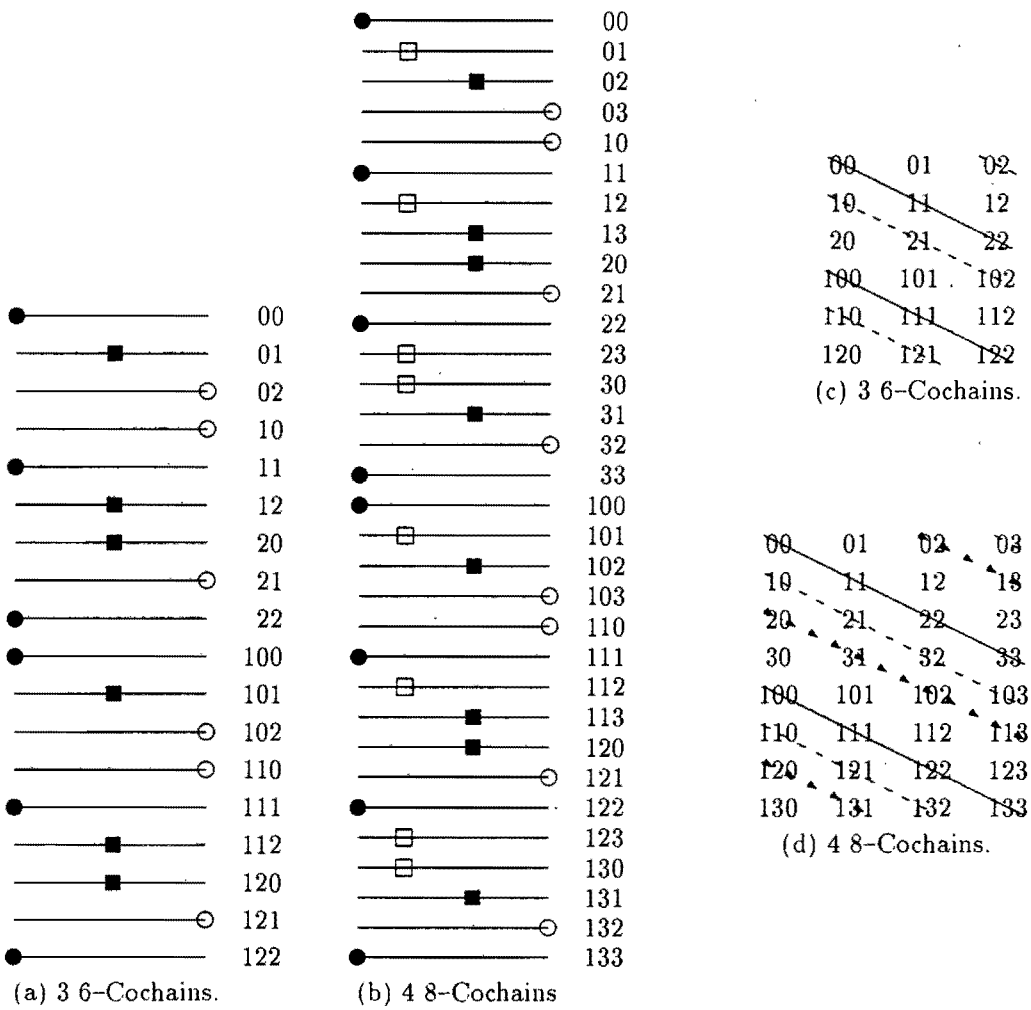
1 *The p–**Cochains**: There are q such subvectors in V:*

$$(v_{00}, v_{11}, v_{22}, \ldots v_{(q-1)(q-1)}, \ldots)$$
$$(v_{01}, v_{12}, v_{23}, \ldots v_{(q-1)q}, \ldots)$$
$$\vdots$$
$$(v_{0i}, v_{1(i+1)}, v_{2(i+2)}, \ldots v_{(q-1)(i+q-1)}, \ldots)$$
$$\vdots$$
$$(v_{0(q-1)}, v_{1q}, v_{2(q+1)}, \ldots v_{(q-1)(2q-2)}, \ldots).$$

A p-cochain has p elements. Hence the i^{th} p-cochain, $(0 \leq i \leq q-1)$, is the subvector of V whose two least significant bits s t in the q-ary expansion of its index satisfy $t - s = i \pmod q$.

2 *The p–**Bichains**: There are q such subvectors in V, where the elements belonging to the same bichain are precisely those that have the same second last digit in their q-ary expansion. A p-bichain has p elements.*

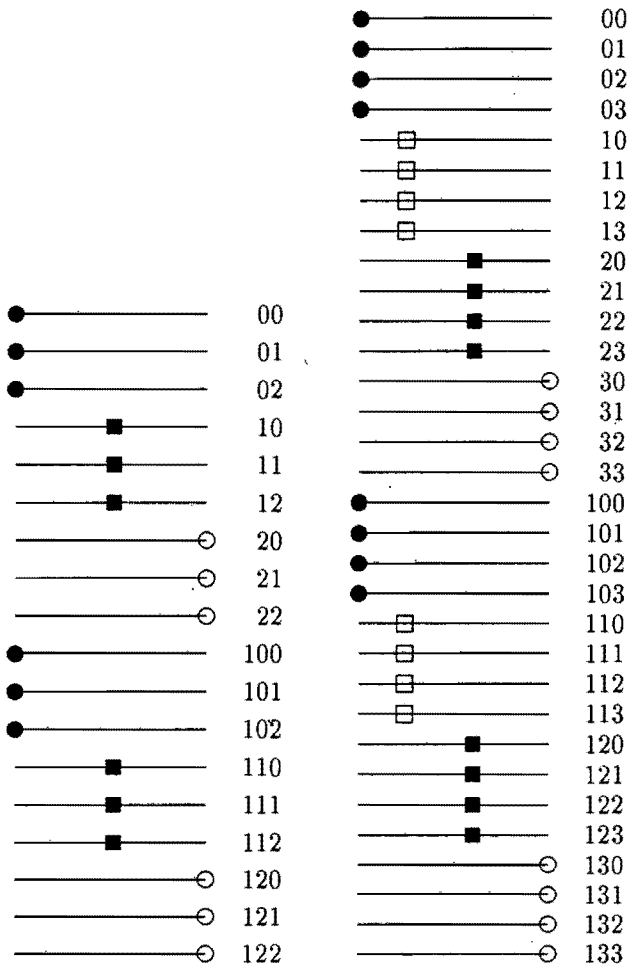
Observation: A natural way to index a set of generalized chains (from 0 to $q-1$) is to use their characteristic representation in the q-ary expansion. We also want to emphasize that if V is a square array ($q \times q$), then the *quotient*(q)-Chains defined in Definition 3.4 are exactly the same as the q-bichains defined in Definition 4.3 here.



(a) 3 6-Cochains. (b) 4 8-Cochains (c) 3 6-Cochains. (d) 4 8-Cochains.

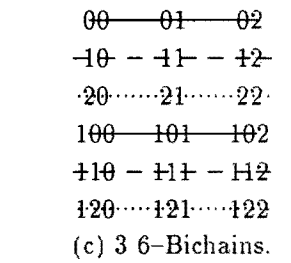
Figure 4.2: Some illustrations of the p-cochains: (a) 3 6-cochains of an 18-vector, (b) 4 8-cochains of a 32-vector, (c) 3 6-cochains on a (6×3) grid. (d) 4 8-cochains on an (8×4) grid.

It is clear from Definition 4.2, that the p-chains of Definition 3.1, the p-bichains and the p-cochains of Definition 4.3 are sets of m-g-chains decomposition of V. This is because the $(q, s-2)$-chains of any of the above decomposition chains are $(q, s-2)$-chains of V. This can be observed in the grid representation of each type of chains, where repeated pattern occurs in the q^2 sub-arrays of each chain. (See Figure 3.2, Figure 4.2 and Figure 4.3 for the illustrations.)

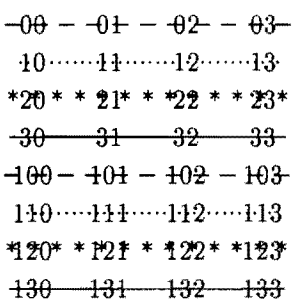


(a) 3 6–Bichains.

(b) 4 8–Bichains



(c) 3 6–Bichains.



(d) 4 8–Bichains.

Figure 4.3: Some illustrations of the p–bichains: (a) 3 6–bichains of an 18–vector, (b) 4 8–bichains of a 32–vector, (c) 3 6–bichains on a (6×3) grid. (d) 4 8–bichains on an (8×4) grid.

Using Definition 4.2, one can further partition each m-g-chains recursively using any chosen type of m-g-chains decomposition. If this process is continued recursively into the smaller m-g-chains, then V is partitioned into smaller subvectors of equal length. We will use this embedded structure (for example, the $(q, s-i-2)$–chains of the $(q, s-i-1)$–chains of V) to construct the merge algorithm, using the property proven in the next lemma, Lemma 4.1.1.

Lemma 4.1.1 *If a q^s–vector V is partitioned recursively, using any combination of different types of m-g-chains, into q^{s-i}–subvectors $(1 \leq i \leq s-2)$, then the $(q, s-i-1)$–chains of those q^{s-i}–subvectors are the $(q, s-i-1)$–chains of V.*

Proof: We will prove this lemma by induction on the number of iterations of decomposition. It is trivially true for $i = 1$ using Definition 4.2 of the m-g-chains which implies that the $(q, s-2)$–chains of the m-g-chains are the $(q, s-2)$–chains of V.

Now, we assume that it is true for all $i \leq k$, where $1 \leq k \leq s-2$. Next, for $i = k+1$, we decompose the m-g-chains at the $(k+1)^{th}$ iterations. At the $(k+1)^{th}$ iteration, each q^{s-k}–subvector is partitioned into q q^{s-k-1}–subvectors, using any type of m-g-chains decomposition. By Definition 4.2, the $(q, s-k-2)$–chains of those q^{s-k-1}–subvectors are $(q, s-k-2)$–chains of the q^{s-k}–subvector. However, the induction hypothesis implies the $(q, s-k-1)$–chains of the q^{s-k}–subvector are the $(q, s-k-1)$–chains of V, hence the $(q, s-k-2)$–chains of the q^{s-k}–subvector are the $(q, s-k-2)$–chains of V.

Hence it is true for all values of $1 \leq i \leq s-2$. ∎

4.2 The p–Cochain Merge Algorithm

In this section, we will introduce Becker and Litman's [10] algorithm, which uses a recursive structure to sort the rectangular grids (sort under the row major indexing scheme). We will show that the p–cochain algorithm sorts any $p \times q$ grid, where $p \geq q(q-1)$ and p is divisible by q. However, we will leave the analysis and the construction of the merge algorithm using the multiple generalized chains to a later section of this chapter. As we have indicated before, we shall use this algorithm of Becker and Litman's [10], combined with all the new algorithms introduced in this chapter, to construct a class of new merge algorithms and sorting algorithms.

We introduce the next algorithm in terms of operations on a rectangular $p \times q$ grid and if the columns (the p–chains) of the grid are sorted, then the resulting grid will be sorted in row major indexing. Here we state the p–cochain algorithm from Becker and Litman's [10]:

The p–Cochain Merge Algorithm

1. Sort the q p–cochains.
2. Sort the rows in groups of $q-1$, starting from Row_0.
3. Sort the rows in groups of $q-1$, starting from $\text{Row}_{\lfloor \frac{q}{2} \rfloor}$.

It is quite clear (as we have observed in the previous chapter), that the last two steps of the p–cochain algorithm of size N can be implemented by an N–k–cotransformer (where $k = q(q-1)$).

We have illustrated in Figure 3.1 that we can regard an N–vector V and a grid A as representing the same data structure interchangeably. Thus it is not difficult to see that a decomposition chain is well defined for both representations. In case of the cochains, we use Figure 4.2 to illustrate how a cochain can be represented in the line representation (in terms of partitioning the N–vector V) and the grid representation.

Figure 4.4 is used to emphasize how the line representation and the grid representation can be used interchangeably. It demonstrates how the operations of the cochain merge algorithm are applied on a $p \times q$ grid, and how this can be related to a p–comparator network of size $p.q$. It is simpler to construct the merge algorithm in terms of k–comparators than on a grid, henceforth in this chapter, we shall use the line representation to illustrate a merge algorithm diagrammatically. Hence in Figure 4.5 we gave an example of the p–cochain merge algorithm on a 32–vector, in terms of the line representation.

We will show the correctness for the p–cochain algorithm. All the proofs and theorems for the p–cochain algorithm are from Becker and Litman [10]. First we can make the following observations in Lemma 4.2.1.

Lemma 4.2.1 *For any N–vector V, the following properties hold:*

(a) *Each element v of V belongs to exactly one p–chain and exactly one p–cochain.*

(b) *Each p–chain and each p–cochain has exactly one element in each row (in the grid representation), except possibly that there may be no element in the final row.*

(c) *Each element of a p–chain lies on the same column in each row (in the grid representation).*

(d) *An element at row r, where $r \geq q-1$, has the preceding $q-1$ elements of its p–cochain on different lines, and this q–line pattern repeats itself.*

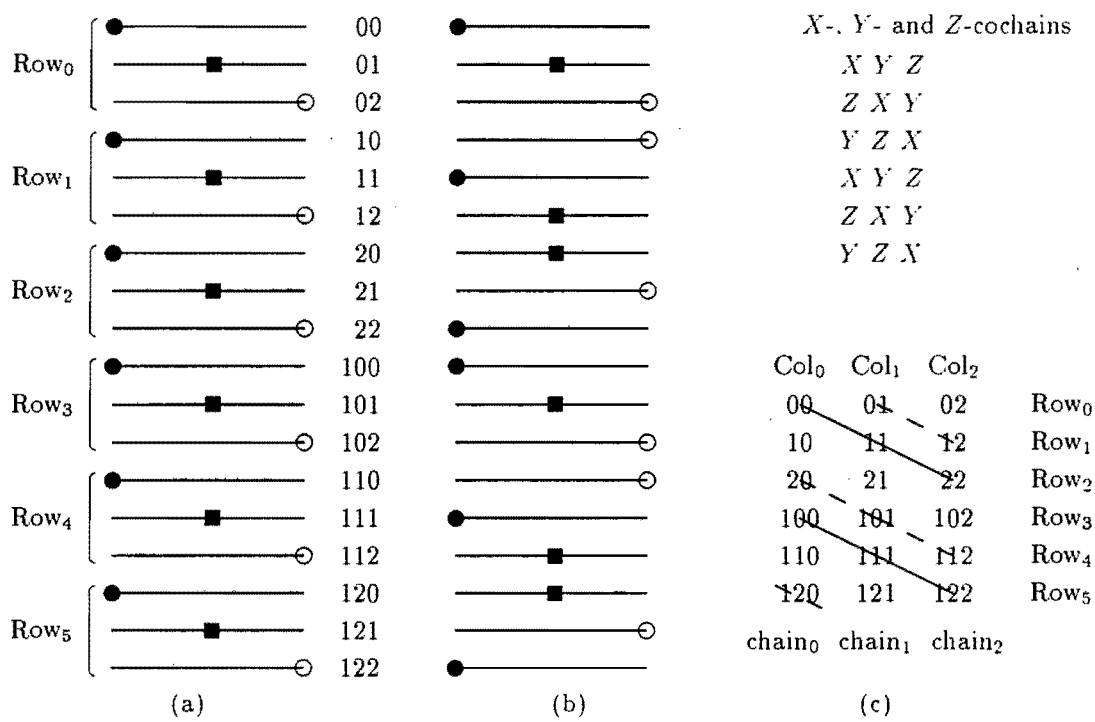


Figure 4.4: p–Chains and p–cochains, $q = 3$, where (a), (b) are in terms of the line representation and (c) is in terms of the grid representation. (a) 3 6–chains of an 18–vector, (b) 3 6–cochains of an 18–vector. (c) 3 6–cochains along diagonals of the grid, 3 6–chains along verticals of the grid.

Note: The proof of Lemma 4.2.1 will be omitted, however, it is apparent from the illustration of Figure 4.4, that Lemma 4.2.1 is trivially true.

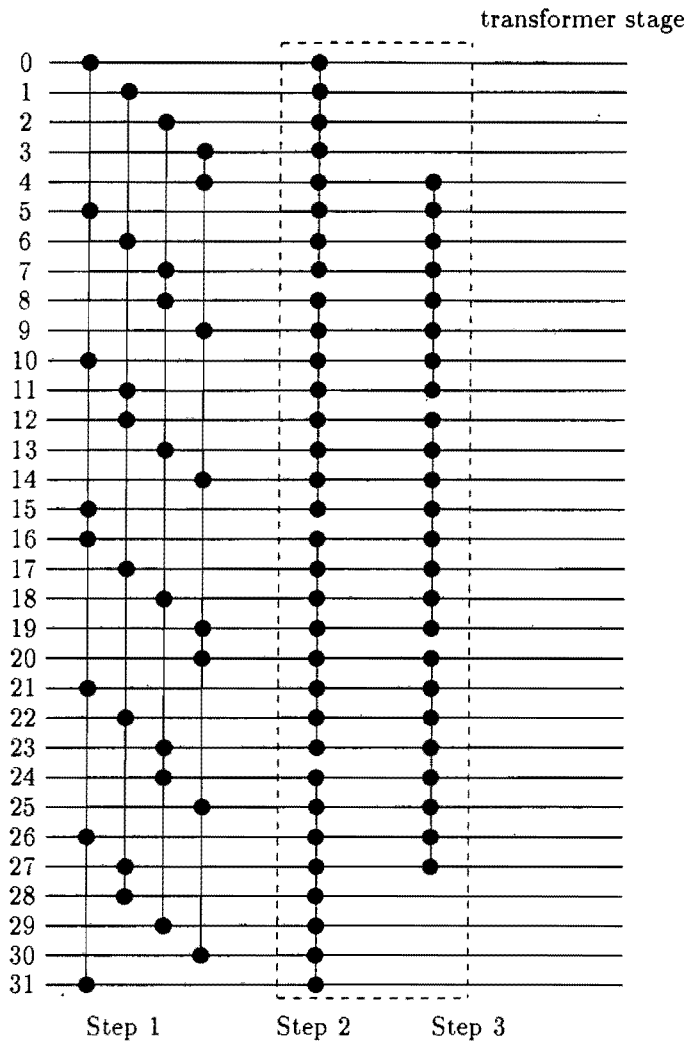


Figure 4.5: A line representation of the p–cochain merge, on a 32–vector

Lemma 4.2.2 *Let* $\{a_i\}$, $\{b_i\}$ *be two sequences of the same length, each sorted in increasing order. Then the sequences* $\{\min\{a_i, b_i\}\}$, $\{\max\{a_i, b_i\}\}$ *are also sorted in increasing order.*

Proof: From the inequalities $a_i \leq a_{i+1}$ and $b_i \leq b_{i+1}$, it follows easily that $\min\{a_i, b_i\} \leq \min\{a_{i+1}, b_{i+1}\}$ and $\max\{a_i, b_i\} \leq \max\{a_{i+1}, b_{i+1}\}$. This implies the result. ∎

Theorem 4.2.3 *If V has sorted p-chains, and the p-cochains of V are then sorted, then the p-chains remain sorted, and the vector then has both sorted p-chains and sorted p-cochains.*

Proof: Suppose the p-chains of V are sorted. We will give a procedure for sorting the p-cochains which at the end of each step leaves the p-chain sorted, and the theorem will follow from this. We will not in future use this procedure to sort the p-cochains, however. The procedure consists of q stages, in each of which a number of pairs of elements (each pair from the same cochain) are sorted, and these q stages are repeated until the cochains are sorted. The stages are illustrated in Figure 4.6.

Stage 1 sorts the first pair of elements of the p-cochain starting with v_{00}; then the first pair of elements of the p-cochain starting with v_{10}; ...; then the first pair of elements of the p-cochain starting with $v_{(q-1)0}$. For each cochain, only complete groups of two are sorted, and any group of less than two remaining is left unsorted. Stage 2 proceeds similarly for the p-cochains containing $v_{01}, v_{11}, \ldots, v_{(q-1)1}$, we start to sort the pair of elements starting with these elements in each cochain. We proceed similarly all the way to Stage q, where Stage q proceeds similarly for the p-cochains starting with $v_{0(q-1)}, v_{1(q-1)}, \ldots, v_{(q-1)(q-1)}$. The Figure 4.6 illustrates this process for $q = 3$. Each consecutive pair in each p-cochain is sorted at some stage, and hence repeating the process a sufficient number of times will sort each cochain.

We now show that each stage leaves the chain sorted. Each stage can be regarded as taking two sorted columns and comparing element i of Col_0 with element $i+1$ of Col_1; the smaller element goes to Col_0. There is one unsorted element at the bottom of Col_0 and one at the top of Col_1 . By Lemma 4.2.2 with $q = 2$, consecutive pairs of compared elements will still be in their correct order after the comparison. Hence the compared elements in each column are in their correct order. What remains is to check the two uncompared elements. Let a be the last compared element in Col_0 before comparison, a' the element in this position after the comparison and b the uncompared element below it. We have $a' \leq a \leq b$ so we have $a' \leq b$ and Col_0 is sorted. Similarly, Col_1 is sorted. Thus each stage leaves all the columns sorted, and thus they are still sorted after all the cochains are sorted at the end of the procedure. ∎

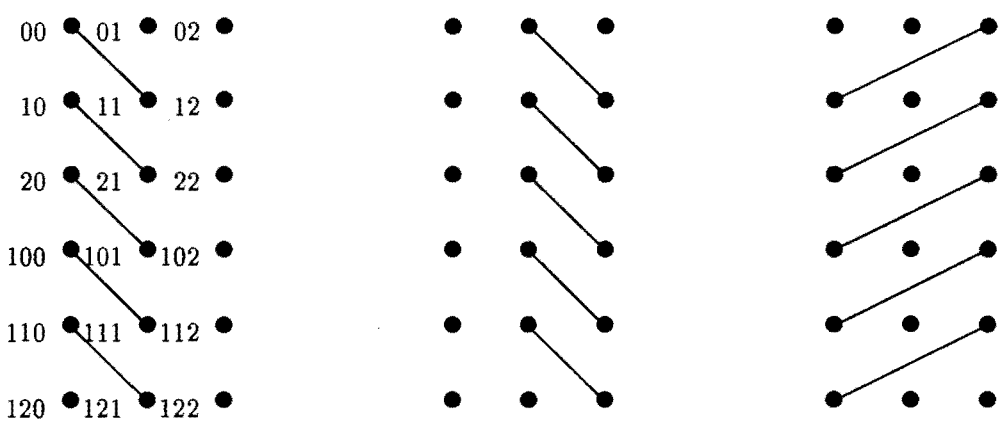


Figure 4.6: Stages in the proof of Theorem 4.2.3, $q = 3$.

Lemma 4.2.4 *Let $A = (a_{ij})_{i \in Z, j \in \{1,\ldots,q\}}$ be an infinite* **0–1** *array such that A is neither all 0's nor all 1's and the columns and the diagonals of A are sorted. Let $V = (v_k)_{k \in Z}$ be an infinite* **0–1** *vector such that $v_{qi+j} = a_{ij}$ for all i and j. Then there is an interval I of Z such that*

1. $|I| = q(q-1)$
2. $v_k = 0$ *for all* $k < \min(I)$
3. $v_k = 1$ *for all* $k > \max(I)$
4. *The subvector $(v_k)_{k \in I}$ has an equal number of 0's and 1's.*

Proof: Let $t = \max\{i : (\exists j) a_{ij} = 0\}$ and $s = \min\{i : (\exists j) a_{ij} = 1\}$ (since the columns and the diagonals are sorted, and the array has at least one 0 and at least one 1, the quantities s and t exist).

By definition, the Row_t of A has at least one 0. Since the columns and the diagonals of A are sorted, the Row_{t+1} has at least two 0's, and the Row_{t-n} has at least $n+1$ 0's for $0 \leq n < q$. Hence all rows with index less than $q-1$ have all entries 0's. This implies $t - q + 1 < s$, that is, $t - s + 1 < q$. By the same argument, the Row_{t+n} has at least $n+1$ 1's for $0 \leq n < q$.

Let B be the sub-array of A composed of $\text{Row}_s, \text{Row}_{s+1}, \ldots, \text{Row}_t$. The number of rows of B is equal to $t - s + 1$, and by the above, this number is strictly less than q and so B has at most $q(q-1)$ entries.

Let I' be the interval of Z that corresponds to B, that is, $\min(I') = qs + 1$ and $\max(I') = qt + q$. Clearly, $v_k = 1, \forall k > \max(I')$ and $v_k = 0, \forall k < \min(I')$.

Let I be an interval such that $I' \subset I$ and $|I| = q(q-1)$. If $\min(I) = \min(I')$ then, by the above argument, the number of 1's in the subvector $(v_k)_{k \in I}$ is at least $\sum_{i=1}^{q-1} i = \frac{q(q-1)}{2}$. Similarly, if $\max(I) = \max(I')$ then $(v_k)_{k \in I}$ is at least $\sum_{i=1}^{q-1} i = \frac{q(q-1)}{2}$ 0's. Hence there is an I for which $(v_k)_{k \in I}$ has property (d) of Lemma 4.2.1. ∎

Since we have realized that the last two steps of the p–cochain algorithm can be implemented on an N–k–cotransformer ($k \geq q(q-1)$), we proceed to state and prove the next theorem in terms of the k–comparators. The proof given in terms of the k–comparators is much more concise and gives a clearer overview of the theorem.

Theorem 4.2.5 *If V has both sorted p–chains and sorted p–cochains, then the N–k–cotransformer sorts V, where $k \geq q(q-1)$.*

Proof: It is sufficient to show that if V is an N–vector with both sorted p–chains and sorted p–cochains which is a permutation of $0 \ldots (N-1)$ then it is sorted by the N–k–cotransformer. Let $0 \leq j \leq N-1$

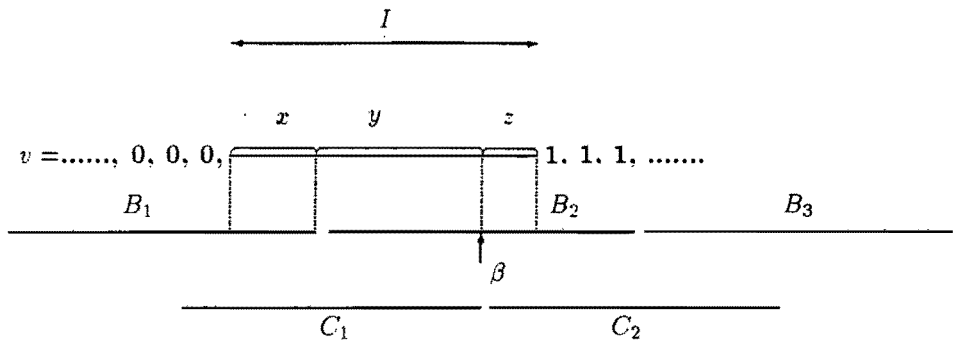


Figure 4.7: ($|I| = k$, $|y| = \frac{k}{2}$).

and let $\hat{V}^j$ be the vector with $\hat{v}_i^j = 1$ if $v_i \leq j$ and $\hat{v}_i^j = 0$ otherwise, $0 \leq i \leq N-1$. It is sufficient to show that $\hat{V}^j$ is sorted for each j, for if $j = 2$ this implies that 1 is in the uppermost place, and if all elements up to s are in their correct place, then so is the $(s+1)^{st}$. (This is the $[0-1]$ **principle**, but it is convenient here to state it in this form).

Firstly, we extend $\hat{V}^j$ to an infinite vector by adding 0's before $\hat{v}_0^j$ and 1's after $\hat{v}_{q-1}^j$; then we extend the cotransformer by adding infinitely many $\frac{k}{2}$-comparators to each layer, each comparator comparing k successive lines. This is for convenience, and clearly the new $\hat{V}^j$ is sorted by the new cotransformer if and only if the old $\hat{V}^j$ is sorted by the old cotransformer. We will not use a new notation, but regard both $\hat{V}^j$ and the cotransformer as being the extended ones from now on, $0 \leq j \leq N-1$.

By Lemma 4.2.4, there exists a subvector of length $q(q-1)$ with the same number of 0's and 1's, and the entries with smaller index are 0, while the entries with larger index are 1. Since $q(q-1) \leq k$, we may extend this vector to one of length k with the same properties. Call this subvector I. We need only show that I is correctly sorted by the network, since I is preceded by 0's and succeeded by 1's. We refer to Figure 4.7 in which I is the subvector we consider, B_1, B_2, B_3 are k-comparators in the first layer of the cotransformer, and C_1, C_2 are k-comparators in the second layer. At least one of comparators must have $\frac{k}{2}$ of its lines overlapping I, since $|I| = |B| = k$ for all comparators B in the cotransformer. Let β be the $\frac{k}{2}$ lines of B_2. We assume that β lies in the lower half of I as illustrated in Figure 4.7. (The case where β lies in the upper half of I is similar and will not be considered here). We call the $\frac{k}{2}$ elements of I overlapping with the bottom half of B_2 y, the elements of I overlapping with the top part of B_1 x, and the elements of I overlapping with the bottom half of B_2 z, as illustrated. Then $|y| = \frac{k}{2}$ and $|x|+|y|+|z| = k$ so that $|z| = \frac{k}{2} - |x|$. There are at most $|x|$ 1's in x. Hence there are $\geq \frac{k}{2} - |x| = |z|$ 1's in $y \cup z$. Hence the portion of I passed on to C_2, which is of length $|z|$, has all 1's, and hence the output of C_2 consists entirely of 1's. The portion of I passed on to C_1 is of length $|x \cup y|$, and is correctly sorted by C_1. Since the output of C_2 consists entirely of 1's, it follows that the subvector I is correctly sorted by the N-k-cotransformer. ∎

4.3 The p–Bichain Merge Algorithm

In this section, we will investigate a new method for sorting on $p \times q$ arrays. We construct a new type of the m-g-chains decomposition, the p–bichains. Then we will combine this new algorithm and the algorithm from Becker and Litman [10] to form a family of merge algorithms, using the decomposition tree. A detailed exploration of the p–cochains can be found in the paper of Becker and Litman [10] (which was introduced in the previous section). We will incorporate their result into this work, which generalizes the merge algorithm.

Again, we will state the p–bichain merge algorithm in terms of operations on a rectangular $p \times q$ grid, and if the columns (the p–chains) of the grid is sorted, then the resulting grid will be sorted in row major indexing. Here is the algorithm:

The p–Bichain Merge Algorithm

1. Sort the p–bichains, the union of all the elements from the "rows of length q" in steps of q, in the array.
2. Sort the rows in groups of q, starting from Row_0.
3. Sort the rows in groups of q, starting from $\text{Row}_{\lfloor \frac{q}{2} \rfloor}$.
4. Sort the rows in groups of q, starting from Row_0.

Observe: Step 2 to step 4 of this algorithm will corresponding to a sub-network of k–comparators ($k = q^2$), which we called the N–k–bitransformer (see Definition 3.2 for the definition of the bitransformer).

Now it is not difficult to see from Figure 4.3 that illustrate how a bichain can be represented in the line representation (in terms of partitioning the N–vector V) and the grid representation. Hence in Figure 4.8 we gave an example of the p–bichain merge algorithm on a 32–vector, in terms of the line representation.

Lemma 4.3.1 *Let V be an array of 0's and 1's, where we first sort the p–chains then we sort the p–bichains. The following will be true for any pair of the p–bichains in V:*

1. *Given the i^{th} p–bichain and the j^{th} p–bichain, where $i < j$, then the number of 1's in the i^{th} p–bichain is less than then the number of 1's in the j^{th} p–bichain.*
2. *The difference in the number of 1's between any pair of the p–bichains is at most q.*

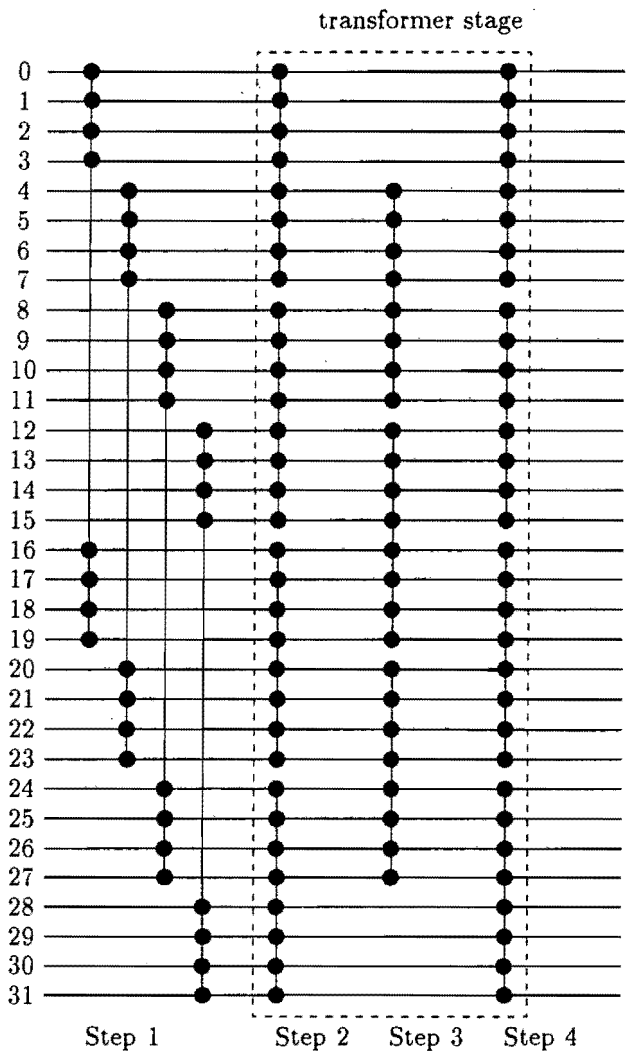


Figure 4.8: A line representation of the p–bichain merge, on a 32–vector.

Proof: After the columns are sorted (the p–chains are sorted), all the 1's will sink to the bottom of each column. From Definition 4.2 of the generalized chains, it is clear that each p–chain intersects a p–bichain $\frac{p}{q}$ $(= q^{s-2})$ times. This implies that the number of 1's in a column will be distributed evenly among the p–bichains, however, some of the p–bichains with higher indices might get one extra 1. For example, if there are r 1's in a column, then each p–bichain will get at least $\lfloor \frac{r}{q} \rfloor$ 1's. However, there might still be $(r \bmod q) > 0$ 1's left, where they are distributed among the $(q-1)^{th}$ p–bichain to the $(q-(r \bmod q))^{th}$ p–bichain. Since there are q columns in the array, the number of 1's in any pair of such sorted bichains will differ by at most q. Furthermore, the p–bichain with the higher index will have more 1's than the p–bichain with the lower index. ∎

Theorem 4.3.2 *If a $(p.q)$-vector V has sorted p-chains, and the p-bichains are then sorted, then the resulting array will have both the sorted p-chains and the sorted p-bichains.*

Proof: We will apply the $[0-1]$ **principle** here: we consider arrays containing 0's and 1's. After the columns are sorted (the p–chains are sorted), all the 1's will be in the bottom of each column. Next we will sort the p–bichains.

Assume that at least one of the p–chains is not sorted, this implies there exists a 1 in the i^{th} position and a 0 in the j^{th} position of this column, where $i < j$. Now, if $i \equiv (j \bmod q)$ then i^{th} and j^{th} entries belong to the same p–bichain, which contradicts that the p–bichains are sorted. Hence $i \not\equiv (j \bmod q)$. So the $(i \bmod q)^{th}$ p–bichain has more 1's than the $(j \bmod q)^{th}$ p–bichain. From Lemma 4.3.1, we know that the p–bichain with higher index has more 1's than the lower indexed p–bichain, this implies that $(i \bmod q) > (j \bmod q)$. But this implies that the $(i \bmod q)^{th}$ p–bichain has at least $q+1$ more 1's than $(j \bmod q)^{th}$ p–bichain, again this is not possible. Hence our assumption is false. ∎

Lemma 4.3.3 *Let $A = (a_{ij})_{i \in Z, j \in \{0,1,\ldots,q-1\}}$ be an infinite array of 0's and 1's which does not consist of all 0's or all 1's. Furthermore, we first sort the p-chains then we sort the p-bichains of A. Let $V = (v_k)_{k \in Z}$ be the infinite 0-1 vector such that $v_{qi+j} = a_{ij}$ for all i and j. Then there is an interval I of Z, such that:*

1. $|I| \leq q(q-1)$.
2. $v_k = 0$ *for all* $k < \min(I)$.
3. $v_k = 1$ *for all* $k > \max(I)$.
4. *The interval I may intersect at most q successive rows.*

Proof: Let $t = \max\{i : (\exists j)\ a_{ij} = 0\}$ and $s = \min\{i : (\exists j)\ a_{ij} = 1\}$. Since there are 0's and 1's in the array, s and t exist, after the p–chains and the p–bichains are sorted.

From Lemma 4.3.1, we know that the difference in the number of 1's between any two p–bichains is at most q.

Suppose there are n' 1's in Row_t. There are two possible cases that need to be considered:
If $(t \bmod q) \equiv q-1$, then Row_t is contained in the $(q-1)^{th}$ p–bichain. From Lemma 4.3.1, we know that the $(q-1)^{th}$ p–bichain has the most 1's among the p–bichains, hence none of the p–bichains will have more than n' 1's in the rows above Row_t. From Lemma 4.3.1, we also know that the difference in the number of 1's between any two p–bichains is at most q, this implies that $t-s-1 \leq q-2$ is the maximum number of dirty rows excluding Row_t and Row_s. In Row_s, there are at least $q-n'$ 0's at the start of the row and no more 1's before this. So we find $|I| \leq (q-2)q + n' + q - n' = (q-1)q$.

A similar argument can apply to the cases where $(t \bmod q) \not\equiv q-1$. Since Lemma 4.3.1 implies that the difference in the number of 1's between any two p–bichains is at most q. Hence the number of dirty rows is restricted to q rows, where the sum of the number of 1's in the Row_s and the number of 0's in the Row_t is bounded by q. ∎

Again, it is easier to prove the next theorem in terms of the operations using the k–comparators, instead of describing the proof in terms of the operations of the underlying grid algorithm on grid A.

Theorem 4.3.4 *The N-k-bitransformer sorts any input N-vector resulting from sorting the p-chains and then sorting the p-bichains (where $N = q^s$ and $k = q^2$).*

Proof: We use the $[0-1]$ **principle**, and consider arrays consisting of 0's and 1's. By Lemma 4.3.3, there exists a subvector of length at most $q(q-1)$. Furthermore, the subvector overlaps at most q successive rows; we will label this dirty subvector as I. Now a q^2–comparator is sufficient to cover the interval. Suppose in the first layer of the q^s–q^2–bitransformer, there is a q^2–comparator that covers I, or a q^2–comparator that covers $\frac{|I|}{2}$ to $\frac{(|I|+q)}{2}$ elements of I. In the later case, there exists a q^2–comparator in the second layer of the q^s–q^2–bitransformer covers I. Then obviously, the array will be sorted.

Otherwise one of the q^2–comparators in the first layer of the q^s–q^2–bitransformer will overlap $\frac{(|I|+q)}{2}$ elements of I. Without loss of generality, we assume the overlap occurs on the right of I (a similar argument can apply to the case when the q^2–comparator is over-lapping I on the left). We label this q^2–comparator B_1 and the q^2–comparator which is adjacent B_1 and also overlaps I is labelled B_2. We also label the q^2–comparator that overlaps B_1 and B_2 in the second layer of the q^s–q^2–bitransformer as C_1. There are two cases to be considered:

Case 1: The number of 0's in B_1 is greater than $\frac{q^2}{2}$. It is clear that the number of 1's in B_1 is greater than the number of 1's in B_2. So the number of 1's in B_2 will be less than $\frac{q^2}{2}$ (since the number of 1's in B_1 is less than $\frac{q^2}{2}$). So after applying the q^2–comparators in the first two layers of the q^s–q^2–bitransformer, every 1 will be covered by the q^2–comparators which are in the position of B_1 and to the

right of B_1. Therefore the extra copy of the first layer of the q^s-q^2-cotransformer will sort the array (The corresponding q^2-comparators, B_1' and B_2'). See Figure 4.9 for the illustrations.

Case 2: The number of 0's in B_1 is less or equal to $\frac{q^2}{2}$. After applying the q^2-comparators in the first layer of the q^s-q^2-bitransformer, one of the comparators in the second layer of the q^s-q^2-bitransformer C_1 will overlap the dirty interval of the array, because C_1 overlaps B_1 by $\frac{q^2}{2}$. Thus the second layer of the q^s-q^2-bitransformer will able to sort the array.

Hence the N-k-bitransformer sorts any N-vector, where $N = q^s$ and $k = q^2$. ∎

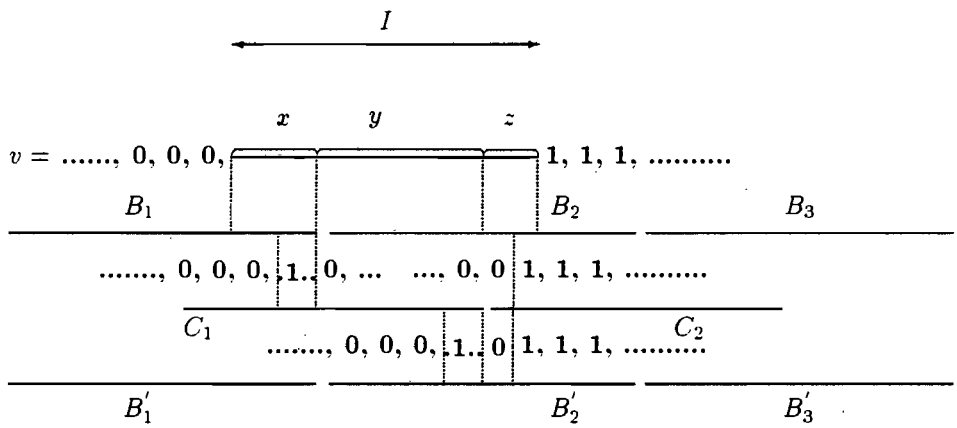


Figure 4.9:

From the proof above, we see that the N-k-bitransformer is needed for the p-bichain algorithm. Furthermore, the N-k-bitransformer needs one extra layer than the N-k-cotransformer used in the p-cochain algorithm. In the next section, we will investigate the question of modifying the p-bichains, such that we can reduce the number of layers of k-comparators needed in the sorting algorithm. For example, we can remove the third layer of the k-comparators.

Furthermore, a basic observation from Definition 4.2 of the m-g-chains decomposition is that we need to consider partitioning each successive block of size q^2 into groups of q elements, such that the pattern is repeated in every consecutive blocks of size q^2. This is to ensure that the $(q, s-2)$-chains of the m-g-chains are the $(q, s-2)$-chains of an N-vector V (where $N = q^s$).

Using the p-cochains decomposition, the p-cochain merge algorithm can be implemented using k-comparators of various sizes, such that $k \geq q(q-1)$. However, this is not the norm for the algorithms derived from the m-g-chain decomposition scheme. For some of the other m-g-chain decompositions, it is more desirable to choose $k = q^2$. Although, the size of q^2-comparators is bigger than the size of $q(q-1)$-comparators, q^2-comparators seem to accommodate far more algorithms deriving from the m-g-chain decomposition scheme. Hence in our generalized merge scheme, we will relax the restriction on the size of the comparators to be q^2. A square array with q^2 elements can be sorted in $O(q)$, using the

$3q$ algorithm of Schnorr and Shamir [89]. We can regard the generalized merge scheme as an extension of those algorithms that sorts square arrays.

4.4 The Alt-q–Bichain Merge Algorithm

After the analysis of the p–bichain merge algorithm in the previous section, we found that the last three steps of the algorithm needs a sub-network with three layers of q^2–comparators to sort a $p \times q$ grid. In this section, we will investigate an alternative algorithm, which can be implemented by a q^2–comparator network that uses two layers of q^2–comparators, by constructing an alternative m-g-chain decomposition (derived from modifying the p–bichains). Hence this alternative algorithm improves on the depth of the p–bichain merge algorithm. Here is the new decomposition chain, the alt-q–bichains:

Definition 4.4 The Alt-q–Bichains: *There are q alt-q-bichains in V (q^{s-1}-subvectors of V), where the i^{th} alt-q-bichain contains elements from the i^{th} p-bichain interleaved with elements from the $(q-i-1)^{th}$ p-bichain in the following manner. The i^{th} alt-q-bichain starts with the first element of the i^{th} p-bichain, immediately followed by the second element of the $(q-i-1)^{th}$ p-bichain, followed by the third elements of the i^{th} alt-q-bichain, where this process is continued (alternating between the i^{th} p-bichain and the $(q-i-1)^{th}$ p-bichain). So the i^{th} alt-q-bichain contains all the elements with even indexes in the i^{th} p-bichain and all the elements with an odd indexes in the $(q-i-1)^{th}$ p-bichain.*

Furthermore, if q is an odd integer, then continue with the above definition: the $(\lfloor \frac{q}{2} \rfloor)^{th}$ alt-q-bichain is equal to the $(\lfloor \frac{q}{2} \rfloor)^{th}$ p-bichain.

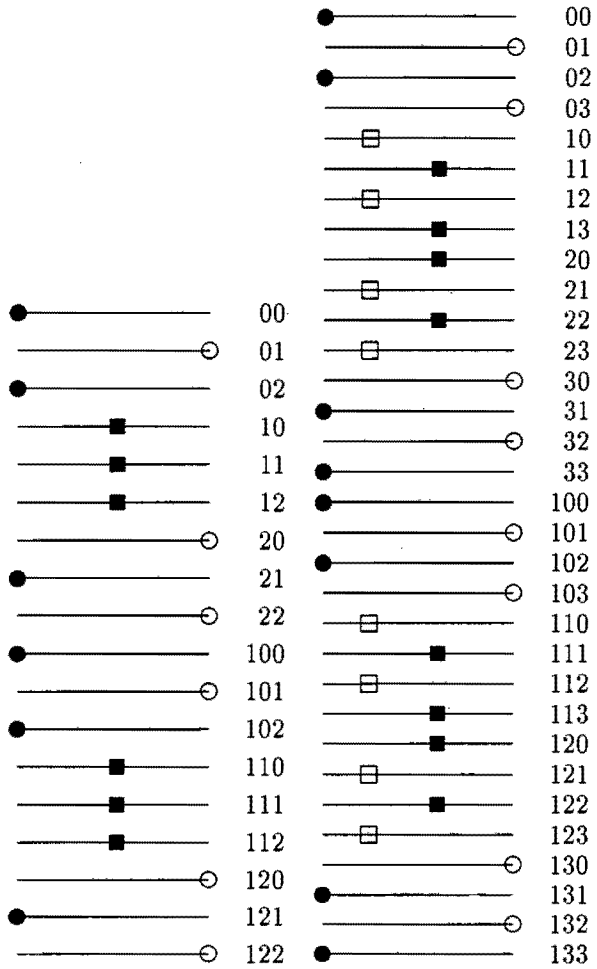
Here is some of the terminology that will assist in proving the correctness theorem and give a better understanding of the algorithm.

Definition 4.5 *Here is a list of chains that corresponds to the respective layer of any q^s-q^2-cotransformer:*

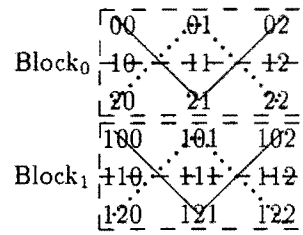
1. **Blocks** *of V: There are q^{s-2} such q^2-subvectors, where each subvector contains elements of V whose index in the q-ary expansion has the same q-ary digits, except the last two right most digits.*

2. **Shift $\lfloor \frac{q^2}{2} \rfloor$ Blocks** *or* **Half Shift blocks** *(h-s-blocks for short): There are $q^{s-2}-1$ such q^2-subvectors, where the i^{th} h-s-block contains elements of the last $\lfloor \frac{q^2}{2} \rfloor$ elements of the i^{th} block of V and the $\lceil \frac{q^2}{2} \rceil$ elements of the $(i+1)^{th}$ block of V.*

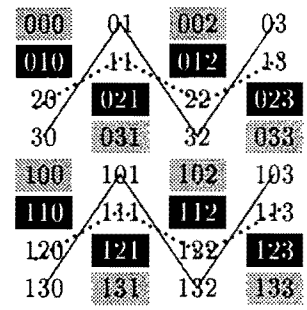
Note: There is a natural way of indexing the blocks, from 0 to $q^{s-2}-1$, using the q–ary expansion up until and excluding the right most two q–ary digits. In actual fact, the blocks of V and the s-h-blocks



(a) 3 Alt-3–Bichains. (b) 4 Alt-4–Bichains.

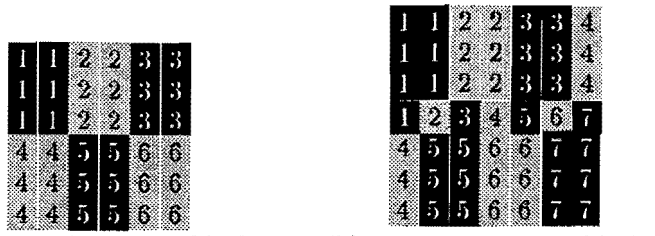


(d) 3 Alt-3–Bichains.



(d) 4 Alt-4–Bichains.

Figure 4.10: Some illustrations of the alt-q–bichain, where (a), (b) use the line representations and (c), (d) use the grid representations. (a) 3 alt-3–bichains of an 18–vector, (b) 4 alt-4–bichains of a 32–vector, (c) 3 alt-3–bichains on a 6×3 grid. (d) 4 alt-4–bichains on an 8×4 grid.



(a) 6 slices in a 6 × 6 block. (b) 7 slices in a 7 × 7 block.

Figure 4.11: Some illustrations of slices in (a) A 6 × 6 grid. (b) A 7 × 7 grid.

of V are just "square" sub-arrays of V. Hence it makes sense to talk about the columns of the block (*B-columns*) or the rows of the block (*B-rows*).

There are q *slices* in a block, they are just q equal partitioning of a block: If q is even, then elements of a slice are from the upper half of a pair of consecutive B-columns or the lower half of a pair of consecutive B-columns. The pair of consecutive B-columns is a pair of an even B-column followed by an odd B-column. However, when q is odd, we can not distribute the elements uniformly as above. We use the same definition as above for the slices in the upper half of the block, all the way to the $(q-2)^{th}$ B-column. Now the next slice consists of elements from the upper half of the $(q-1)^{th}$ B-column, and the lower half of the 0^{th} B-column. Next, we continue to allocate elements of the lower half of a pair of B-columns (an odd B-column and followed by an even B-column) to each new slice. Again we are able to apply the row major indexing scheme to index the slices, using the index of the block and the rank of the slice in the block (0 to $q-1$).

Here is the alt-q–bichain merge algorithm described in terms of operations on a $p \times q$ grid, and if the columns (the p–chains) of the grid are sorted, then the resulting grid will be sorted in row major ordering.

The Alt-q–Bichain Merge Algorithm

1. Sort the alt-q–bichains, groups of p elements alternating from pair of rows, Row_i and Row_{q-i-1} from each block of the array.

2. Sort the rows in groups of q, starting from Row_0.

3. Sort the rows in groups of q, starting from $\text{Row}_{\lfloor \frac{q}{2} \rfloor}$.

Again, we can make a similar observation as the one we made for the q–way r–merge grid algorithm, the last two steps of the alt-q–bichain merge algorithm can be implemented by two layers of q^2–comparators of the N–q^2–cotransformer. Furthermore, it is not difficult to see from Figure 4.10 that illustrate how an alt-q–bichain can be represent in the line representation (in terms of partitioning the N–vector V) and

the grid representation. Hence in Figure 4.12 we gave an example of the alt-q–chain merge algorithm on a 32–vector, in terms of the line representation.

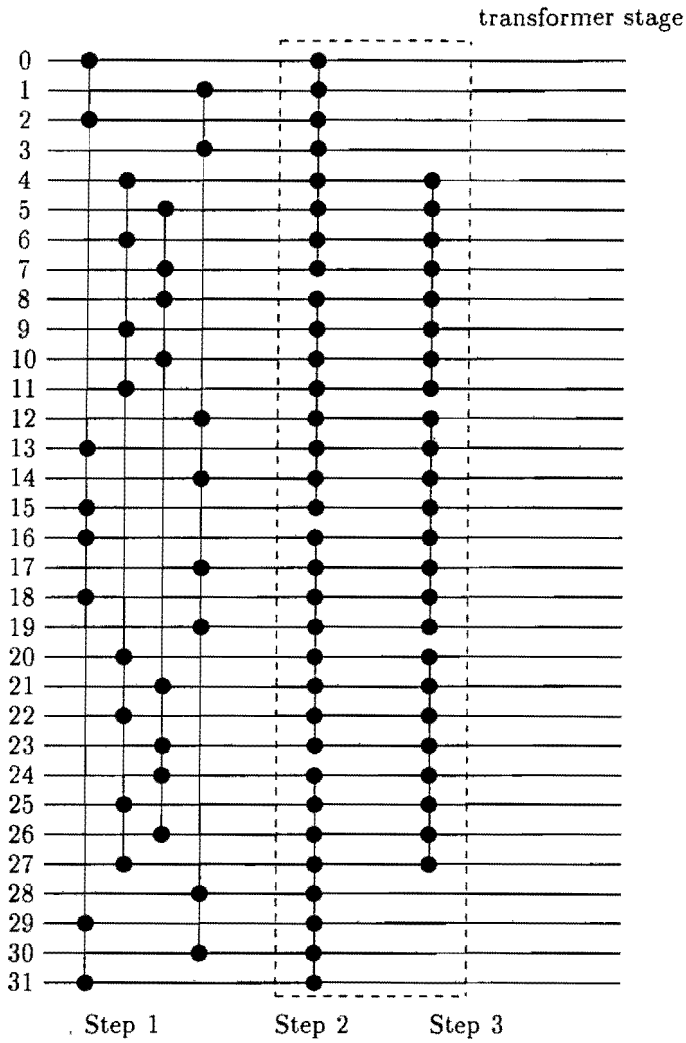


Figure 4.12: A line representation of the alt-q–bichain merge, on a 32–vector.

Next, we will state some obvious observations, without providing any proof:

(1). The alt-q–bichains decomposition belongs to the family of m-g-chains decomposition.

(2). The pattern of how the alt-q–bichains are interleaved in each block of V is identical for every block.

(3). Each B-column and each slice of the block intersects every alt-q–bichain exactly once.

(4). Suppose V is an array which consists only of 0's and 1's, and its p–chains are sorted, then the effect of sorting the alt-q–bichains of V is to exchange the 1's from B-columns and the 0's of dirty slices or all–0 slices of V. We do it in such manner, that we fill up the dirty slices or the all–0 slice with 1's, from the bottom upwards in V.

Lemma 4.4.1 *If the array V consists only of 0 and 1 elements, and one sorts successively the p–chains and the alt-q–bichains, then the resulting array has at most q number of dirty rows. Furthermore, those q dirty rows are contained either in one of the blocks or in one of the h-s-blocks.*

Proof: We will be focusing on how the 1's of the 0–1 array are moving down the array. A similar argument can apply to the case where we consider the 0's of the 0–1 array (we move the 0's up the array instead).

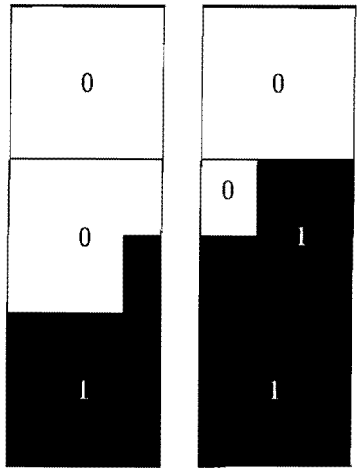
After the p–chains are sorted, all the 1's sink to the bottom of each column. Now we shall regard each column as a union of B-columns. Clearly, each column will have at most one dirty B-column. Therefore there are at most q dirty B-columns.

From observation (4), we know that an all–1 B-column is transferred to replace the 0's in a slice at the bottom of the array. To illustrate the process of sorting the alt-q–bichains, we introduce a second array which contains only of 0's as elements. Now we take a clean B-column from the first array and replace the 0's of the slice with the highest index (among the slices with 0's only) of the second array. We continue with this process until every all-1 B-columns in the first array is transferred into the second array.

Since there are only q columns in V, and each column has a finite number of B-columns, the above process will terminate after a finite number of steps. Now we are left with at most q dirty B-columns in the first array (the original array). The second array should have a jag or a flat joint between the area of 1's and the area of 0's (see Figure 4.13 for the illustration). In all the cases, the array has a row with minimum index, which we will call the *base*, where all the rows with a higher index are all–1 and the rows with a lower index are either dirty rows or all–0.

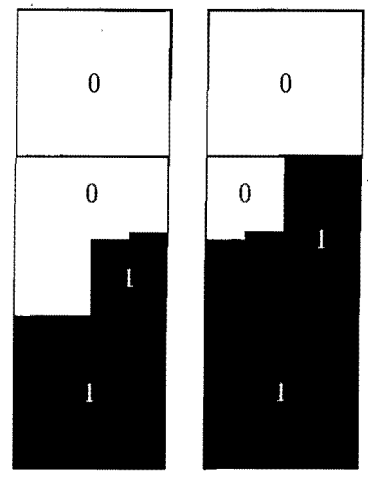
Claim: If any 1's from those dirty B-columns (at most q) in the first array are transferred to a row which is at least $q+1$ rows higher than the base, then at least $\lfloor \frac{q}{2} \rfloor$ rows immediate above the base are clean.

> **Proof:** We will ignore the cases in which it is impossible to get q rows higher (those arrays with a flat base, which means the next q rows will contain q slices). Now, for the array to get 1's in the row which is $\geq q+1$ higher than the base, the alt-q–bichain that contains this particular 1 must intersect at least $\lceil \frac{q}{2} \rceil$ of the B-column. Hence it must fill $t \geq \lceil \frac{q}{2} \rceil$ slices with 1's in the corresponding positions. Without loss of generality, we may assume it is the i^{th} alt-q–bichain and $i \leq \lceil \frac{q}{2} \rceil$ (a similar argument can apply, if $q \geq i > \lceil \frac{q}{2} \rceil$).



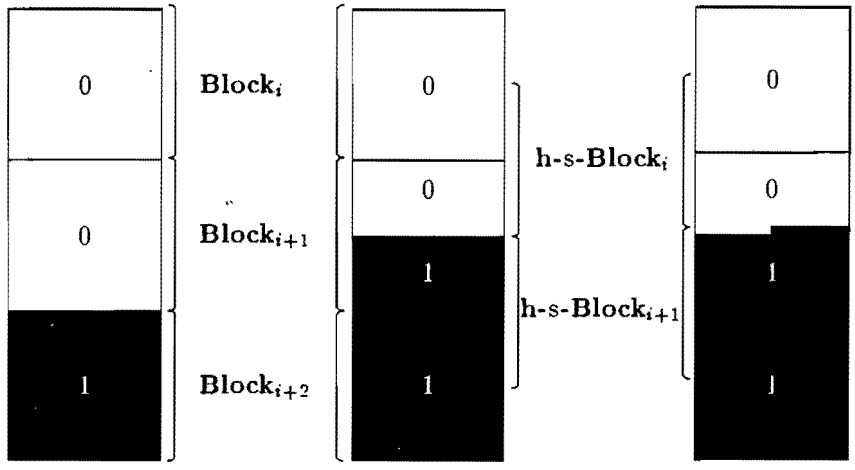
(i) (ii)

(B) 2 Possible Cases for q is even.



(i) (ii)

(C) 2 Possible Cases for q is odd.



(i) (ii) (iii)

(A) Flat Based Cases.

Figure 4.13: Here are all the possible cases of the second array derived from observation (4): (A) These are the possible cases for flat bases. However, when q is odd, we have (*iii*), since it full up the h-s-blocks. (B) Here are two possible cases for the partially full cases for q is even. (C) Here are two possible cases for the partially full cases for q is odd.

By definition the i^{th} alt-q–bichain intersects at position i of at most $\lceil \frac{q}{2} \rceil$ of the dirty B-columns: suppose there are l such dirty B-columns. Similarly, the i^{th} alt-q–bichain intersects at position $q-i-1$ of at most $\lfloor \frac{q}{2} \rfloor$ of the dirty B-columns; suppose there are l' such dirty B-columns. Now to flatten the base, we need to fill $k < \lfloor \frac{q}{2} \rfloor$ slices, but $k \leq \min\{l, l'\}$. This implies there are more than k pairs of dirty B-columns intersecting the i^{th} alt-q–bichain at position i and more than k pairs of dirty B-columns intersecting the i^{th} alt-q–bichain at position $q-i-1$.

Now using the result from the above claim, we have proved the lemma. Furthermore, these q successive dirty rows are contained either in one of the blocks or one of the h-s-blocks, because they are contained in two levels of slices. ∎

By the fact that we have observed the last two steps of the alt-q–bichain merge algorithm can be implemented by the two layers of q^2–comparators of the q^s–q^2–cotransformer, we can state and prove the next theorem in terms of procedures that use k–comparators.

Theorem 4.4.2 *After the p–chains and the alt-q–bichains are sorted in V successively, then the resulting array can be sorted by the N–q^2–cotransformer.*

Proof: It is obvious from Lemma 4.4.1, after the p–chains and the alt-q–bichains of V are sorted successively, we just need to sort the dirty block or the dirty h-s-block in V. However, the first layer of the N–q^2–cotransformer sorts every blocks of V and the second layer of the N–q^2–cotransformer sorts every h-s-block of V. Hence the resulting array is sorted. ∎

Following from Theorem 4.4.2, we have observed that the alt-q–bichain algorithm will work with specific size of k–comparators (in particular, $k = q^2$), unlike the previous two algorithms (the cochain algorithm and the bichain algorithm), where the size of the k–comparators used just needs to be bigger than $q(q-1)$.

4.5 New Type of N–q^2–Transformer

In every instance of the grid algorithm that we have constructed so far, each algorithm will first sort the columns of the array, and follow by sorting some subparts of the array. Then we complete the algorithm by sorting the blocks and the h-s-blocks of the array in succession (see Definition 4.5 for the definition of the blocks and the h-s-blocks of an array). In the case of the p–bichain merge algorithm, an extra step was needed, where we sort the blocks of the array again.

For each algorithm we made the connection between the operations of sorting the blocks of the array to a corresponding k–comparator sub-network (a transformer). Hence to simplify the presentation of the

results found in this section, we shall refer to the corresponding transformer instead, when we need to refer to the operations of sorting the blocks and the h-s-blocks of the array.

In the previous section, we have shown that by sorting the modified p–bichains, the new algorithm needs to use the N–k–cotransformer only, instead of the N–k–bitransformer. However, there are other approaches which can reduce the depth of the q^2–comparator networks of the p–bichain merge algorithm. Another method of reducing the number of q^2–comparators needed in the p–bichain merge algorithm is to modify the N–q^2–transformer.

We know the following fact from the previous sections: It is proved in Theorem 4.3.2, that if a $p \times q$ grid A has sorted p–chains, and the p–bichains of A are then sorted, then the resulting array will have both sorted p–chains and sorted p–bichains.

We assume that V (the $(p.q)$–vector derived from A) satisfies the above condition and contains only 0's and 1's as elements (the $[0-1]$ **principle**). Now, given the i^{th} row and the j^{th} row of the resulting array, where $i < j$, then the number of 1's in the i^{th} row is less than the number of 1's in the j^{th} row. From Lemma 4.3.3, we know there is an interval $|I| \leq q(q-1)$, such that $v_l = 0$ for all $l < \min(I)$ and $v_l = 1$ for all $l > \max(I)$. We will continue with the notation of the proof of Lemma 4.3.3, we let $t = \max\{i : (\exists j) a_{ij} = 0\}$ and $s = \min\{i : (\exists j) a_{ij} = 1\}$, where the rows are denoted as Row_t and Row_s respectively. Furthermore, the interval I intersects at most q successive rows ($t - s + 1 \leq q$). Henceforth in this section, I will be used to denote the interval found in Lemma 4.3.3.

Definition 4.6 *The* **Skew Blocks** *or the* **Diagonal Blocks** *are q^{s-2} q^2–subvectors of V. There are three different types of skew blocks; we use the elements of the rows of V to describe each skew block:*

(1.) *The first skew block contains the first $(\lfloor \frac{q}{2} \rfloor + 1)q$ elements of V, plus either*

- *i* *If q is odd, then the first $q-i$ elements from each $(\lfloor \frac{q}{2} \rfloor + i)^{th}$ row, where $1 \leq i \leq q-1$.*
- *ii* *If q is even, then the first $q-i$ elements from each $(\lfloor \frac{q}{2} \rfloor + i)^{th}$ row, where $1 \leq i \leq (\frac{q}{2} - 1)$; and the first $q-i-1$ elements from each $(\lfloor \frac{q}{2} \rfloor + i)^{th}$ row, where $\frac{q}{2} \leq i \leq q-2$.*

(2.) *The intermediate blocks: There are $q^{s-2} - 2$ such q^2–subvectors in V, where they are indexed from 1 to $q^{s-2}-2$. The i^{th} skew block starts at the s_i^{th} row, where $s_i = (\lfloor \frac{q}{2} \rfloor + 2) + (i-1)q$. Now*

- *i* *If q is odd then the elements of the i^{th} block are listed as follow:*
  - *The last $k - s_i + 1$ elements from each Row_k, where $k = s_i$ to $s_i + q - 1$.*
  - *The first $s_i + 2q - k - 1$ elements from each Row_k, where $k = s_i + q$ to $s_i + 2q - 2$.*
- *ii* *If q is even then the elements of the i^{th} block are listed as follow:*
  - *The last $k - s_i + 1$ elements from each Row_k, where $k = s_i$ to $s_i + \frac{q}{2} - 2$.*
  - *The last $k - s_i + 2$ elements from each Row_k, where $k = s_i + \frac{q}{2} - 1$ to $s_i + q - 2$.*

- *The first $s_i + 2q - k - 1$ elements from each Row_k, where $k = s_i + q - 1$ to $s_i + q + 3$.*
- *The first $s_i + 2q - k - 2$ elements from each Row_k, where $k = s_i + q + 4$ to $s_i + 3\frac{q}{2} + 2$.*

(3.) *The last skew block contains the last $(\lfloor \frac{q}{2} \rfloor + 1)q$ elements of V, plus either*

i *If q is odd, then the last $q - i$ elements from each $(q^{s-1} - (\lfloor \frac{q}{2} \rfloor + i))^{th}$ row, where $1 \leq i \leq q - 1$.*

ii *If q is even, then the last $q - i$ elements from each $(q^{s-1} - (\lfloor \frac{q}{2} \rfloor + i))^{th}$ row, where $1 \leq i \leq (\frac{q}{2} - 1)$; and the last $q - i - 1$ elements from each $(q^{s-1} - (\lfloor \frac{q}{2} \rfloor + i))^{th}$ row, where $\frac{q}{2} \leq i \leq q - 2$.*

Note: The last skew block is equal to the first skew block turned upside down.

From Definition 4.6, it is clear that q consecutive rows in V can intersect at most three skew blocks. We will continue using the notion of "dirty" and "clean" on the skew blocks. We can make the following trivial observations :

1. For any three consecutive skew blocks, label the blocks successively from top to bottom as skew_1, skew_2 and skew_3. Any element of skew_1 will need to cross at least q rows to reach an element of skew_3 that is in the same column of V and vice versa. Henceforth, we will use skew_1, skew_2 and skew_3 to label any three consecutive skew blocks from top to bottom.

2. Except for the first skew block of V, all the skew blocks are inclined from the left of the array to the right of the array. This means that the column in the right of a skew block contains elements from rows of V with lower indices. Compared to the elements of any column to the left of a block, they are from rows with higher indices.

See Figure 4.14 for some illustrations of the skew blocks.

Lemma 4.5.1 *If A contains 0's and 1's as elements, and the p-chains and the p-bichains of A are sorted successively, then the resulting grid will have at most two consecutive skew blocks that are dirty.*

Proof: Let I denote the interval that is dirty in A, after the p-chains and the p-bichains of V are sorted successively. So this is trivially true if I intersects at most two successive skew blocks.

Suppose I intersects three successive skew blocks, they are labelled as skew_1, skew_2 and skew_3 respectively. Assume that all three skew blocks are dirty. Let a_{ij} be the highest index element of A in skew_3 that contains 0, and let $a_{i'j'}$ be the lowest index element of A in skew_1 that contains 1 (a_{ij} and $a_{i'j'}$ exist, since we assume that the blocks are dirty).

Now there are at most q dirty rows, so a_{ij} and $a_{i'j'}$ can be at most q rows apart. However, following

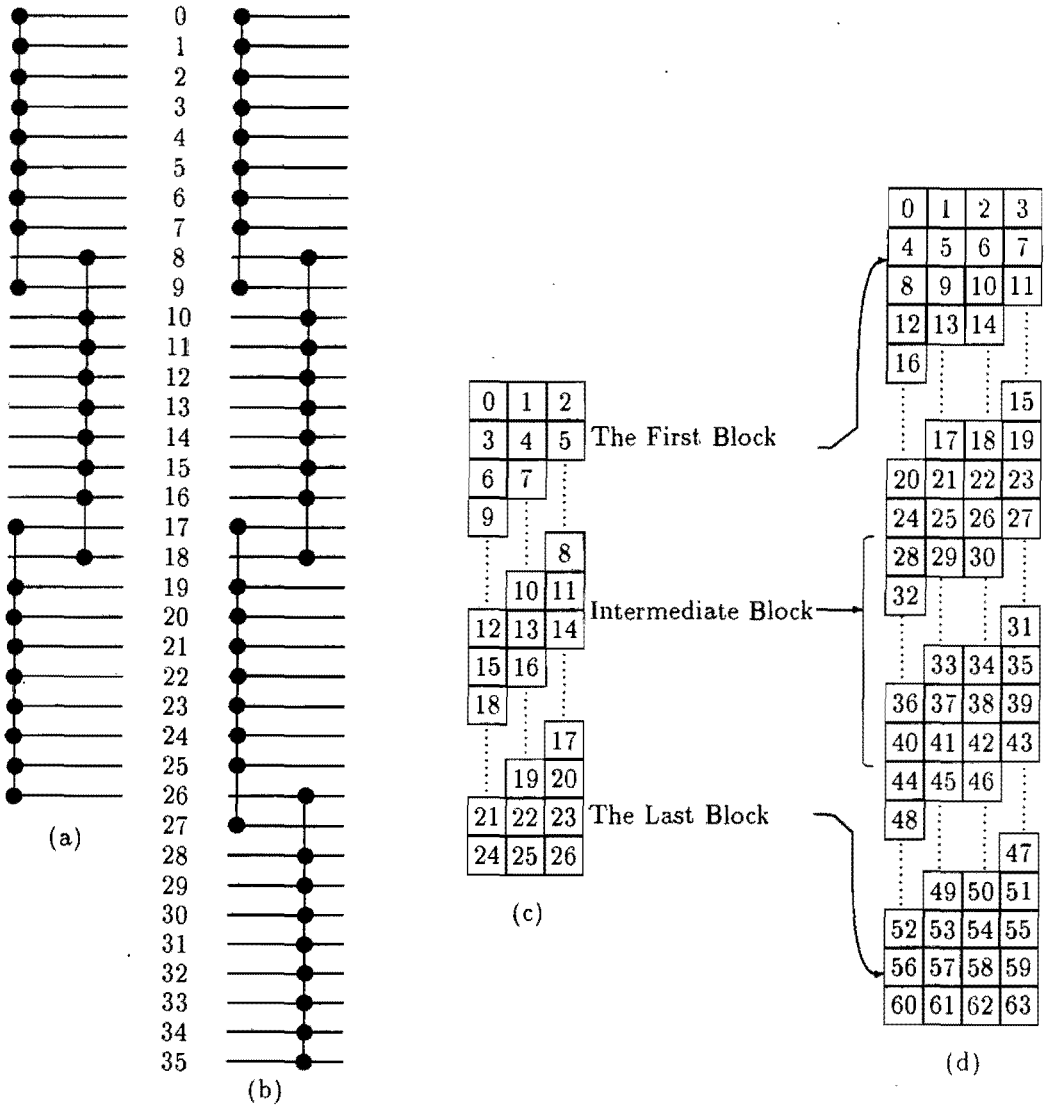


Figure 4.14: Here are the skew blocks for an 9×3 grid and a 16×4 grid.

from the above two observations, we can deduce that Col_i must be to the left of Col_j $(j < j)$. Since the bichains are sorted, Row_i must contains 0 at a_{ij}, this implies that the p–chains are not sorted. This contradicts Theorem 4.3.2, hence our assumption is false. So there are at most two skew blocks are dirty. ∎

Next, we define the N–q^2–skewtransformer. The aim here is to improve on the p–bichain merge algorithm that use the N–q^2–bitransformer, by replacing it with the N–q^2–skewtransformer. The consequence of replacing the N–q^2–bitransformer by the N–q^2–skewtransformer enables the algorithm to use two levels of q^2–comparators to sort any array, where its p–chains and p–bichains are sorted successively.

Definition 4.7 *Here is a k–comparator network, defined in terms of the Knuth's [49] line representation for a network on V, the N–k–***skewtransformer***: This network consisting of two layers: the first layer has $\frac{N}{k}$ k–comparators applied to the skew blocks of V. The second layer has $\frac{N}{k} - 1$ k–comparators, each applied to successive lines, starting from the $(\lceil \frac{k}{2} \rceil)^{th}$ line. Note: In the second layer, the first $\lfloor \frac{k}{2} \rfloor$ lines and the last $\lceil \frac{k}{2} \rceil$ lines are not operated on.*

Here we describe the modified p–bichain merge algorithm in terms of operations on a $p \times q$ grid, and if the columns (the p–chains) of the grid are sorted, then the resulting grid is sorted in row major ordering:

The Modified p–Bichain Merge Algorithm

1. Sort the p–bichains, the union of all the elements from the "rows of length q" in steps of q, in the array.

2. Sort the skew blocks of the array (see Definition 4.6).

3. Sort the rows in groups of q, starting from $\text{Row}_{\lfloor \frac{q}{2} \rfloor}$.

Clearly, the last two steps of the modified p–bichain merge algorithm of size N can be implemented by an N–q^2–skewtransformer.

Before we prove the correctness of the modified p–bichain merge algorithm, we will analyze the structure of the *diagonal layer* (we will call the first layer of the N–q^2–skewtransformer the diagonal layer). Now we will take a closer look at the intermediate blocks (from Definition 4.6). We regard each intermediate block as a union of two parts: An up-right upper triangle union with a flipped-over lower triangle. We will illustrate this using Figure 4.15. For short, we will denote those two parts as the *upper triangle* and the *lower triangle* of the skew block, respectively. Here we introduce the terminology formally:

Definition 4.8 *Given a skew intermediate block, we can partition the skew block into two parts, which contain* $\lfloor \frac{q^2}{2} \rfloor$ *and* $\lceil \frac{q^2}{2} \rceil$ *elements respectively.*

1. *If q is odd, then:*

   i *The* **upper triangle** *of a skew block contains elements from the first* $q-1$ *rows of the block and the first* $\lfloor \frac{q}{2} \rfloor$ *elements of the* q^{th} *row of the block.*

   ii *The* **lower triangle** *of a skew block contains elements from the last* $q-1$ *rows of the block and the last* $\lceil \frac{q}{2} \rceil$ *elements of the* q^{th} *row of the block.*

2. *If q is even, then:*

   i *The* **upper triangle** *of a skew block contains elements from the first* $q-1$ *rows of the block.*

   ii *The* **lower triangle** *of a skew block contains elements from the last* $q-1$ *rows of the block.*

Furthermore, we index the rows of the upper triangle successively from top to bottom, starting from 1 to the last row, which is either q or $q-1$ *depending on if q is odd or even respectively. We index the rows of the lower triangle successively from bottom to top, again starting from 1 to the last row.*

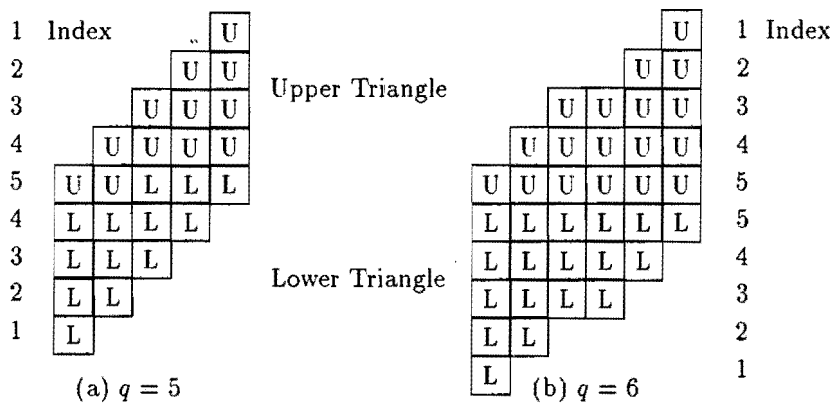


Figure 4.15: Examples of the upper triangle (elements of the upper triangle are labelled by letter U) and the lower triangle (elements of the lower triangle are labelled by letter L).

To prove the correctness proof, we analyze the number of 1's in the skew blocks. As each skew block is sorted, the 0's float to the top and the 1's sink to the bottom of the block. The skew block has an inverse image between the top half of the skew block to the bottom half of the skew block, hence we only need to consider how the 1's are sorted in the block. This means the same analytic argument can be

applied if we want to consider the 0's instead.

Theorem 4.5.2 *After the p-chains and the p-bichains of V are sorted successively, if there are two dirty skew blocks (labelled $skew_1$ and $skew_2$) in the resulting array, then the following conditions are true:*

1. $skew_1(1) < \lfloor \frac{q^2}{2} \rfloor$
2. $skew_2(1) > \lceil \frac{q^2}{2} \rceil$

Proof: From observation 1 and 2 on page 107, we know that to reach skew_1 from skew_2, we need to consider the following facts:

1. The array needs to have enough all-1 rows in skew_2 for the dirty rows to effect skew_1 (make skew_1 dirty). We will call this effect on the number of 1's in the blocks, the **height factor**.
2. The array needs to have enough dirty columns containing 1's and protrude left enough in the array to reach skew_1. We will call this effect on the number of 1's in the blocks, the **left factor**.

Intuitively, for the dirty rows of I (defined in Lemma 4.3.3) to cause skew_1 and skew_2 both to be dirty, the number of 1's needs to satisfy these two factors. So the height factor will create a base area that contains 1's. The left factor will create an area of 1's along the columns to the right of the array (trapezium). Using these two factors, we are then able to find the boundary condition for the number of 1's in the blocks.

If I ended in the lower triangle of the skew_1, then it is trivially true that $\text{skew}_1(1) < \lfloor \frac{q}{2} \rfloor$. However, if I ended in the upper triangle of the skew_1, without loss of generality, let Row_t of I coincide Row_k of the block. We have two cases to consider:

(a) If both skew_1 and skew_2 are the intermediate blocks, and $3 \leq k \leq q-1$.
Now if q is even, then

$$\begin{aligned} \text{skew}_1(1) &\leq \frac{(q+1-k)(q-k)}{2} + \frac{q}{2} + \frac{((q-k)+(q-2))(k-1)}{2} \\ &\leq \frac{q^2}{2} - k + 1 \end{aligned} \tag{4.1}$$

Else q is odd

$$\begin{aligned} \text{skew}_1(1) &\leq \frac{(q+1-k)(q-k)}{2} + \frac{((q-2)+(q-k+1))(k-2)}{2} \\ &\leq \frac{(q^2-3q+2)}{2} \end{aligned} \tag{4.2}$$

(b) If skew_1 is the first skew block of V, and Row_k is one of the first $\lfloor \frac{q}{2} \rfloor + 1$ rows of V $(0 \leq k \leq \lfloor \frac{q}{2} \rfloor)$.

Now if q is even, then

$$\begin{aligned} \text{skew}_1(1) &\leq \frac{1}{2}((\frac{q}{2}-k)+(q-2))(\frac{q}{2}+k-1)+\frac{1}{2}(\frac{q}{2}-k-1)(\frac{q}{2}-k) \\ &\leq \frac{(q^2-3q+2)}{2} \end{aligned} \tag{4.3}$$

Else q is even

$$\begin{aligned} \text{skew}_1(1) &\leq \frac{1}{2}(\lfloor\frac{q}{2}\rfloor-k)(\lfloor\frac{q}{2}\rfloor-k+1)+ \\ &\quad \frac{1}{2}(q-2+\lfloor\frac{q}{2}\rfloor-k+1)(\lfloor\frac{q}{2}\rfloor+k-1) \\ &\leq \frac{q^2-3q}{2} \end{aligned} \tag{4.4}$$

Hence the greatest upper bound of $\text{skew}_1(1)$ is always less than $\lfloor\frac{q^2}{2}\rfloor$. So it is true that $\text{skew}_1(1) < \lfloor\frac{q^2}{2}\rfloor$.

If I starts from the upper triangle of skew_2, then it is trivially true that $\text{skew}_2(1) > \lceil\frac{q^2}{2}\rceil$. However, if I starts from the lower triangle of skew_2, without loss of generality, let Row_t of I coincide Row_k of the block. We have two cases to consider:

(c) If both skew_1 and skew_2 are the intermediate blocks, and $3 \leq k \leq q-1$.
Now if q is even, then

$$\begin{aligned} \text{skew}_2(1) &\geq \frac{k(k+1)}{2}+\frac{q}{2}+\frac{(q+k-1)(q-k)}{2} \\ &\geq \frac{q^2}{2}+k \end{aligned} \tag{4.5}$$

Else q is odd

$$\begin{aligned} \text{skew}_2(1) &\geq \frac{k(k-1)}{2}+(q-1)+\frac{(q-k+1)(q+k)}{2} \\ &\geq \frac{q^2+3q-2}{2} \end{aligned} \tag{4.6}$$

(d) If skew_2 is the last skew block of V, and Row_k is one of the last $\lfloor\frac{q}{2}\rfloor+1$ rows of V ($q^{s-1}-\lfloor\frac{q}{2}\rfloor-1 \leq k \leq q^{s-1}-1$).
Now if q is even, then

$$\begin{aligned} \text{skew}_2(1) &\geq (\frac{q}{2}+1)q \\ &\geq \frac{q^2}{2}+q \end{aligned} \tag{4.7}$$

Else q is odd

$$\begin{aligned} \text{skew}_2(1) &\geq (\lfloor\frac{q}{2}\rfloor+2)q-1 \\ &\geq \frac{q^2+3q}{2}-1 \end{aligned} \tag{4.8}$$

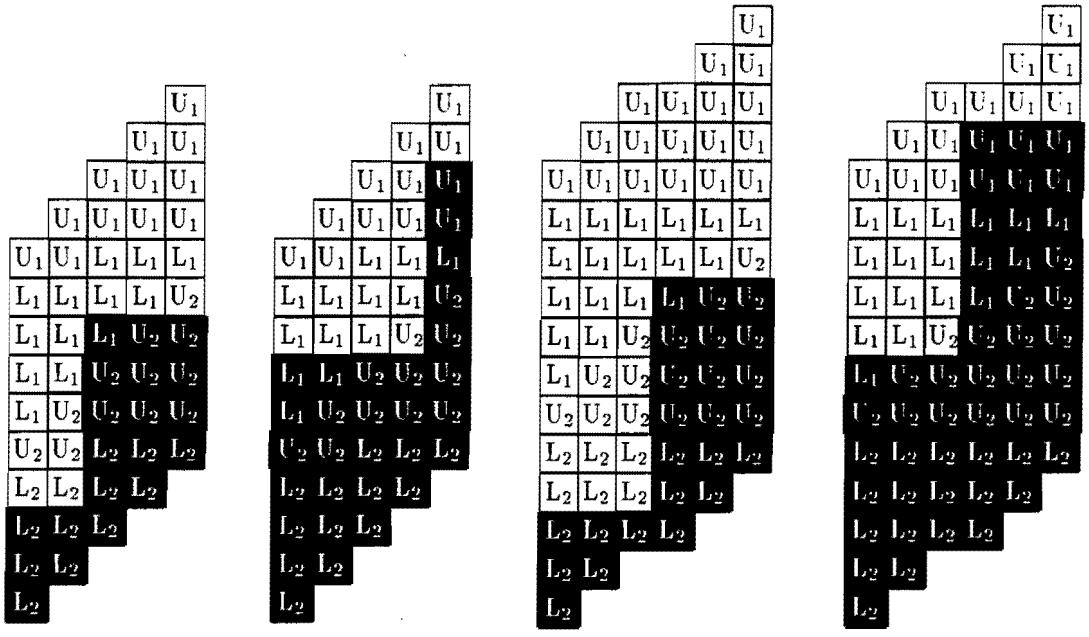


Figure 4.16: Some illustrations to demonstrate the left factor and the height factor.

Hence the least lower bound of $\text{skew}_2(1)$ is always greater than $\lceil \frac{q^2}{2} \rceil$. So it is true that $\text{skew}_2(1) > \lceil \frac{q^2}{2} \rceil$. ∎

In the next theorem, we will prove in terms of k–comparators used in the transformer stage, instead of operations described in the grid algorithm.

Theorem 4.5.3 *The N–q^2–skewtransformer will sort any N–vector, which resulted from sorting the p–chains followed by sorting the p–bichains.*

Proof: Suppose V has one dirty skew block, this implies the array contains only 0's above this skew block and only 1's below this skew block. If the number of 1's in this dirty skew block is $\leq \lceil \frac{q^2}{2} \rceil$, then after the diagonal layer of the N–q^2–skewtransformer, all the 1's will be in the lower triangle of this block. There is a q^2–comparator in the 2^{nd} layer of the N–q^2–skewtransformer that overlaps the lower triangle, hence V is sorted. Else the number of 1's in this dirty skew block is $> \lceil \frac{q^2}{2} \rceil$. In this case, a similar argument can apply. After the diagonal layer of the N–q^2–skewtransformer, we need to sort the upper triangle of this block and there is a q^2–comparator in the 2^{nd} layer of the N–q^2–skewtransformer that overlaps the upper triangle. Again, V is sorted.

Suppose V has two dirty skew blocks, which we labelled skew_1 and skew_2, then, by Theorem 4.5.2, we know that $\text{skew}_1(1) < \lfloor \frac{q^2}{2} \rfloor$ and $\text{skew}_2(1) > \lceil \frac{q^2}{2} \rceil$. This implies that after the diagonal layer of the N–q^2–skewtransformer, only the lower triangle of the skew_1 block and the upper triangle of the skew_2

block are not sorted. However, they are overlapped by one of the q^2–comparator in the 2^{nd} layer of the N–q^2–skewtransformer, hence it sorts the array.

It is clear that the first skew block and the last skew block are either sorted after the diagonal layer, or the block will reach the same result as above. Hence the same argument as above can apply to them respectively. So the skewtransformer sorts the corresponding N–vector correctly. ∎

4.6 The Generalized Merge Algorithm

We will construct a family of merge algorithms which uses the m-g-chains to sort any N–vector. Furthermore, these family of merge algorithms can be implemented on k–comparator networks, where the k–comparators used have the same size $k = q^2$. Although the new algorithm formulated in the paper of Becker and Litman [10] is using k–comparators where $k \geq q(q-1)$, the size of the k–comparators used in the generalized merge algorithm must accommodate all the underlying grid algorithms. In particular, we choose to use the q–way q–merge algorithms as the underlying algorithms and we implement the algorithm on the q^2–comparator networks. The q^2–comparators will satisfy all the underlying grid algorithms that are incorporated in the generalized merge algorithm. For detailed explanations on the p–cochain algorithm readers are referred to the paper of Becker and Litman [10]. We will incorporate their result into our merge algorithm, which is a more generalized version. Next, we will prove the correctness of the merge algorithm, using the decomposition tree.

In our generalized merge scheme, we will relax the restriction and fixed q^2 as the size of our comparators. One further advantage of using the comparator of size q^2, is that such a comparator can be regarded as sorting on a square array. There are many efficient sorting algorithms, in the literature, that sort square arrays, hence we do have methods to sort the square arrays. We can regard the generalized merge scheme as an extension of those algorithms to the rectangular arrays.

We concentrate on a few types of sub-chains, which partition the $p \times q$ array into groups of p–subvectors ($p = q^{s-1}$), where the $(q, s-2)$–chains of those p–subvectors are the $(q, s-2)$–chains of the original array. In fact, there are many different ways in grouping the elements, however, we will consider only three types in this chapter, the p–cochains, the p–bichains and the alt-q–bichain. In the last section of this chapter, we will touch on one of the m-g-chain decompositions, which is very inefficient if we incorporate it into our scheme.

4.6.1 The Construction of the Merge Algorithm

Most of the procedures that are used in the construction of the generalized merge algorithm have already been introduced on page 49, hence we will not redefine the same operations twice. Next we state the

generalized merge algorithm formally, in terms of the procedures defined on page 49. Here is a description of the main procedure:

- **G-Merge(In:** v, q^s, q; **Out:** w): Construct a q–merge with input q^s–vector v and output q^s–vector w. If the input q^s–vector v has q sorted $(q, s-1)$–chains, then this procedure will output a sorted q^s–vector w.

The merge procedure goes as follow:

Procedure G-Merge(In: v, q^s, q; **Out:** w)

If $s = 2$ **then** apply a single q^2–comparator;
Else
Begin
Choose one of the following decomposition (1) cochain. (2) bichain. (3) alt-q–bichain.
Let **m-g-chain** be the type chosen.
Divide(v, q^s, q, **m-g-chains**; $\{c_i : i = 0, 1, \ldots, q-1\}$);
G-Merge(c_0, q^{s-1}, q; w_0), **G-Merge**(c_1, q^{s-1}, q; w_1), ..., **G-Merge**(c_{q-1}, q^{s-1}, q; w_{q-1}).
Unite($\{w_i : i = 0, 1, \ldots, q-1\}$, q^{s-1}, **m-g-chains**; w').
Transformer(w', q^s, q^2, q, **m-g-chain**; w)
End

Now we will prove the correctness for the Generalized Merge Procedure: (In fact, similar arguments used in this correctness proof of Theorem 4.6.1 can be used to prove the correctness of every merge algorithm introduced in the previous chapter.)

Theorem 4.6.1 *Suppose for each* $N = q^s$, $s \geq 2$, *there is an* N-q^2-*transformer for a vector of size* q^{s-1}, *which sorts any vector* V *of size* q^p *with sorted* q^{s-1}-*m-g-chains. The corresponding m-g-chains decomposition can be the cochain, the bichain or the alt-*q-*bichain. Then the generalized merge procedure sorts any input* N-*vector* V *with sorted* $(q, s-1)$-*chains.*

Proof: We will prove this theorem, by induction on the power s of q. The generalized merge procedure clearly sorts if $s = 2$, since we have $N = q^2$, therefore a single q^2–comparator will sort.

Now assume that the merge algorithm sorts vectors with sorted $(q, s-2)$–chains for $p = \frac{|V|}{q} = q^{s-1}$. Apply the merge algorithm to each m-g-chain (the cochain, the bichain or the alt-q–bichain) of a vector

V of size q^s with sorted p–chains. The $(q, s-2)$–chains of each m-g-chain of the chosen type are easily seen to be sub-chains of the chains of V, and are therefore sorted (see Lemma 4.1.1). By the hypothesis, therefore, the G-Merge algorithm correctly sorts each m-g-chain of the chosen type. Hence, after the application of all the merges, the vector has sorted m-g-chains with the respective property. This is true because we have already prove the correctness for the three different m-g-chain merge algorithms: Theorem 4.2.5 for the cochains, Theorem 4.3.4 for the bichain, lastly Theorem 4.4.2 for the alt-q–bichain. The application of the respective N–q^2–transformer will therefore now sort the vector. The result follows by induction. ∎

4.6.2 Time Complexity of the Generalized Merge Procedure

We assumed throughout this section that $N = q^s, s \geq 2$, and that $k = q^2$, $q \geq 2$. For $q > 2$ let $T_k(N)$ denote the depth of the merge of size N, let $C_k(N)$ denote the number of k–comparators of the merge. For $q = 2$, we define these to be the corresponding quantities associated with Batcher's Bitonic network. To simplify the calculation, we assume that the algorithm uses the N–q^2–cotransformer and N–q^2–skewtransformer only, not the bitransformer. Since if the bichain merge is applied and the bitransformer is used, the depth of the transformer is increased by 1, hence we used the skewtransformer only. The time complexity of the merge procedure is the same as the multi-way multi-merge algorithm calculated on page 58, where $r = q$ and $k = q^2$. So we have

$$\begin{aligned} T_k(N) &= 2\lceil \log_q N \rceil - 3 \\ &= 4(\log_k N) - 3 \end{aligned}$$

$$\begin{aligned} C_k(N) &= qC_{q^2}(\frac{N}{q}) + (2\frac{N}{q^2} - 1) \\ &= (2\log_q(N) - 3)\frac{N}{q^2} - \frac{N-q}{q(q-1)} \\ &= (4\log_k(N) - 3)\frac{N}{q^2} - \frac{N-q}{q(q-1)} \\ &= 4\log_k(N)\frac{N}{k} - O(\frac{N}{k}). \end{aligned}$$

4.7 The Multiple Merge Sorting Algorithm

Now we construct a sorting algorithm, using the family of the generalized merge algorithm. We will call this the Multiple Merge Sorting algorithm, which is stated formally as follows:

The Multiple Merge Sorting Algorithm

Input V (a q^s–vector).
Decompose V into q^{s-2} $(q,2)$–chains
For $i = 2$ to s **Do**

Apply the G-Merge procedure on each (q,i)–chain (We regard the (q,i)–chains as q^i–subvectors, where each q^i–subvector has sorted $(q,i-1)$–chains).

If $i \leq s-1$, then construct the $(q,i+1)$–chains from merging the resulting (q,i)–chains.

Output the resulting q^s–vector.

Next we will use the correctness of the merge algorithm to prove the correctness for the Multiple Merge Sorting algorithm:

Theorem 4.7.1 *The Multiple Merge Sorting algorithm sorts any q^s-vector.*

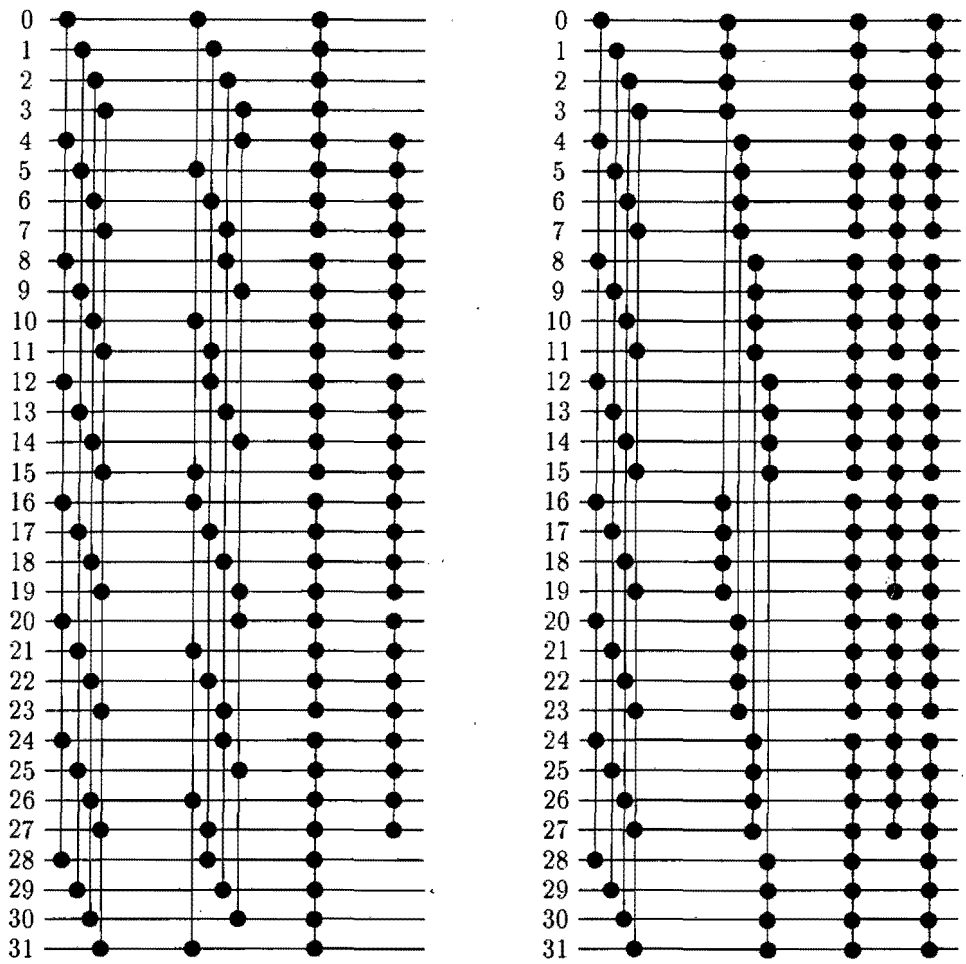
Proof: We will prove this inductively on the power of q of the input vectors. If the input vector is a q^2–vector, then we apply the merge network on a q^2–subvector, which uses just one q^2–comparator to sort, then exit. Hence the resulting vector is sorted.

Inductive hypothesis: Assume that the sorting algorithm sorts any q^l–vector, where $l \leq s-1$. Now we need to show it sorts any q^s–vector. At the $(s-1)^{th}$ iteration for q^s–vector, by the inductive hypothesis, we have sorted $(q,s-1)$–chains. Since the merge algorithm is correct, the merge algorithm will sort the q^s–vector. Hence the sorting algorithm sorts any q^s–vector. ∎

We illustrate some merge networks derived from the multiple merge sorting algorithm in the following figures (Figure 4.17 and Figure 4.18).

4.7.1 Time Complexity of the Multiple Merge Sorting Algorithm

The Multiple Merge Sorting algorithm uses $s-1$ stages of the merge algorithm to sort a q^s–vector. In the first stage, the sorting algorithm uses q^{s-2} merge networks for q^2–vectors in parallel. In the second stage the sorting algorithm uses q^{s-3} merge networks for q^3–vectors, in parallel. This continues, until the q^s–vector is sorted. Let $T_k^s(N)$ be the depth of the sorting algorithm for any N–vector. Let $C_k^s(N)$ be the number of k–comparators used in the sorting algorithm for any N–vector. Again, here we assume that only the N–q^2–cotransformer is used. Again the time complexity is the same as the sorting algorithm using the multi-way multi-mergers, where the underlying merge are q–way q–merge algorithms



(a) Cochain Sorting Network.

(b) Bichain Sorting Network.

Figure 4.17: Here are two examples of the line representation of networks constructed from the multiple merge sorting algorithm: (a) An 8-comparator network that used the cochain merge algorithm as the underlying merge algorithm. (b) An 8-comparator network that used the bichain merge algorithm as the underlying merge algorithm, where the 32-8-bitransformer is used.

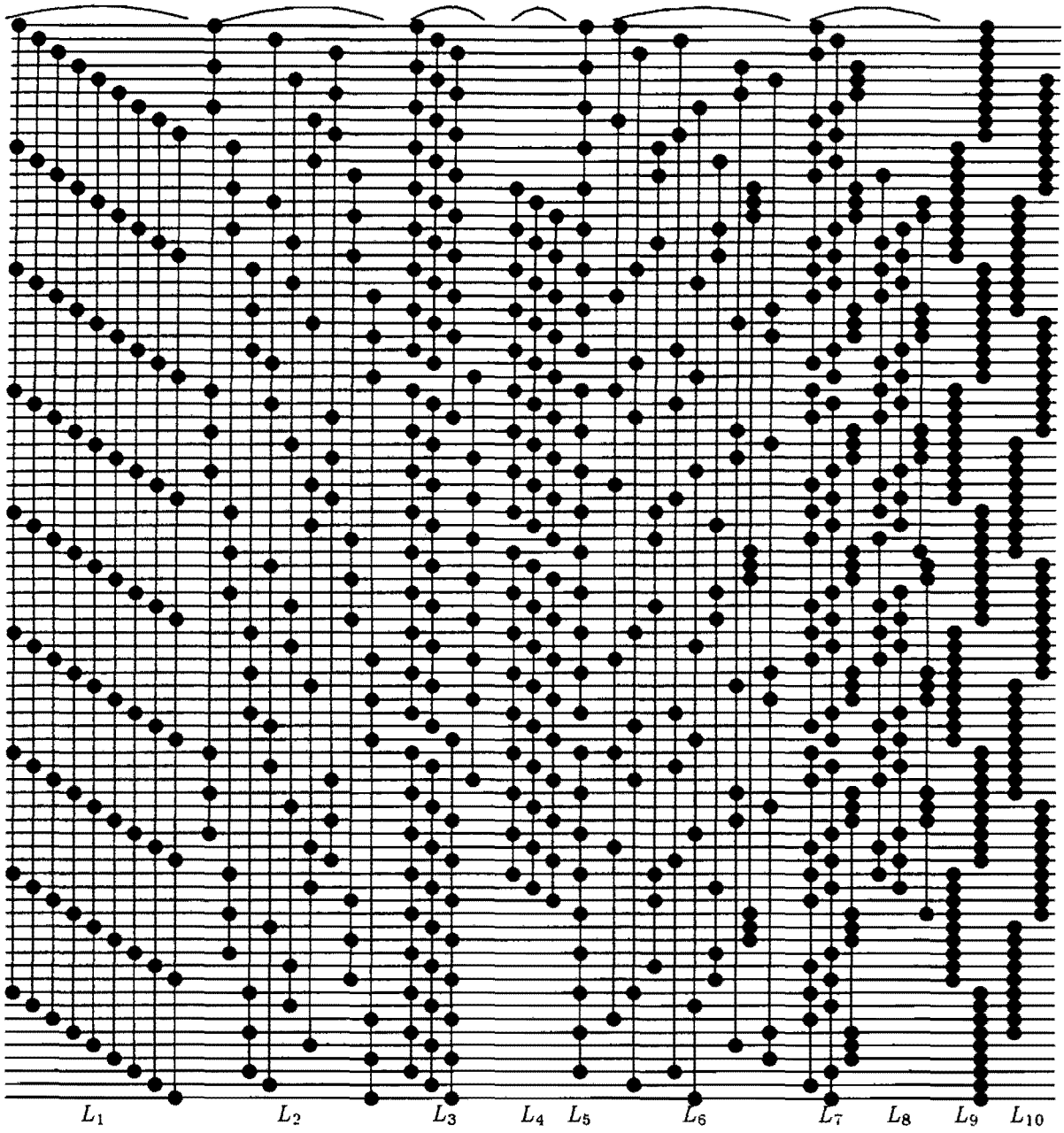


Figure 4.18: An example of the line representation of a network of size 81 constructed from the multiple merge sorting algorithm. There are 10 levels of 9-comparators in this network (the extra level occurred in L_5, where the bitransformer is applied).

(see page 67 for the calculation) and $k = q^2$.

$$\begin{aligned} T_k^s(N) &= \sum_{i=0}^{\lceil \log_q \frac{N}{q^2} \rceil} T_{q^2}(q^i q^2) \\ &= (\lceil \log_q(N) \rceil - 1)^2 \\ &= (2\log_k(N) - 1)^2. \end{aligned}$$

and we have the following:

$$\begin{aligned} C_k^s(N) &= \sum_{i=0}^{\lceil \log_q \frac{N}{q^2} \rceil} \lceil \frac{N}{q^{i+2}} \rceil C_{q^2}(q^{i+2}) \\ &= \frac{N}{q^2}(\log_q(N) - 1)^2 - \frac{N}{q^2(q-1)}(\frac{N-q}{q-1} - (\log_q(N) - 1)). \end{aligned}$$

4.8 Sundry and Summary

In this section, we want to emphasize some extra observations for this chapter, which were not discussed in the previous sections. If we apply different types of the m-g-chains decompositions, then various types of transformer sub-networks will be required. Furthermore, not all of the m-g-chains decompositions work efficiently with sorted p–chains of V. Here we will briefly touch on how our merge algorithm becomes inefficient, when the wrong type of the m-g-chain is used. The basic idea is this, after the p–chains are sorted, if we sort the wrong type of m-g-chains, the number of clean rows in A is not affected. No rigorous proofs or formal set-up will be presented here, although, we will give enough details to demonstrate the idea.

One such example, is the *left-right*(q)–chains defined in Definition 3.4 for q an even integer. The *left-right*(q)–chain is defined in a manner very similar to how the alt-q–bichain is defined: instead of grouping the elements with respect to the rows, we group the elements with respect to the columns. Now if the i^{th} p–chain and the $(i + \lfloor \frac{q}{2} \rfloor)^{th}$ p–chain each contains the same number of 1's, then sorting the pair of *left-right*(q)–chains in these two columns will not increase the number of clean rows in A. Furthermore, if there are more than $p - q$ 1's in these two columns and the other columns contain only 0's, then we need to apply $O(q^{s-2})$ stages of N–q^2–cotransformer. Under the scenario outlined here, one just needs to make the following trivial observation: the maximum distance that the element 1 in the top row can be moved downwards by successively applying the N–q^2–cotransformer is at most q^2 positions for each iteration. Hence it will take $O(q^{s-2})$ stages of N–q^2–cotransformers to move this 1 to the bottom of the interconnection network, assuming the other elements are 0's.

From the diagonalizing column sort in Leighton [56], we notice that skew (diagonal) blocks of size q^2 can also be used. We then show that by sorting the skew (diagonal) blocks of size q^2 in the first layer of the cotransformer, we are able to implement the p–bichain merge algorithm using two layers of

q^2–comparators (we called such a sub-network the skewtransformer). Hence there might be other types of transformer that we have not investigated.

The characteristic feature of this merge scheme using the m-g-chains decomposition, is the fact that the patterns of the m-g-chains decompositions are regular and they can be easily converted to sort a 3–dimensional interconnection network. We will demonstrate this using the bichain sorting algorithm, after we calculated the time upper bound of each sorting algorithm.

Theorem 4.8.1 *The time complexity of the $N^{\frac{2}{3}}$–cochain merge algorithm on an $N^{\frac{2}{3}} \times N^{\frac{1}{3}}$ interconnection network is $5N^{\frac{2}{3}} + O(N^{\frac{1}{3}})$.*

Proof: To calculate the time complexity of the cochain merge algorithm, we will refer back to the description of the procedure of this sorting algorithm on page 88.

The algorithm first sorts the columns of size $N^{\frac{2}{3}}$, where this can be done by the odd-even transposition sort in $N^{\frac{2}{3}} + N^{\frac{1}{3}}$ time cycles. Next we will need to sort the $N^{\frac{2}{3}}$–cochains on a planar interconnection network. Since each processor only connects to its four immediate neighbours, there is no direct connection between elements in the cochain from different levels. Hence elements will need to be transported between levels, this will need $N^{\frac{2}{3}}$ time cycles. Furthermore, since there is no diagonal connection, we will need to move the elements in a step-like manner (for example, to move an element down the diagonal, we need to move the element to the right in the interconnection network then down the interconnection network). Thus it takes $N^{\frac{2}{3}}$ time cycles to move elements in the diagonal manner. We then apply the odd-even transposition sort to sort the bichain, which can be done in $N^{\frac{2}{3}} + N^{\frac{1}{3}}$ time cycles.

Following Lemma 4.2.4, we know that the resulting interconnection network has at most $N^{\frac{1}{3}}(N^{\frac{1}{3}} - 1)$ elements in successive positions that need to be sorted still. So the odd-even transposition sort sorts in less than $N^{\frac{2}{3}} + N^{\frac{1}{3}}$ time cycles. So the algorithm will need $5N^{\frac{2}{3}} + O(N^{\frac{1}{3}})$ time cycles. ∎

The cochain algorithm needs to use $5N^{\frac{2}{3}} + O(N^{\frac{1}{3}})$ time cycles, under the basic model, where each processor is connected to at most four of its nearest neighbour processors without the wraparound connection. However, if the mesh-type interconnection network has horizontal wraparound and diagonal connections then it is possible to reduce the time complexity to $3N^{\frac{2}{3}} + O(N^{\frac{1}{3}})$. The time lower bound of the rectangular mesh-type interconnection network, with horizontal wraparound and diagonal connections, has the same leading term as the rectangular mesh-type interconnection network with diagonal connections. This is because the horizontal wraparound reduces the number of time cycles of the row operations by a factor half, but it does not contribute to the column operations. Following the reasoning of Schnorr and Shamir [89], the time lower bound of the rectangular mesh-type interconnection network with diagonal connections is $2N^{\frac{2}{3}}$, which means that this algorithm is $N^{\frac{2}{3}} + O(N^{\frac{1}{3}})$ more than the optimal time.

Theorem 4.8.2 *The time complexity of the $N^{\frac{2}{3}}$-bichain merge algorithm on an $N^{\frac{2}{3}} \times N^{\frac{1}{3}}$ interconnection network is $4N^{\frac{2}{3}} + O(N^{\frac{1}{3}})$.*

Proof: To calculate the time complexity of the bichain merge algorithm, we will refer back to the description of the procedure of this sorting algorithm on page 94.

The algorithm first sorts the columns of size $N^{\frac{2}{3}}$, using the odd-even transposition sort in $N^{\frac{2}{3}} + N^{\frac{1}{3}}$ time cycles. Next we need to sort the $N^{\frac{2}{3}}$-bichains on the planar interconnection network. Since each processor only connects to its four neighbours, there is no direct connection between elements in the bichain from different levels. Hence elements will need to be transported between levels, this will need $N^{\frac{2}{3}}$ time cycles. We apply the odd-even transposition sort to sort the bichain, this can be done in $N^{\frac{2}{3}} + N^{\frac{1}{3}}$ time cycles.

Following Lemma 4.3.3, we know that the resulting interconnection network has at most $N^{\frac{1}{3}}(N^{\frac{1}{3}} - 1)$ elements in successive positions that need to be sorted still. So the odd-even transposition sort sorts in less than $N^{\frac{2}{3}} + N^{\frac{1}{3}}$ time cycles. So the algorithm will need $4N^{\frac{2}{3}} + O(N^{\frac{1}{3}})$ time cycles. ∎

The usual connecting models used on planar interconnection networks (such as the diagonal connection, the wraparound connection) do not seem to improve on the time upper bound of this algorithm. So the time complexity of the bichain merge algorithm seems to be the worst amongst the new sorting algorithms on two-dimensional interconnection networks. However, under closer inspection, this algorithm is most readily to be converted to sort on 3-dimensional interconnection networks. One of the reasons is that the bichain merge has very good packaging, compared to the algorithms introduced in the previous chapter. The reason that some of the algorithms in the previous chapter are not good candidates to implement on 3-D interconnection networks is because rewiring operations are needed for each algorithm. The type of rewiring operations used are operations that reverse the rows, skew the interconnection network, and transposes the columns into rows. These operations are non-trivial operations on a 3-D interconnection network. Without getting into too much detail, we will describe how the bichain merge is implemented on the 3-D interconnection networks.

For simplicity, we will illustrate the method on an $N^{\frac{1}{3}} \times N^{\frac{1}{3}} \times N^{\frac{1}{3}}$ interconnection network. The elements in the interconnection network are indexed as illustrated in Figure 4.19 and we also indicate the directions of the 3-D co-ordinates (the x-axis, the y-axis and the z-axis). See Figure 4.20 for an illustration of the algorithm. The algorithm first sorts the columns of a 2-D interconnection network, which are the vertical cross sections of the 3-D interconnection network on the yz-plane (Step 1 of Figure 4.20). Next the algorithm sorts the bichains of a 2-D interconnection network, which are the vertical cross sections of the 3-D interconnection network on the xz-plane (Step 2 of Figure 4.20). Next, we briefly describe how to implement the bitransformer on a 3-D interconnection network: the first step of the bitransformer is to sort the horizontal cross sections of the 3-D interconnection network on the xy-plane (Step 3 of Figure 4.20). Next the procedure regards the 3-D interconnection network as a partition of

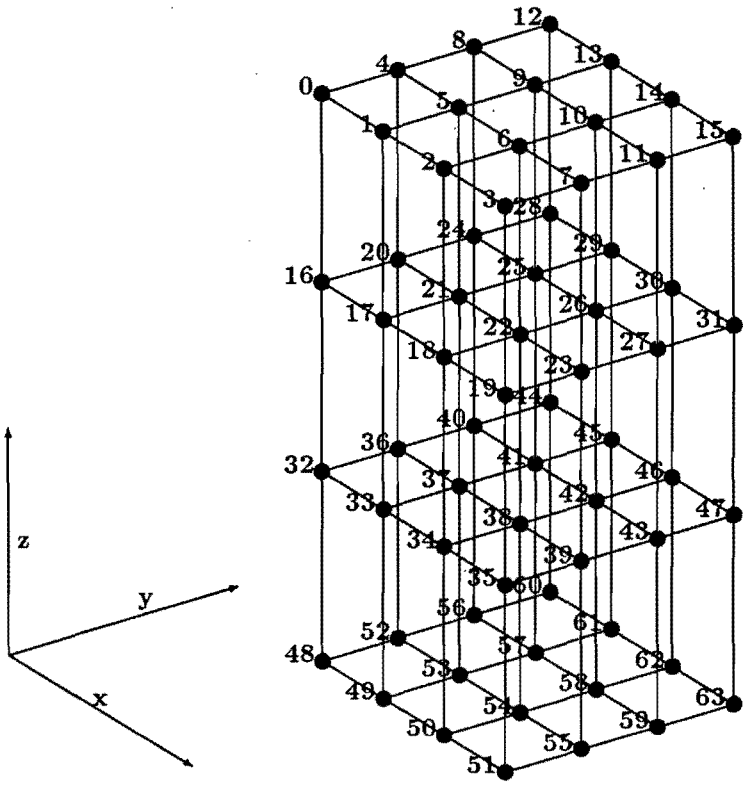


Figure 4.19: An example to illustrate the indexing scheme on a 3–D interconnection network.

two equal parts on the xz–plane, where the sub-interconnection network that starts with index 0 is risen by one layer (Step 4 of Figure 4.20). Again, the procedure sorts the horizontal cross sections of the 3–D interconnection network on the xy–plane (Step 4 of Figure 4.20). Next the procedure slides the top half layer backwards and the bottom half layer forwards, then these two sub-interconnection networks are directed back into the original 3–D block shape (Step 5 of Figure 4.20). Finally, the procedure sorts the horizontal cross sections of the 3–D interconnection network on the xy–plane one last time (Step 6 of Figure 4.20).

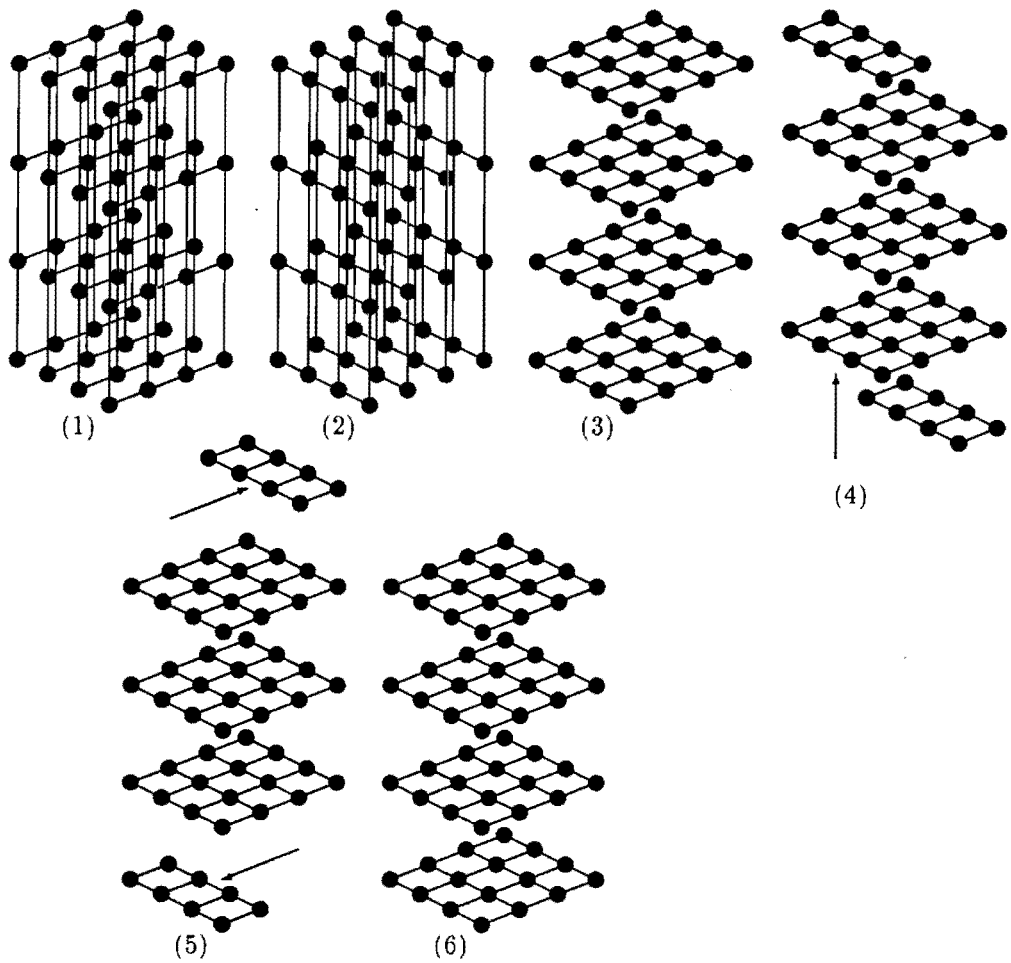


Figure 4.20: An example illustrates the procedures of the bichain merge using the bitransformer on a 3–D interconnection network.

A similar procedure can be used to describe the skewtransformer. We will give a very brief description and illustrate this in Figure 4.21. The skewtransformer slides all the elements that belong to the first skew block in the top layer forwards and all the elements that belong to the last skew block in the bottom layer backwards. Then those vertical columns which contain the remaining elements from the first layer of the 3–D interconnection networks are dropped down by one layer; and those vertical columns which

contain the remaining elements from the last layer of the 3–D interconnection networks are raised up by one layer. Next the procedure sorts the horizontal cross sections of the 3–D interconnection network on the xy–plane. Next, we reverse the operation on the vertical columns and the slide operation. Next the procedure regards the 3–D interconnection network as a partition of two equal parts along the xz–plane, where the sub-interconnection network that starts with index 0 is raised by one layer. Finally, the procedure sorts the horizontal cross sections of the 3–D interconnection network on the xy–plane.

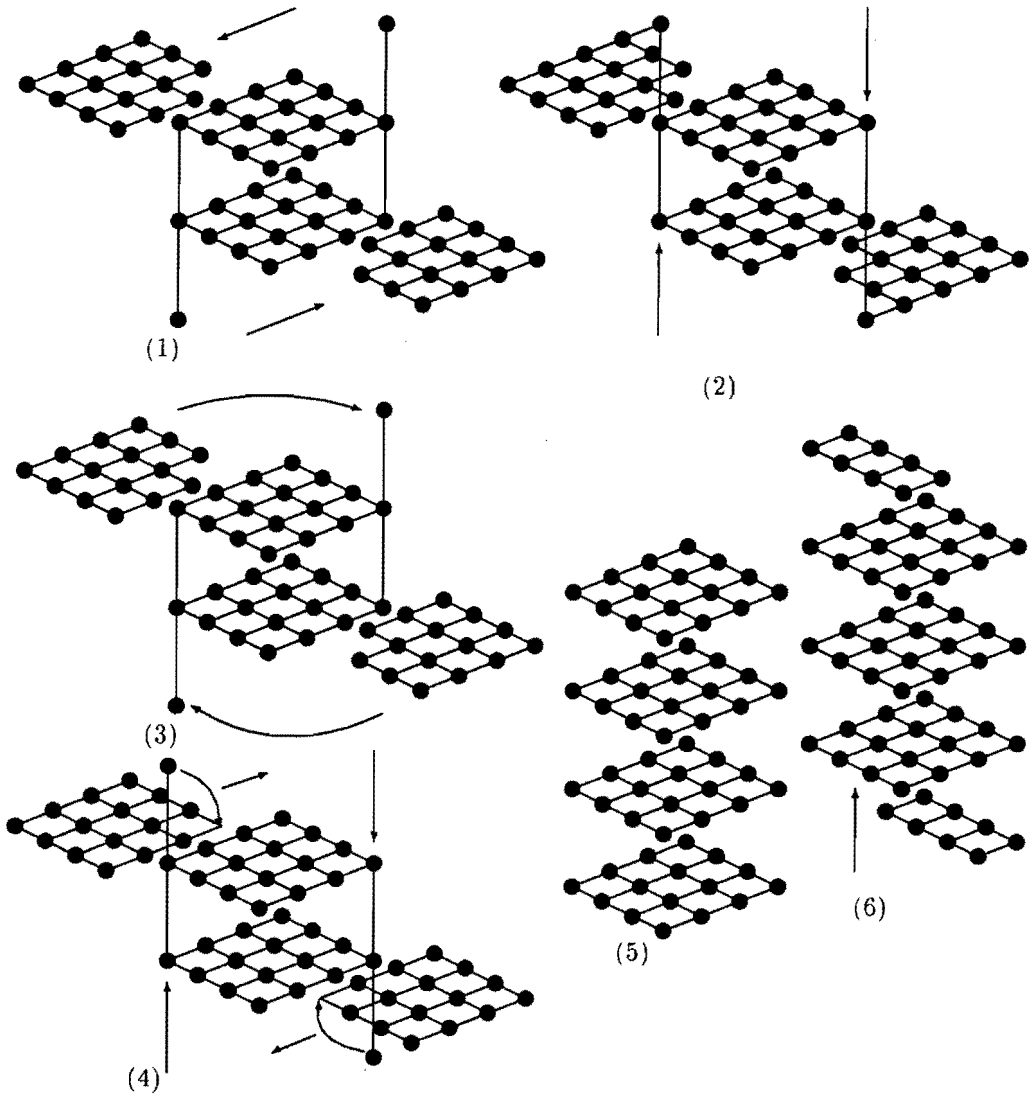


Figure 4.21: An example that illustrates how the procedures of the skewtransformer is implemented on a 3–D interconnection network. The procedure sorts in Step 2 and Step 6 of the diagram.

As demonstrated in the previous few paragraphs, the procedure of the skewtransformer on 3–D interconnection networks without any further modification will be more complicated than the bitransformer.

Under our interpretation of these two transformers, the bichain merge using the bitransformer uses less operations to sort 3–D interconnection networks and are more efficient than the bichain merge using the skewtransformer to sort 3–D interconnection networks. Finally, if the processing units of the 3–D interconnection network are capable of storing data, then we can design a model for the 3–D interconnection networks that is analogous with the architecture of the MIMD model on the 2–D interconnection networks. This implies the implementation of the 3–D algorithms can be simplified: instead of shifting or sliding the sub–interconnection network around, we can just send the required data in the processor unit to the queue of a processor unit in the appropriate sector, for example the xy–plane. We shall not get too involved with sorting on 3–D interconnection networks here, however, further study on sorting for the 3–D interconnection networks will need to be investigated. More discussion on the study of sorting on higher dimensions interconnection networks can be found in Section 1.9 of Leighton [58]. As for the aspect of implementing the algorithm under the MIMD model, we will just refer the reader to the paper of Sibeyn [91], where various algorithms that sort planar interconnection networks are presented there. those algorithms will give some indications of how the 3–D algorithms can be implemented on processor units that are able to queue.

Chapter 5

The Double Butterfly Network is Rearrangeable for $N \leq 32$

In the previous two chapters, we examined some sorting and merging using comparators with size k. In this chapter, we investigate a property of switching networks using standard 2-input and 2-output switches, where we will demonstrate the properties that link the switching network to the 2-comparator network. There were intensive studies done on the 2-comparator networks and the switching networks, as early as the 60's. This unabated interest in the analysis of 2-comparator networks and the switching networks is generated by the intrinsically simple structure of the switches (or 2-comparators), and the diverse networks that can be generated by interconnecting them, such as the Clos network, the Baseline network, the Benes network and the Shuffle/Exchange network.

A switch and a 2-comparator are similar in structure, since both devices have two input terminals and two output terminals. Although, in the mathematical perspective, a switch is more complex than a 2-comparator: a switch can be regarded as a relation and a 2-comparator can be regard as a function, we shall give a more detailed explanation in the next section. Hence the complexity of a problem of a switching network can be reduced if we are able to use the properties of a 2-comparator network to assist in solving the problem. In this chapter we try to prove, using the known properties of the Batcher's Bitonic sorting network (a comparator network), that the double butterfly network (a switching network) is rearrangeable.

We shall use the following example to demonstrate the importance of the rearrangeability of a network. In parallel computer systems with several interconnection networks, the interconnection networks consist of multiple stages (or levels) of switching elements. The interconnections of switching networks provide the communication paths among the processes and are generally required to perform permutations of data between their input and output. Hence it is of vital importance, that the networks are capable of passing all the $N!$ permutations of N elements. In the literature, there are various studies done on the

rearrangeability of various networks, such as Feng and Seo [38], Lee [55], Yeh and Feng [108], Varma and Raghavendra [98].

A switching network is rearrangeable if its permitted states (switch states) realize any one-to-one assignment of input points to output points. For example, the Benes network is proved to be rearrangeable in Leighton [58] and the Double Baseline network is proved to be rearrangeable in Even and Kupershtok [36]. However, the question of whether the double butterfly network is rearrangeable seems not to have been answered (Parker [74] has proved that the three butterfly network is rearrangeable and Raghavendra and Varma [81] have proved that five stages of the Shuffle/Exchange network is rearrangeable for $N = 8$).

We will investigate whether the double butterfly network is rearrangeable. The double butterfly network consists of two copies of the butterfly network in tandem. We will prove that the double butterfly network is rearrangeable for $N \leq 16$, however, we postulate that the proof can be extended to any $N > 16$ where $N = 2^n$, because of the symmetric structure between the upper half and the lower half of the butterfly network.

We present a control algorithm for finding the states of the switches for an arbitrary permutation, for the double butterfly networks of size N, where $N \leq 16$. In 1987, Raghavendra and Varma [81] published an algorithm for $N = 8$, where the algorithm used five stages of Shuffle/Exchanges to produce the required switch setting for any arbitrary permutation. A proof and a control algorithm is presented in their paper. Their algorithm is based on a method that produces the required partition of the connections corresponding to a given permutation. I found that their algorithm is not as easy to comprehend as our algorithm, because their algorithm needs to find conflict free partitioning of the connections over the whole network (from the input terminals to the output terminals). Where as in our algorithm, we only need to be concerned with the first copy of the butterfly network, the bitonic sorting algorithm will generate the required state for each switch in the second copy of the butterfly network.

More concisely, the majority of calculation in our new algorithm is focused on dealing with the switches of the first copy of the butterfly networks. The goal is to find a switch set connection for the corresponding permutation, such that the first butterfly sub-network outputs a recursively balanced vector. Next, the switches of the second butterfly sub-network is chosen to sort the recursively balanced vector, by applying the bitonic sorting algorithm. Since the switches in the second copy of the butterfly network are regarded as 2–comparators, we are able to reduce the complexity of the problem.

Furthermore, Raghavendra and Varma [98] were able to use the result for $N = 8$ to show that for the network to be rearrangeable it is sufficient to use $3\log_2(N) - 4$ layers of Shuffle/Exchange for $N > 8$, without finding the control algorithm. We improved on their results, and construct a control algorithm for finding the states of the switches for arbitrary permutations, for the case $N = 16$. We then present an outline of how this method can be extended for $N = 32$. Hence following from Theorem 3.3 of Raghavendra and Varma [98], we can improve the sufficient condition to $3\log_2(N) - 5$.

Another major difference between the works published in the literature and our new work is in the presentation style. Majority of the past works used the modular representation of the Shuffle/Exchange network, however, we are using the line representation in our algorithm.

This chapter is organized as follows: In section 1, we introduce some new definitions and terminology for the double butterfly network. We also introduce some of the relevant theorems that incorporate Batcher's Bitonic sorting network into the butterfly network. In section 2, we explain in detail our approach and introduce an important procedure, the split procedure. The split procedure is used to set-up the required states of a level of switches in a butterfly network. In section 3, we prove that the $\mathbf{B_8}$ and the $\mathbf{B_{16}}$ network are balanced, hence we can deduce that the $\mathbf{2 \times B_8}$ network and the $\mathbf{2 \times B_{16}}$ network are rearrangeable. In section 4, we present a routing algorithm to show that the $\mathbf{2 \times B_8}$ network and the $\mathbf{2 \times B_{16}}$ network are rearrangeable. In section 5, we discuss the possibility of extending the algorithm for higher order N, where $N \geq 32$ and the prospect of future work on this problem.

5.1 Introduction and Definitions

To assist in the formulation of the double butterfly problem, some definitions and notation from the paper of Bilardi [21] will be introduced. We will denote the double butterfly network of size N by $\mathbf{2 \times B_N}$ and the butterfly network of size N by $\mathbf{B_N}$. We gave a brief description of the butterfly network in the introduction chapter, however, the butterfly network is also defined in Definition 5.6.

A switch is a device with two input terminals and two output terminals, which receives values x and y and produces values x' and y' respectively. In a switch, there is a binary state s that can be set independently of the inputs: If $s = 0$ (the off state) then $x' = x$ and $y' = y$, else $s = 1$ (the on state) and $x' = y$ and $y' = x$. Given a switching network S of size N, we assume that both the input terminals and the output terminals are numbered from 0 to $N-1$. We say that a sequence $\vec{x} = (x_0, x_1, \ldots, x_{N-1})$ is the input of S when x_i is input at terminal i. Similarly, $\vec{y} = (y_0, y_1, \ldots, y_{N-1})$ is the output sequence, if y_i is output at terminal i. We denote by $S(\vec{x})$ the output of network S corresponding to input $\vec{x}$.

The output $\vec{y}$ of a switching network S is a fixed permutation of the indices of the input $\vec{x}$ (independent of the input), that is $\vec{y} = S(\vec{x}) = (x_{\tau(0)}, x_{\tau(1)}, \ldots, x_{\tau(N-1)})$, where $\tau = (\tau(0), \tau(1), \ldots, \tau(N-1))$ is a permutation of $(0, 1, \ldots, N-1)$. The permutation τ is a function of the switch states for a switching network. So, if a switch setting generates a permutation, then we should say that the switch states *realize* the permutation. If the switch state S realizes the permutation τ we will index the corresponding switch setting as S_τ. Hence, when it is not ambiguous, we will also consider S_τ as a permutation operator.

Without loss of generality, **we shall consider only input sequences that are permutations of** $(0, 1, \ldots, N-1)$, **in this chapter**. Also we will be using word sequence and vector interchangeably. Hence an N–vector can be regarded either as a sequence of numbers or as a permutation of the indices.

We do not distinguish them, when it is clear to which underlying structure we are referring. We follow the notation of Berman and Frayer [16] to represent permutation. The product $\tau\pi$ of two permutations τ and π on the set $\{0, 1, \ldots, N-1\}$ is defined to be the single permutation on the set equivalent to the successive performance of permutations first τ then π. So we have the following:

$$\begin{aligned} S_\tau(\pi) &= S_\tau(\pi(0), \pi(1), \ldots, \pi(N-1)) \\ &= (\tau(\pi(0)), \tau(\pi(1)), \ldots, \tau(\pi(N-1))) \\ &= \tau\pi \end{aligned}$$

Now we are ready to prove our first result.

Lemma 5.1.1 *$\mathcal{N}$ is a rearrangeable switching network if and only if for any N-vector input there exist switch states in $\mathcal{N}$ such that the network outputs a sorted N-vector, denoted by ι .*

Proof: If $\mathcal{N}$ is a rearrangeable switching network then this will be trivially true using the definition. Now, suppose for any input N-vector, π, there exists a switch state $S_{\pi^{-1}}$ in $\mathcal{N}$ which outputs ι (here π and ι are regarded as permutations on the set $\{0, 1, \ldots, N-1\}$). Now, if π is an arbitrarily chosen input and τ is an arbitrarily chosen output, we need to show that there exits a switch state in $\mathcal{N}$ which transforms π into τ. By assumption, we know there exits a switch state $S_{(\pi\tau^{-1})^{-1}}$ where $S_{(\pi\tau^{-1})^{-1}}(\pi\tau^{-1}) = \iota$. this implies $S_{(\pi\tau^{-1})^{-1}}$ realizes the permutation $(\pi\tau^{-1})^{-1}$. So we have the following

$$\begin{aligned} S_{(\pi\tau^{-1})^{-1}}(\pi) &= (\pi\tau^{-1})^{-1}\pi \\ &= \tau\pi^{-1}\pi \\ &= \tau \end{aligned}$$

This proves the lemma. ∎

Lemma 5.1.1 establishes the fact that to show the $\mathbf{2 \times B_N}$ is rearrangeable, one just needs to show that the $\mathbf{2 \times B_N}$ sorts any permutation. A permutation is *admissible* for a switching network, if there is a setting of the switches which sorts the permutation.

The next Proposition is proved in Bilardi [21]. Here we state the Proposition without providing the proof.

Proposition 5.1.2 *A permutation π is sorted by Batcher's Bitonic sorting network [8] if and only if π is in the class of permutations admissible for the Omega network.*

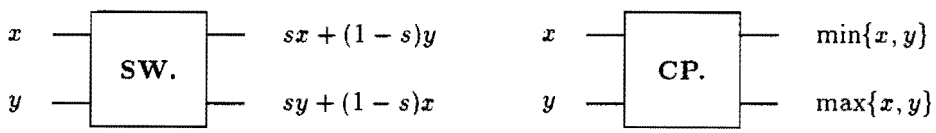
The Omega network (also known as the Shuffle/Exchange network) is really the same as the butterfly network ($\mathbf{B_N}$) (Leighton [58]). $\mathbf{2 \times B_N}$ consists of two copies of $\mathbf{B_N}$'s in tandem. Therefore, we shall

regard each butterfly as a sub-network with different functionality. If the vector is sorted by $\mathbf{2 \times B_N}$ then the first $\mathbf{B_N}$ will output an N-vector which can be sorted by Batcher's Bitonic Sorting network. From Proposition 5.1.2. the output N-vector is admissible for the second $\mathbf{B_N}$. We need to show that given any vector there is a switch setting for a single $\mathbf{B_N}$, such that the output is admissible for $\mathbf{B_N}$. Hence we have the following theorem.

Theorem 5.1.3 *If the butterfly network can generate admissible N-vectors for any input N-vector, then the double butterfly network is rearrangeable.*

Proof: Follows from Lemma 5.1.1 and Proposition 5.1.2, hence this theorem is proved. ∎

The ability to regard some of the switches as the 2-comparators can simplify the complexity of the double butterfly network problem. Although a 2-comparator is similar in structure to a switch, the functionality of a 2-comparator is included in the functionality of a switch: A 2-comparator is a device with two input terminals and two output terminals, which receives values x and y and produces values x' and y' respectively, where $x' = \min\{x, y\}$ and $y' = \max\{x, y\}$. See Figure 5.1 for the illustration of a switch and a 2-comparator. However, this more rigid form of outputs by the 2-comparators implies there are fewer outcomes to consider, hence it reduces the complexity in solving this problem. We will discuss how we can regard a 2-comparator as a switch, in the later part of this chapter.



(a) A switch, with state $s = 0$ or 1. (b) A 2-comparator.

Figure 5.1: A diagrammatic representation of a switch and a 2-comparator.

The characteristics of the input N-vectors which are sorted by the Bitonic merging network are investigated in Perl [77]. The next three definitions and Theorem 5.1.4 are from Perl [77]. Again, we will state the theorem without proving it.

Definition 5.1 *Suppose $N = 2^k$ and V is an N-vector then the 2^i-chains, $1 \leq i \leq k-1$, of V is defined inductively as follows:*

- *Suppose the elements of V are indexed binary representation. Then the sub-sequence of the elements in V with least significant bit of the index equal to 0 is called a 2^{k-1}-chain and it is said to be the* **even** *chain of V. A 2^{k-1}-chain is said to be the* **odd** *chain of V, if the sub-sequence of the elements with least significant bit of the index equal to 1 in V. Let* $\mathbf{Even}(V)$ *be the even 2^{k-1}-chain and* $\mathbf{Odd}(V)$ *be the odd 2^{k-1}-chain of a sequence V.*

- *The 2^i-chains are the even and the odd chains of the 2^{i+1}-chains of V.*

Note: N does not need to be restricted to be a number of power of 2, however, the 2^i-chains generated for such N will not have equal length. Actually Definition 5.1 is a special case of Definition 4.1 of (n,p)-chains, where we consider $n = 2$. However, we include Definition 5.1 here to give a complete overview on this section.

Definition 5.2 *A sequence V is* **balanced** *if, when elements are arranged in non-decreasing order and then partitioned into pairs, one element from each pair lies in* **Even**(V) *and the other element lies in* **Odd**(V).

Definition 5.3 *A sequence V is* **recursively balanced** *if it is balanced and both its* **Even**(V) *and* **Odd**(V) *are recursively balanced.*

Theorem 5.1.4 *The Bitonic network of size N sorts an N-vector if and only if the N-vector is recursively balanced.*

Hence, to show that $\mathbf{2 \times B_N}$ is rearrangeable, it is sufficient to show that the first copy of $\mathbf{B_N}$ can output a recursively balanced N-vector for any input N-vector.

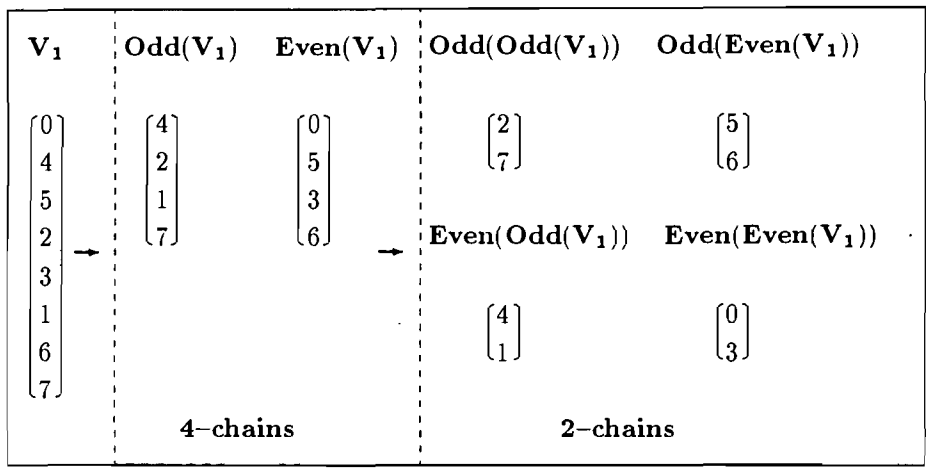
Definition 5.4 *A switching network is* **balanced**, *if the switching network can generate a recursively balanced sequence for any arbitrary input sequence.*

Next, we will introduce new terminology that will simplify the presentation when we show that $\mathbf{B_N}$ is balanced. Given any N-vector V, we will call a pair of elements of V *co-ranked elements* of V, if the pair of elements is from the same partition set of the sorted N-vector of V, where the sorted N-vector is partitioned into consecutive pairs.

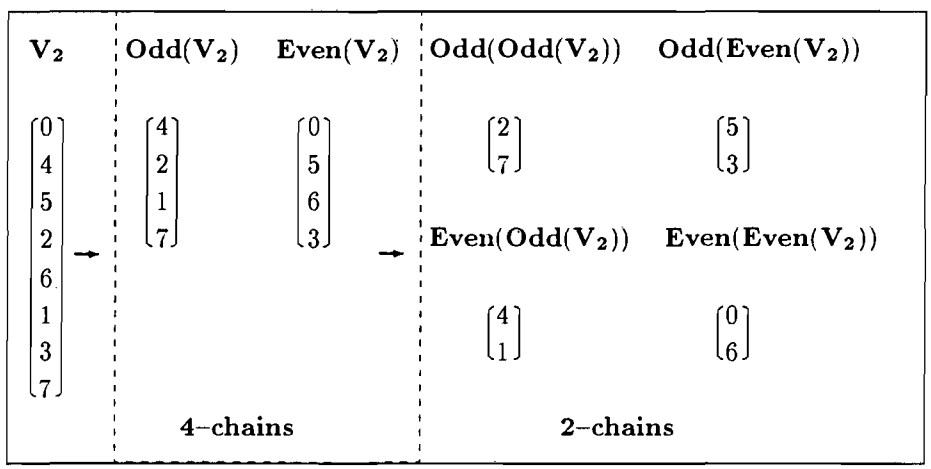
Let $V = (v_0, v_1, \ldots, v_{N-1})$ be an N-vector with distinct integral elements. Any two elements of V with indices i and $i+1$, where i is an even integer in the range of 0 to $N-2$, will be called *co-neighbours* (Alternatively, we say that they are from the same *co-neighbourhood*).

Henceforth we use $\bar{V} = (\bar{v}_0, \bar{v}_1, \ldots, \bar{v}_{N-1})$ **to denote the N-vector with elements of V sorted in increasing order**. Then the pairs of co-neighbours of $\bar{V}$ are the sets of co-ranked elements in V, namely $\{\bar{v}_0, \bar{v}_1\}, \{\bar{v}_2, \bar{v}_3\}, \ldots, \{\bar{v}_{N-2}, \bar{v}_{N-1}\}$.

Note: The pairs of co-neighbours of an N-vector V depend on the indices of the elements in V. However, the pairs of co-ranked elements of an N-vector V depend on the ranks of the elements in V; the pairs of co-ranked elements of any V depend on the indices of the elements in $\bar{V}$.



(a)



(b)

Figure 5.2: (a) V_1 is a balanced but not recursively balanced 8–vector, since the even chain is not balanced. However, the odd chain is balanced. (b) V_2 is a recursively balanced 8–vector.

The Definition 5.3 of recursively balanced N–vectors is a recursive one. The notion of pairs of co-ranked elements can be extended to the subvector. For example, the pairs of consecutive elements of the 2^{n-1}–chains, the pairs of co-ranked elements of the 2^{n-2}–chains. In the next definition we will give a formal description:

Definition 5.5 *Given an N–vector V, where elements of V are distinct integral numbers $0, 1, \ldots, N-1$ and $N = 2^n$. Then the elements of V can be partitioned systematically into 2^{n-k} sets that contain 2^k elements ($1 \leq k \leq n$) as follows. If the binary representation of the elements of V has the same first $n-k$ significant bits, then the elements belong to the same quotient-2^k set. For example, $\{i, i+1, \ldots, i+2^k-1\}$ where $i \in \{0, 2^k, 2^{k+1}, \ldots, N-2^k\}$. Such sets will be called the* **quotient**–2^k *sets of V, or simply denote as $Q_{2^k}^N$ sets. Note that the $Q_{2^k}^N$ sets has 2^k elements and the superscript ^{N} denote this $Q_{2^k}^N$ set is from an N–vector.*

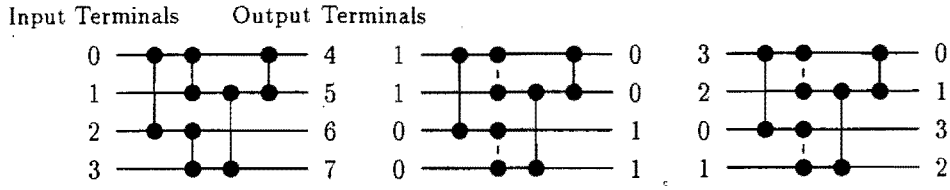
For example, in any 16–vector, the Q_2^{16} sets are $\{0,1\}, \{2,3\}, \ldots, \{14,15\}$ (note: each set contains a pair of co-ranked elements). The Q_4^{16} sets are $\{0,1,2,3\}$, $\{4,5,6,7\}$, $\{8,9,10,11\}$ and $\{12,13,14,15\}$. This notation can be extended to any N which is a power of 2. In the sorting problem, the *rank* of an element in V is equal to the index of the same element in $\bar{V}$ ($\bar{V}$ is the sequence obtained from sorting V). So we shall call an element a *high element* of V if this element has a rank ranging from $N/2$ to $N-1$. Similarly we shall call an element a *low element* of V if this element has a rank ranging from 0 to $N/2-1$. Similarly, the $Q_{2^k}^N$ sets can be used to classify the elements in an N–vector.

In the next lemma, we will derive an alternative way to describe a recursively balanced N–vector, using the notion of the elements of $Q_{2^k}^N$ sets where $1 \leq k \leq n$. The converse statement of Lemma 5.1.5 is also true (it is an if and only if statement), but we do not need it here (Note: this alternative statement is just an iterative definition for the recursive definition).

Lemma 5.1.5 *For every $Q_{2^k}^N$ set and $\forall k \in \{1, 2, \ldots, n\}$, if the elements of the $Q_{2^k}^N$ set are in different 2^{n-k}–chains of the N–vector (there are 2^k 2^{n-k}–chains) then the N–vector is recursively balanced.*

Proof: We will prove this lemma inductively on the length of the input N–vector. It is trivially true for $N = 2$. Suppose this is true for any N–vector, where $N \leq 2^n$. Now consider the case for $N = 2^{n+1}$: We will need to show that such N–vector is balanced. Furthermore, we need to show that the 2^{n-1}–chains are recursively balanced.

It is clear that the N–vector is balanced, since every pair of co-ranked elements has one element from the even chain and from the odd chain. It is clear from Definition 5.1 that every 2^{n-i}–chain contains 2 $2^{n-(i+1)}$–chains. Hence the iterative property holds for the even chain and the odd chain, so by hypothesis the 2^{n-1}–chains are recursively balanced. Since the choice of even integer N is arbitrary, the lemma is proved by the inductive reasoning. ∎



(a) A switching network. (b) Sorts this 0-1 vector. (c) Does not sort this vector.

Figure 5.3: Here is a counter example which illustrate the $[0-1]$ **principle** can not be applied on switching networks.

Now we have all the necessary facts to assert our approach in solving the rearrangeability of the double butterfly network. However, there is one last point we want to clarify: In our approach, the first copy of $\mathbf{B_N}$ in the double butterfly network is regarded as a switching network, hence we will not be able to apply the $[0-1]$ **principle** on the switching sub-network, to assist in solving the problem. A switching network is ineligible to incorporate the $[0-1]$ **principle**, because a switch can not be regarded as a monotonic function, unlike the 2-comparator. So if a switching network with N input lines can sort any sequences of 0's and 1's into non-decreasing order, it will not guarantee to be able to sort every sequence of N numbers into non-decreasing order. We present a counter example here to illustrate this: a switching network with 4 input lines shown in Figure 5.3 is a network that can sort all 2^4 sequences of 0's and 1's into non-decreasing order, however it is not able to sort any arbitrary sequence of four numbers into non-decreasing order. We label the input terminals of this network from 0 to 3 and the output terminals of this network from 4 to 7. The fact that it does not sort is clear from the observation that there is no path that connects the input terminal 1 and the output terminal 6, hence any input sequence with $x_1 = 3$ will not be sorted.

5.2 The Double Butterfly is Rearrangeable Problem

Shuffle/Exchange networks have been shown to provide an efficient interconnection scheme for implementing various types of parallel processes, such as in Chen et al [26], Kumar and Reddy [51], Lawrie [54], Wu and Feng [103] [104], Patel [76], Steinberg [92], Lee [55], Raghavendra and Boppana [80], Yeh and Feng [108] and Feng and Seo [38]. However, to the best of my knowledge, the question of whether the double butterfly network is rearrangeable seems not have been answered. Here is a list of papers that investigated on the rearrangeability of the Shuffle/Exchange network: in 1980, Parker [74] published a paper showing that a network implemented by three copies of butterfly networks (or $3\log_2 N$ of Shuffle/Exchanges) is rearrangeable. In 1981, Wu and Feng [105] improved on this bound and showed that a network of $3\log_2(N) - 1$ of Shuffle/Exchanges is rearrangeable. In 1987, Raghavendra and Varma [81] published a control algorithm for $N = 8$, where five stages of Shuffle/Exchanges are used. Their network uses one less level of switches than the double butterfly network, which uses six levels of switches.

Then Raghavendra and Varma [98] showed that if there is an algorithm for the double butterfly network of a given N', then for any $N > N'$, it is sufficient to use $3\log_2(N) - \log_2(N')$ stages of Shuffle/Exchanges. Since they have proved that there is a control algorithm for $N' = 8$, it follows that it is sufficient to use $3\log_2(N) - 4$ stages of Shuffle/Exchanges.

Throughout this chapter, we will use N–vectors of A's and B's to generalize certain classes of N–vectors. An N–vector with a pattern of A's and B's, either means all the A's are high elements and the B's are low elements in this N–vector, or the converse is true.

Theorem 5.2.1 *An 8–vector with the following pattern $(ABAB\ BABA)$ is recursively balanced, providing that every two elements at positions i and $i+2$, where $i \in \{0,1,4,5\}$, are from different Q_2^8 sets.*

Proof: Clearly, any 8–vector that satisfies the condition given in this theorem is balanced, since its 2^2–chains each contains two high elements from different Q_2^8 sets and two low elements from different Q_2^8 sets.

It is also obvious that the even chain and the odd chain (the 2^2–chains) is balanced, since every 2–chain contains one low element and one high element. Hence the 8–vector is recursively balanced. ∎

Theorem 5.2.2 *A 16–vector with the following pattern $(ABABABAB\ BABABABA)$ is recursively balanced, providing that it satisfies the following conditions:*

1. *Elements occupying positions i, $i+2$, $i+4$ and $i+6$, where $i \in \{0,1,8,9\}$, contains no two elements that are from the same Q_2^{16} set.*

2. *Elements occupying positions i and $i+4$, where $i \in \{0,1,2,3,8,9,10,11\}$, are not from the same Q_4^{16} set.*

Proof: Any 16–vector which satisfies condition (1) of Theorem 5.2.2 is balanced, since the upper half of either 2^3–chain is occupied by four high elements or four low elements. Furthermore, the four elements in the lower half and the four elements in the upper half of each 2^3–chain are from different Q_8^{16} sets.

If we regard each 2^3–chain as an 8–vector that satisfies condition (2) of the statement, then each pair of elements of a Q_4^{16} set can be regarded as a pair of elements of a Q_2^8 set, in the 8–subvector. So we can use the argument used in Theorem 5.2.1. Thus the even chain and the odd chain are both recursively balanced. Therefore, the 16–vector is recursively balanced. ∎

We would like to emphasize here, that not every recursively-balanced vector is an instance of the pattern we are using. For example, $(0,1,2,3,4,5,6,7)$ is obviously recursively balanced. However, it has a $(AAAABBBB)$ pattern.

Next we give a formal description of the $\mathbf{B_N}$ network. We will follow the notation of Knuth [49], to represent a switching network as an ordered set of N input lines (drawn horizontally) connected by a set of switches (drawn as vertical connection between the lines). Now we can recursively define the $\mathbf{B_N}$ network.

Definition 5.6 *The Butterfly network* $\mathbf{B_N}$, *where* $N = 2^n$, *has* n *levels of switches and the horizontal input lines are indexed from 0 to* $N-1$. *The first layer (level 1) of the* $\mathbf{B_N}$ *consists of* $N/2$ *switches connecting line* i *and* $i+n/2$, $0 \leq i < n/2$. *The level 1 switches are connected to two parallel copies of the* $\mathbf{B_{N/2}}$. *The* i^{th} *input line of the upper* $\mathbf{B_{N/2}}$ *is the* i^{th} *output line of the previous level and the* j^{th} *input line of the bottom* $\mathbf{B_{N/2}}$ *is the* $(j+N/2)^{th}$ *output line of the previous level. (Notice that the* $\mathbf{B_2}$ *network is just two consecutive input lines and one switch connecting them.)*

5.2.1 The Split Procedure

In this section, we will state the split procedure and prove its correctness. The split procedure is used to produce the required states of the switches in one level of the butterfly network. Given any input N–vector V, the procedure moves one element from each pair of co-ranked elements in V to the upper half of the N–vector, and the corresponding element is moved to the lower half of the N–vector. The split procedure will be the main operation that will generate the recursively balanced N–vector.

Let v_p be an element in the p^{th} position of sequence V.
Let v_p^* be the co-ranked element of v_p in V.
Let v^+ be the element at the other end of the switch which contains v.

SPLIT PROCEDURE

Split(V)
begin
if there are unmarked switches **then**

1. $v \leftarrow v_p$;
2. Set the switch so as to send v to the top half and mark the switch
3. **if** $(v^+ = v_p^*)$ **then**
   Split{elements on the unmarked switches} ; **Stop**
4. **else** $v \leftarrow (v^+)^*$, **goto** 2.

endif
end

Theorem 5.2.3 *The split procedure will terminate in a finite number of steps. The resulting output sequence will satisfy the following: The upper half of the resulting sequence and the lower half of the resulting sequence each contains one co-ranked element from each pair. The procedure sets the switches on $M/2$ switches for input sequence of length M (M is assumed to be any positive even integer).*

Proof: We will prove this theorem using induction on the input length of sequence V. It is trivially true for any input sequence of length 2, since there is only one switch, the algorithm marks the switch and exits. Assume the algorithm terminates and outputs the correct k–vector, for all the input sequences of length $k \leq N-2$, where N is an even integer. Now, consider the case for input sequences of length N: If the algorithm is called at line 3, the length of the input vector is $\leq N-2$ and hence the algorithm terminates and is correct, by inductive hypothesis. If the algorithm is not called at line 3, then the algorithm executes line 4 and loops. In each loop, the number of unmarked switches decreases by one. Since there are a finite number of switches, this implies the algorithm will eventually terminate. Furthermore, the output sequence is correct. Suppose the last switch marked has x on the top and y on the bottom. Since line 3 of the algorithm is not executed, we know that $x \neq y^*$, and none of the bottom elements before the last switch is v_p^* or v_p. Thus none of the top elements is v_p^*, and thus y must be v_p^*. Hence the algorithm is correct. ∎

5.3 The B_8 and the B_{16} Network are Balanced

In this section, the butterfly network is proved to be balanced, for the case $N = 8$. By Definition 5.4, this implies that for any input N–vector there exists a switch setting in the butterfly network, which outputs a recursively balanced N–vector.

The Balanced B_8 Algorithm

(Note: We need to record the state of each switch, so we can use it in the $2 \times B_8$ algorithm)

1. Input an 8–vector V.

2. Use the split procedure to produce the level 1 switch setting and split V. Let $V_1 = \mathbf{Split}(V)$. (Using the first $N/2$ switches.)

3. Use the split procedure to produce a switch setting at the upper half of level 2. We use the split procedure to produce a switch setting at the lower half of level 2. Let U be the resulting 8–vector (the joining of the 2 halves).

4. Use the level 3 switches to do the following:

   (i). In the upper half of U, we use the first two switches to move all the high elements to the

top of the switches and all the low elements to the bottom of the switches.

(ii). In the lower half of U, we use the last two switches to do the opposite (low elements to the top and the high elements to the bottom).

Theorem 5.3.1 *The algorithm terminates. The resulting 8-vectors of the algorithm are recursively balanced.*

Proof: The algorithm terminates, since the split procedure terminates.

In **Step 2**, the split procedure outputs V_1, where V_1 has two high elements and two low elements in each half. Furthermore, the elements in each half are from different Q_2^8 sets.

In **Step 3**, the algorithm makes sure that every pair of consecutive neighbours in U is occupied by one high element and one low element.

In **Step 4**, the algorithm arranged the resulting 8-vector into the pattern of Theorem 5.2.1. Hence, by Theorem 5.2.1, the resulting 8-vector is recursively balanced. ∎

Now that we have proved that the $\mathbf{B_8}$ network is balanced, we shall continue extending this elegant method to prove that the $\mathbf{B_{16}}$ network is balanced. A similar approach can be used, since the split procedure works on any even integer length vector. The approach is to first split the input 16-vector into an upper 8-subvector and a lower 8-subvector, using the first level of switches. After the split procedure, we classify the resulting 16-vector and apply the appropriate scheme. The algorithm checks each 8-subvector (the upper 8-subvector and the lower 8-subvector): How many low elements are there in the even chain of the 8-subvector?

If there is no low element or there are four low elements in the even chain of the 8-subvector, then the 8-subvector will be said to be an instance of the type 4 combination. If there is one low element or there are three low elements in the even chain of the 8-subvector, then the 8-subvector will be said to be an instance of the type 3 combination. If there are two low elements in the even chain of the 8-subvector, then the 8-subvector will be said to be an instance of the type 2 combination.

Hence any 16-vector resulting from the split procedure in level 1 will belong to one of the following classes: $(4 = 4)$, $(4 = 3)$, $(4 = 2)$, $(3 = 3)$, $(3 = 2)$ and $(2 = 2)$. Note: $(p = q)$ implies that the 16-vector is an instance of the type p 8-subvector interleaved with an instance of the type q 8-subvector (the notation does not express the relative positions of the 8-subvectors in the 16-vector). For example a 16-vector V from the class of $(3 = 2)$ would mean one of the following is true:

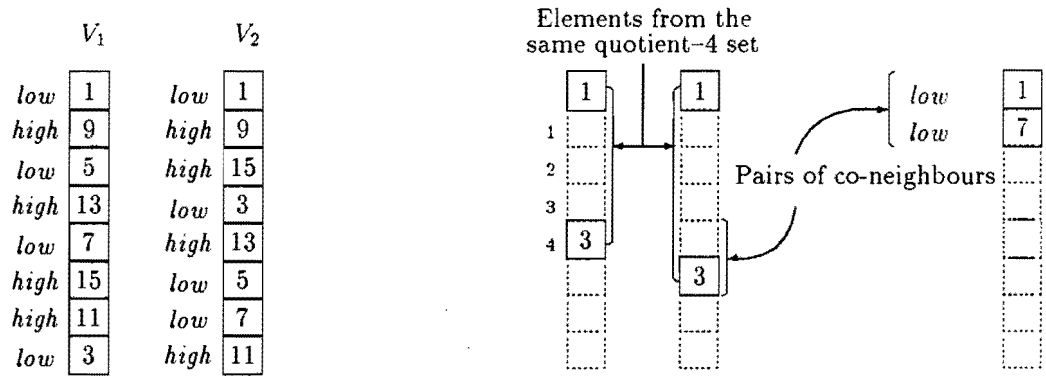
- The even chain of V is an instance of the type 3 8-subvector and the odd chain of V is an instance

of the type 2 8–vector.

- The odd chain of V is an instance of the type 3 8–subvector and the even chain of V is an instance of the type 2 8–vector.

We will show that for an 8–subvector of type 2, 3 or 4, there is a switch setting which derives the desired pattern given in Lemma 5.3.2. We focus our attention on the 8–subvector, because the butterfly network has a symmetric structure, between the upper half and the lower half of the network. Hence, after the split procedure in level 1, each half of the switching network can be operated on independently as two copies of $\mathbf{B_8}$. (An in-depth discussion on how smaller butterfly networks are embedded in the larger butterfly network is given in Bermond, Frayer and Jean-Marie [17]). Furthermore, the resulting pattern of the 16–vector of Theorem 5.2.2 implies the following: If we can find a procedure which transforms the upper 8–subvectors, then the same procedure can transform the lower 8–subvectors also.

New terminology is introduced here to facilitate our explanation: Any switch in a given level is said to be **locked** if and only if the corresponding pair of elements connected by the switch at the input terminals is from the same $Q^N_{2^{n-1}}$ set where $N = 2^n$ (either both are high elements or both are low elements). Alternatively, we simply say that the pair of elements is locked in the given level. We use the word **locked**, because if the pair of elements is processed by the switch in the given level, the result of this switch will not contribute to the change in the pattern of the A's and B's.



(a) 2 matched 8–subvectors

(b) Some 8–subvectors that are not matched.

Figure 5.4: (a) Two examples of matched 8–subvectors. (b) Two possible instances where an 8–subvector is not matched.

For any 16–vector resulting from the split procedure using the level 1 switches, the upper half and the lower half of the resulting 16–vector each contains two elements from each Q^{16}_4 set. Henceforth, we will refer to the upper half or the lower half of such 16–vector as an 8–subvector of the resulting 16–vector, unless it is stated otherwise.

Definition 5.7 *If x* **matches** *y in an 8–subvector V (the upper half or the lower half of an 16–vector)*

then the following two conditions hold:

1. *x and y are elements from the same Q_4^{16} set in V. If x is in the i^{th} position of V, then y is not in the $(i+4)^{th}$ position of V nor in the co-neighbourhood of the $(i+4)^{th}$ position of V.*

2. *x and x's co-neighbour are from different Q_8^{16} sets and similarly for y.*

Note: The notion of **match** is a symmetric one, for example x matches y is equivalent to y matches x.

Furthermore, we say the 8–subvector is **matched** if every pair of elements from the same Q_4^{16} set is matched in the 8–subvector.

In level 2 of the B_{16} network, if two elements are locked in one of the level 2 switches, this implies that the pair of elements in the i^{th} position and the $(i+4)^{th}$ position are either both high elements or both low elements, where $i \in \{0,1,2,3,8,9,10,11\}$.

After applying the split procedure to the 16–subvector using the level 1 switches, there are four high elements and four low elements in each of the 8–subvector of the resulting 16–vector (the upper 8–subvector and the lower 8–subvector, see Figure 5.5 for an illustration). Note that each pair of elements from the Q_4^{16} set in such an 8–subvector are regarded as a pair of co-ranked elements in the 8–subvector (elements of the Q_2^8 set). In this section we will extend the notation of the superscript $*$ to denote pairs of co-ranked elements in the 8–subvector, and this should not be confused with pairs of co-ranked elements in the 16–vector.

Lemma 5.3.2 *If an 8–subvector is "matched", then we can use the level 4 switches to get the pattern (ABAB ABAB) and (BABA BABA). Furthermore the resulting 8–vector satisfies condition (2) of Theorem 5.2.2.*

Proof: Suppose the 8–subvector is matched, then a high element and a low element occupy every pair of co-neighbours. Therefore, the level 4 switches will be able to move all the high elements to occupy the even positions and all the low elements to occupy the odd positions or vice versa. Hence, the desired pattern is achieved.

Assume that the 8–subvector does not satisfy condition (2) of Theorem 5.2.2, then there exists at least one element x in the j^{th} position and the corresponding element y from the same Q_4^{16} set is in the $(j+4)^{th}$ position. Without loss of generality, assume j is an even integer, so x occupies the j^{th} or the $(j+1)^{th}$ position of the input terminals of the level 4 switches. However, the 8–subvector is matched, therefore y can not be in the $(j+4)^{th}$ or the $(j+5)^{th}$ positions. Since the level 4 switches are not able to move y into those positions, y must be in one of those position already, but this is a contradiction. ∎

Suppose the 8–subvector is an instance of the type 2 combination, which implies that there are two

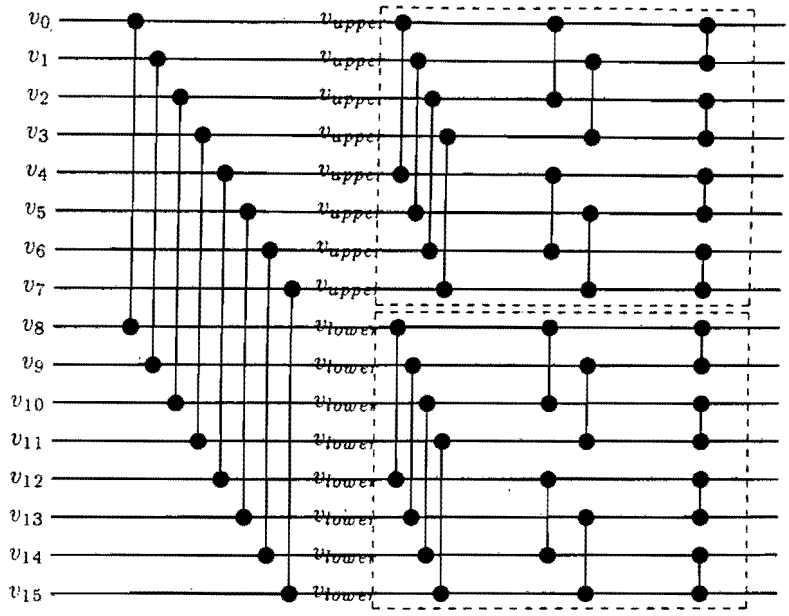


Figure 5.5: A diagram to illustrate the upper 8–subvector and the lower 8–subvector.

high elements and two low elements in each 2^2–chain of the 8–subvector.

In this section, when we split an 8–subvector using the level 2 switches, we are constructing a vector that has equal number of high elements (low elements) in each half. And when we split elements in any half of the 8–subvector, then we are using the level 3 switches to split the pair of high elements and the pair of low elements of that half of the 8–subvector.

Next, we define a procedure for setting the level 3 switches: Take one of the halves of an 8–subvector that is arranged by the level 3 switches, let this half be denoted by H_s and other half we will denote by H_m. Now we can construct the following procedure to match the elements of H_m to the elements of H_s, using level 3 switches.

Procedure Match-2-halfs(H_s, H_m)

Split the pair of high elements and the pair of low elements in H_m satisfying the following condition:

If $x \in H_s$ and $x^* \in H_m$, where x and x^* are from the same Q_4^{16} set **then**

If x is in the upper half of H_s **then**

x^* is sent to the lower half of H_m

Else

x^* is sent to the upper half of H_m

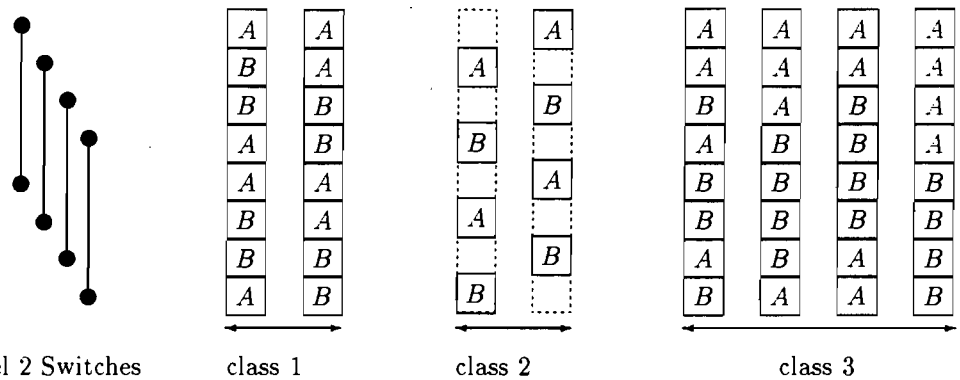


Figure 5.6: Here are some examples of three different classes of the instances of each type 2 classification. on the upper half or the lower half of the level 2 switches in $\mathbf{B_{16}}$.

We want to further classify the instances of the type 2 8–subvectors:

1. Both pairs of the low elements are locked in level 2.

2. Only one pair of the low elements is locked in level 2.

3. None of the low elements is locked in level 2.

Actually, further observations can be made using this classification, which we will prove in the following lemmas.

Lemma 5.3.3 *For any instance of the type 2 8–subvector, one pair of the low elements of the odd (even) chain is locked in level 2 if and only if one pair of the high elements of the same 2^2–chain is locked in level 2.*

Proof: Suppose that two of the low elements or 2 of the high elements are locked in one of the 2^2–chains. Since each 2^2–chain of the 8–subvector is four elements long, respectively the other two positions must be occupied by the other two high elements or the other two low elements. Furthermore, they are locked also. ∎

Follow immediately from Lemma 5.3.3, we have the following corollaries. We shall state the corollaries without giving the proofs:

Corollary 5.3.4 *In the level 2 switches, two pairs of the low elements are locked in the type 2 8–subvector if and only if two pairs of the high elements are locked in the type 2 8–subvector.*

Corollary 5.3.5 *In the level 2 switches, none of the low elements are locked in the type 2 8-subvector if and only if none of the high elements are locked in the type 2 8-subvector.*

It is easier to find the switch setting for any instance of the type 2 8–subvector by using this extra classification for the type 2 8–subvectors. In the following section, we will prove the existence of switch setting for each type. In all the cases, the scheme for finding the switch setting for each individual case is outlined in the proof of the lemma.

Lemma 5.3.6 *Given an instance of the class 1 type 2 8-subvector V. If the elements of at least one of the pairs of the locked level 2 elements in V are from different Q_4^{16} sets, then we can use the level 2 switches and the level 3 switches to produce a matched 8-subvector.*

Proof: Suppose we have an instance of the class 1 type 2 8–subvector, then from Corollary 5.3.4 we know that all four pairs of elements in level 2 are locked. Furthermore, if one pair of elements of those four locked pairs contains elements from different Q_4^{16} sets, then the corresponding elements (other elements from those 2 Q_4^{16} sets) are also locked together in one of the level 2 switches. Without loss of generality, suppose the elements are high elements (similar argument applies if the elements are low elements). We use this pair of level 2 switches to send one pair of elements from the same Q_4^{16} set to the upper half of the 8–subvector. In the process, the other pair of elements from the other Q_4^{16} set is sent to the bottom half of the 8–subvector.

So after applying the level 2 switches, each half of the 8–subvector contains two high elements from the same Q_4^{16} set. Furthermore, one of those high elements is in the even chain and the other one is in the odd chain. Hence every level 3 switch links one high element and one low element. Note that the upper half of the 8–subvector and the lower half of the 8–subvector have the same pattern with respect to the position of the high and the low elements.

It is clear that it is possible to derive a matched 8–subvector from these patterns. We choose to do the following: in the upper half of the 8–subvector, we use the level 3 switches to split the pair of high elements and the pair of the low elements. Next, we will need to split elements in the lower half of the 8–subvector also. However, we need to match these elements with respect to the elements in the upper half. This can be accomplished in the manner described by the procedure Match-2-halfs on page 142. ∎

Lemma 5.3.7 *Given an instance of the class 1 type 2 8-subvector V. If each pair of the locked level 2 elements in V are from the same Q_4^{16} set, then the switches in the level 2 and in the level 3 of the* $\mathbf{B_{16}}$ *network can produce a matched 8-subvector.*

Proof: For any instance of the class 1 type 2 8–subvector, if the elements of each locked level 2 pair are from the same Q_4^{16} set, then any setting of the state of such level 2 switches will not have any effect

on the pattern of such an 8–subvector, since those positions occupied by the high elements (the odd elements) will remain occupied by the high elements (the odd elements). Furthermore, the upper half of the 8–subvector and the lower half of the 8–subvector will have the same pattern, with respect to the relative positions of the elements from each Q_4^{16} set.

So we use the level 2 switches to split the elements from each Q_4^{16} set, then we use the level 3 switches to arrange elements, such that every co-neighbourhoods in the upper half is occupied by a low element and a high element. Next, we set the corresponding level 3 switches in the lower half, using the opposite state. For example, if the switch that connecting the j^{th} position and the $(j+2)^{th}$ position, where $j = 0, 1$ (in the upper half), is set to the **on** state (exchange elements), then the switch that connecting the $(j+4)^{th}$ position and the $(j+6)^{th}$ position (in the lower half) is set to the **off** state. Hence the resulting 8–subvector is matched. ∎

Lemma 5.3.8 *Any instance of the class 2 type 2 8–subvector can be transformed into a matched 8–subvector, using the switches in level 2 and in level 3.*

Proof: From Lemma 5.3.3, we know that the elements in one of the 2^2–chains (either the odd or the even chain) of the 8–subvector are locked; it has a pair of locked high elements and a pair of locked low elements. Now there are two possible cases to consider.

If the elements of each locked pair are from the same Q_4^{16} set, then we use the level 2 switches to split the 8–subvector. Each level 3 switch receives elements from the same 2^2–chain of the 8–subvector. This implies that any level 3 switch, which receives one element from the outputs of the locked level 2 switch as the input, will receive one high element and one low element as inputs. Furthermore, the high elements are from the same Q_4^{16} set and the low elements are from the same Q_4^{16} set. So one of these two switches sends the high element to the top and the other switch sends the high element to the bottom. Next the rest of the level 3 switches will insure that a low and a high element occupy each pair of the co-neighbours.

If the elements of each locked pair are not from the same Q_4^{16} set, then we let A_p and A_q be the elements of a locked pair, which are elements of different Q_4^{16} sets. Now we can use the level 2 switches to move both A_p and A_p^* to the top half of the 8–subvector. Simultaneously, the level 2 switches move A_q and A_q^* to the bottom of the 8–subvector. Note: There are two high elements and two low elements in each half of the resulting 8–subvector. Since each pair of A's from the same Q_4^{16} set is in the same half of the 8–subvector, none of the A's can violate condition (2) of Theorem 5.2.2. Hence we can use the level 3 switches to match the B's, and every pair of the co-neighbours is occupied by a low and a high element.

In both cases, the resulting 8–subvector is matched. ∎

Lemma 5.3.9 *Any instance of the class 3 type 2 8-subvector can be transformed into a matched 8-subvector, using the switches in level 2 and in level 3.*

Proof: From Corollary 5.3.5, we know that none of the level 2 switches is locked. Hence we use the level 2 switches to move all the high elements from one of the Q_4^{16} set to the upper half of the 8-subvector and all the high elements from other Q_4^{16} set to the lower half of the 8-subvector. Note that we could have chosen to use the low elements here, and applied similar arguments. Again, each pair of high elements from the same Q_4^{16} set is in the same half of the 8-subvector, thus none of the high elements will violate condition (2) of Theorem 5.2.2. Hence, we use the level 3 switches to match the low elements. Next, we use the level 3 switches to split the upper half of the 8-subvector, so that the larger low element is moved to the top and the smaller low element is moved to the bottom. Furthermore, the level 3 switches split the lower half of the 8-subvector, so that the larger low element is moved to the bottom and the smaller low element is moved to the top. ∎

Lemma 5.3.10 *Any instance of the type 2 8-subvector can be transformed into a matched 8-subvector, using the switches in level 2 and in level 3.*

Proof: From Lemma 5.3.6, Lemma 5.3.7, Lemma 5.3.8 and Lemma 5.3.9, we know Lemma 5.3.10 is true, since the classification of type 2 8-subvectors describe every instance of the type 2 8-subvector. ∎

Procedure For a Type 2 8-subvector

Identify the class of the type 2 8-subvector: (1) Class 1; (2) Class 2; (3) Class 3.

Class 1: *There are two possible outcomes for the class 1 subvectors:*

If *there exist a locked level 2 switch that contains elements from different* Q_4^{16} *sets* **then**

Use the level 2 switches to split the elements in the 8-subvector such that it sends pairs of elements from the same Q_4^{16} *set to the same half, when it is possible to do it without affecting the process of splitting the elements. We then use the level 3 switches to split the elements in the upper half of the 8-subvector. Next we apply the procedure Match-2-half(the upper half, the lower half).*

Else

Use the level 2 switches to split the elements in the 8-subvector, where we split every pair of elements from the same Q_4^{16} *set. We then use the level 3 switches to split the elements in the upper half of the 8-subvector. Next we apply the procedure Match-2-half(the upper half, the lower half).*

Class 2: *There are also two possible outcomes for the class 2 subvectors:*

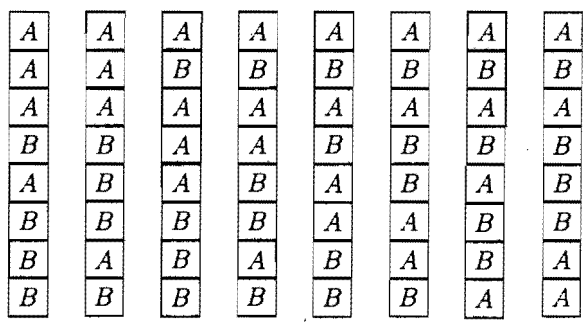
If *elements of each locked level 2 switch belong to the same* Q_4^{16} *set* **then**

Use the level 2 switches to split the elements in the 8-subvector, where we split every pair of elements from the same Q_4^{16} set. We then use the level 3 switches to split the elements in the upper half of the 8-subvector. Next we apply the procedure Match-2-half(the upper half, the lower half).

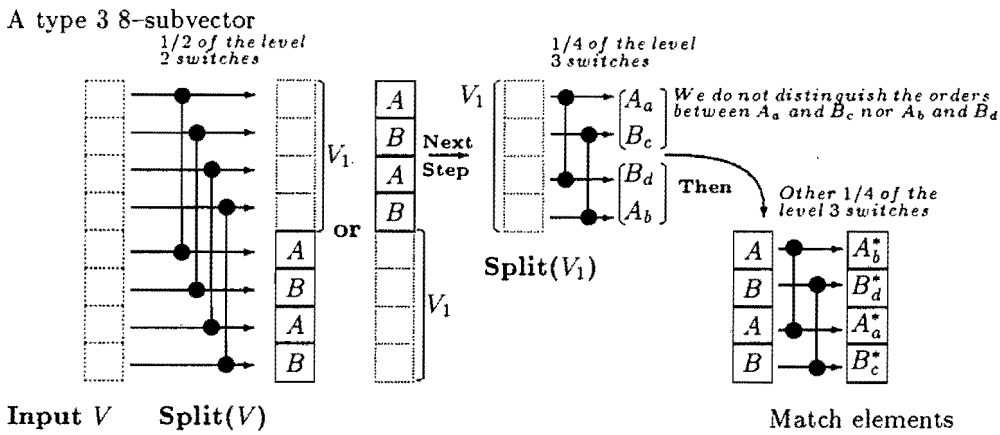
Else

Use the level 2 switches to split the elements in the 8-subvector such that the locked level switch that contains the pair of elements from different Q_4^{16} sets sends the elements from the same Q_4^{16} set to the same half, when it is possible to do it without affecting the process of splitting the elements. We then use the level 3 switches to split the elements in the upper half of the 8-subvector. Next we apply the procedure Match-2-half(the upper half, the lower half).

Class 3: *Use the level 2 switches to split the elements in the 8-subvector such that they send pairs of high elements from the same Q_4^{16} set to the same half. Then we use the level 3 switches to split the elements in the upper half of the 8-subvector. Next we apply the procedure Match-2-half(the upper half, the lower half).*



(a) All the possible patterns for the type 3 8-subvectors.



(b) How to derive a matched 8-subvector from a type 3 8-subvector.

Figure 5.7: In (b), we use superscripts $*$ some of the A's and the B's to indicate that, in the resulting 8-subvector, the pair of corresponding elements with the same index is from the same Q_4^{16} set (they are regarded as pairs of co-ranked elements (the Q_2^8 sets) in the 8-subvector).

Lemma 5.3.11 *Any instance of the type 3 8–subvector can be transformed into a matched 8–subvector, using the switches in level 2 and in level 3.*

Proof: For any instance of the type 3 8–subvector, there are three high elements in one of its 2^2–chains and three low elements in the other 2^2–chain. Hence, after the level 2 switches are used to split the 8–subvector, one of the halves of the resulting 8–subvector will have two pairs of elements which are locked in level 3: one pair of locked high elements and one pair of locked low elements. Next, if we use the level 3 switches to split the other half, then it is trivial to derive a matched 8–subvector. See Figure 5.7 for more details. ∎

Procedure For a Type 3 8–subvector

Use the level 2 switches to split the elements in the 8–subvector, where we split every pair of elements from the same Q_4^{16} set. Next, we use the level 3 switches to split the elements in the half that contains no locked level 3 pair. Next we apply the procedure Match-2-half(the half that is split, the other half).

Lemma 5.3.12 *Any instance of the type 4 8–subvector can be transformed into a matched 8–subvector, using the switches from level 2 and level 3.*

Proof: It is obvious that we can derive a matched 8–subvector, since one of its 2^2–chains contains the high elements only and the other 2^2–chain contains the low elements only. Thus, if the high elements are matched and the low elements are matched, then the 8–subvector is also matched (just use the split procedure). See Figure 5.8 for more details. ∎

Procedure For a Type 4 8–subvector

Use the level 2 switches to split the elements in the 8–subvector, where we split every pair of elements from the same Q_4^{16} set. Then we use the level 3 switches to split the elements in the upper half of the 8–subvector. Next we apply the procedure Match-2-half(the upper half, the lower half).

Again, like in the case of $N = 8$, we shall prove the butterfly network is balanced for the case $N = 16$. By Definition 5.4, this implies that for any input N–vector there exists a switch setting in the butterfly network, which outputs a recursively balanced N–vector. The next algorithm will produce the required switch setting for any input 16–vector.

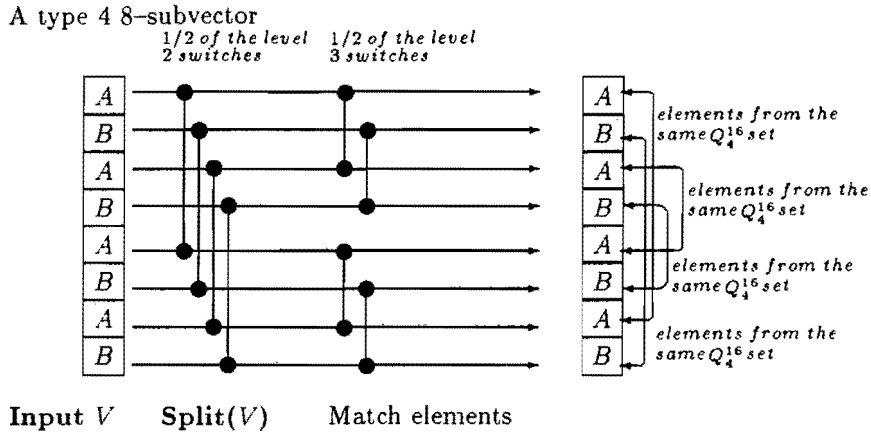


Figure 5.8: A diagrammatic illustration of how the type 4 8–subvector can be transformed into a matched 8–subvector.

The Balanced $\mathbf{B_{16}}$ Algorithm

(Note: We need to record the state of each switch, so that it can be used in the $\mathbf{B_{16}}$ algorithm)

1. Input the 16–vector V.

2. Use the split procedure to produce a switch setting for the level 1 switches and split V. Let $V_1 = \mathbf{Split}(V)$.

3. Classify the upper 8–subvector and the lower 8–subvector of V_1 (type 2, type 3 or type 4). If the 8–subvector is an instance of the type 2 8–subvector, then further classification is needed (class 1, 2 or 3). For each half of V_1, we use the level 2 switches and the level 3 switches to do the following: Apply the appropriate scheme based on the classification to derive a matched 8–subvector: refer to procedure on page 146 for a type 2 subvector; refer to procedure on page 148 for a type 3 subvector; refer to procedure on page 148 for a type 4 subvector. Let U be the resulting 16–vector (the concatenation of the matched upper 8–subvector and the matched lower 8–subvector).

4. Use the level 4 switches to do the following:

   (i) In the upper half of U, use the first four switches to move all the high elements to the top of the switches and all the low elements to the bottom of the switches.

   (ii) In the lower half of U, use the last four switches to do the opposite (low elements to the top and the high elements to the bottom).

Theorem 5.3.13 *The algorithm terminates. The resulting 16–vectors transformed by the algorithm are recursively balanced.*

Proof: It is obvious that the algorithm terminates. By Theorem 5.2.3, we know that the split procedure produces the required switch setting for the level 1 switches. The procedure splits the input 16–vector, where the co-ranked elements of each individual pair are split and sent to the upper half and to the lower half respectively (there are two elements in each Q_2^{16} set). Therefore, after the split procedure in **Step 1**, we are able to classify the upper 8–subvector and the lower 8–subvector. Our classification describes every possible 8–subvector produced by the split procedure, since there are four high elements and four low elements in each 8–subvector.

By Lemma 5.3.10, Lemma 5.3.11 and Lemma 5.3.12, we know that the level 2 and the level 3 switches can transform V_1 into U, where each half of U is matched. Thus by Lemma 5.3.2, each half of U satisfies condition (2) of Theorem 5.2.2, therefore U also satisfies condition (2) of Theorem 5.2.2.

The level 4 switches in **Step 4** make sure that condition (1) of Theorem 5.2.2 holds and the resulting 16–vector has the desired pattern. Hence the resulting 16–vector is recursively balanced. ∎

5.4 The Rearrangeability Algorithm for $2 \times B_8$ or $2 \times B_{16}$

We describe the Rearrangeability Algorithm for the double butterfly network of size 8 or 16. The algorithm uses the input N–vector and the required output N–vector to produce the required switch states in the double butterfly network, which enables the network to transform the input N–vector into the output N–vector.

The Rearrangeability Algorithm for $2 \times B_8$ or $2 \times B_{16}$

1. Input the input sequence V and output sequence U with length N (8 or 16). Let $W = V \circ U^{-1}$. (We regard V and U as permutations of the indices, therefore W is just the composition of two permutations.)

2. Assign the first copy of the butterfly network to execute The Balanced Butterfly Algorithm for size N on W. Record the switch states of the butterfly network, S_1.

3. Assign the second copy of the butterfly network to execute Batcher's Bitonic Sort for N on the resulting recursively balanced N–vector from **Step 2**. Record the switch states S_2 of the butterfly network.

4. Output $S_1 + S_2$, this is the desired switch setting which maps V onto U, $S_2(S_1(V)) = U$.

In **Step 3**, the algorithm uses the Bitonic Sorting, which uses comparators not switches. However, a comparator and a switch are both devices with two input terminals and two output terminals. We label the output terminals as upper output terminal and lower output terminal respectively, then for any input values a and b, we have the following:

1 The comparator will output $\max\{a, b\}$ in the upper out terminal and $\min\{a, b\}$ in the lower out terminal.

2 The switch will output $sb + (1 - s)a$ in the upper out terminal and $sa + (1 - s)b$ in the lower out terminal, where s is a binary state which is independent of the inputs a and b.

In both cases, the device exchanges the elements or does not exchange the elements. Although the Bitonic sorting network uses comparators only, after the state of a comparator is set (exchange or not), we can consider it as a switch.

Furthermore, we can implement the algorithm more efficiently, because we do not need to wait for the desired switch setting in **Step 4**. Since both The Balanced Butterfly Algorithm and Batcher's Bitonic Sort execute each level of switches independently, we can pass the input vector immediately after each level switch setting is found.

Theorem 5.4.1 *The Rearrangeability Algorithm is correct for* $\mathbf{2 \times B_8}$ *or* $\mathbf{2 \times B_{16}}$.

Proof: The algorithm is correct, since the Balanced Butterfly Algorithm and the Bitonic Sorting Algorithm are both correct. By Lemma 5.1.1, the Rearrangeability Algorithm outputs the mapping for any input N–vector and outputs N–vector in the double butterfly network. ∎

5.5 Conclusion and Extension of the Rearrangeability Algorithm

We postulated in the introduction that the same method can be extend to any $N > 16$ where $N = 2^n$. As an exercise we extend this method for the case $N = 32$ and we show that it is possible to extend the method in Chiang [27]. However, we are not able to include the results here, since the algorithm is very complicated and there are large numbers of sub-cases that need to be considered. In this section, we shall give an outline of the algorithm and a summary of our approach. It is clear from the algorithm used in $\mathbf{2 \times B_8}$ and $\mathbf{2 \times B_{16}}$, that the split procedure in the first level and the resulting patterns from the last level of switches ensured that:

1 The resulting N–vectors are balanced.

2 The resulting N–vectors have balanced 2–chains, where $N = 2^n$.

Thus, if we extend this scheme (in particular for $N = 32$), we know that the resulting 32–vector is balanced and its 2–chains are balanced. But we still need to ensure that the resulting N–vector's 16–chains, ..., 4–chains are balanced also, so that the resulting N–vector is recursively balanced.

Before we get on with the discussion, we want to point that our algorithm does have a shortcoming, it does not recognize any input N–vector, which is already recursively balanced and is not an instance of our standard pattern. The algorithm will still process the N–vector and output it into our standard pattern. We may rectify this by scanning all the input N–vector at the initial stage of the algorithm, however it does not reduce the complexity of the algorithm. Furthermore, this algorithm is not a recursive algorithm, because the network has symmetric upper and lower sub-network structure, but $\mathbf{B_N}$ does not contain symmetric structures in the direction from the input terminal to the output terminal.

In the extension, we will retain the split procedure using the first level switches and the resulting pattern of the last level switches (level 5 in the case of $N = 32$). The rest of the network is used to arrange the N–vector into a pattern where the rest of the sub-chains are also balanced. The last observation is that we cannot exchange elements from different 16–chains before the last level of switches of $\mathbf{B_N}$. We cannot exchange elements from different 2–chains, before the second last level of switches of $\mathbf{B_N}$. We may deduce this inductively.

Theorem 5.5.1 *A 32–vector with the following pattern (ABAB ABAB ABAB ABAB BABA BABA BABA BABA) is recursively balanced, provided it satisfies the following conditions:*

1. *Elements occupying positions $i, i+2, i+4, \ldots, i+14$, where $i \in \{0, 1, 16, 17\}$, contains no two elements that are from the same Q_2^{32} set.*

2. *Elements occupying positions $i, i+4, \ldots, i+12$, where $i \in \{0, 1, 2, 3, 16, 17, 18, 19\}$, contains no two elements that are from the same Q_4^{32} set.*

3. *Elements occupying positions i and $i+8$, where $i \in \{0, 1, 2, \ldots, 7, 16, 17, 18, \ldots, 23\}$, contains no two elements that are from the same Q_8^{32} set.*

Theorem 5.5.1 implies that to output a recursively balanced 32–vector, the algorithm needs to re-arrange the 32–vector into a pattern, before the last level switches (level 5) are used. After applying the split procedure, the upper half of the resulting 32–vector and the lower half of the resulting 32–vector contains four elements from each Q_8^{32} set respectively. Furthermore, those four elements from each individual Q_8^{32} set can be partitioned into two pairs of elements, where the elements from each pair are from the same Q_4^{32} set. So what the algorithm needs to ensure is that:

(1). If an element is in position i, then the corresponding element from the same Q_4^{32} set is not in

any of the following position in the 16–subvector simultaneously: j or j's consecutive neighbour. where j is 4λ positions away from i, $\lambda \in \mathbf{Z}$.

(2). If an element is in position i, then none of the corresponding elements from the same Q_8^{32} set are in the positions j and j's co-neighbour in the 16–subvector simultaneously, where j is 8 positions away from i.

(3). Every pair of elements in the co-neighbourhood must be made of a high element and a low element.

Again, we can consider the upper 16–subvector and the lower 16–subvector individually. We will call a 16–subvector "highly matched" (or "h-matched" for short), if the 16–subvector satisfies these three conditions.

Lemma 5.5.2 *If a 16–subvector is "h-matched", then it is possible to get the pattern (ABAB ABAB ABAB ABAB) and (BABA BABA BABA BABA), using the level 5 switches, which satisfies the conditions stated in Theorem 5.5.1.*

Now we are ready to solve the $\mathbf{2 \times B_{32}}$ problem. Again, continue with the previous approach, we use the first level switches to split the 32–vector. From Lemma 5.5.2, we know such resulting 16–subvectors in the upper half and the lower half can be transformed independently. Each 16–subvector can be classified into categories, where the classification depends on the compositions of the high elements and the low elements in the 8–chains of the 16–subvector. After the split procedure in level 1, there are eight high elements and eight low elements in the 16–subvector. So there is an obvious one-to-one relation between the number of the high elements (or the low elements) in each 8–chain. This implies that we can categorize the 16–subvector using just one of the 8–chains, since we do not need to distinguish the 8–chains in the 16–subvector (a given 8–chain predetermines the composition of the other 8–chains). In Figure 5.9 we will give a diagrammatic representation of how we classified all the possible 16–subvectors.

From the buddy property of the butterfly network, the $\mathbf{B_{32}}$ network consists of a pair of $\mathbf{B_{16}}$ sub-networks, hence after the level 1 switches, we can implement each $\mathbf{B_{16}}$ sub-network independently. Furthermore, each $\mathbf{B_{16}}$ sub-network can be regarded either as the level 2 switches plus two sets of $\mathbf{B_8}$ or as two sets of $\mathbf{B_8}$ interleaved plus the level 5 switches. So we are able to transform any 16–subvector into an h-matched 16–subvector, therefore the resulting 32–vector is recursively balanced. Hence $\mathbf{B_{32}}$ is a balanced network. We can now apply the Rearrangeability Algorithm on page 150 for any 32–vector. thus solving the $\mathbf{2 \times B_{32}}$ problem.

To conclude this chapter, we will like to summarize what we are able to contribute to the process of solving the rearrangeability of the double butterfly network. We introduce a fresh perspective of how to approach this problem, which reduces the complexity of the problem by half. Although this exhaustive

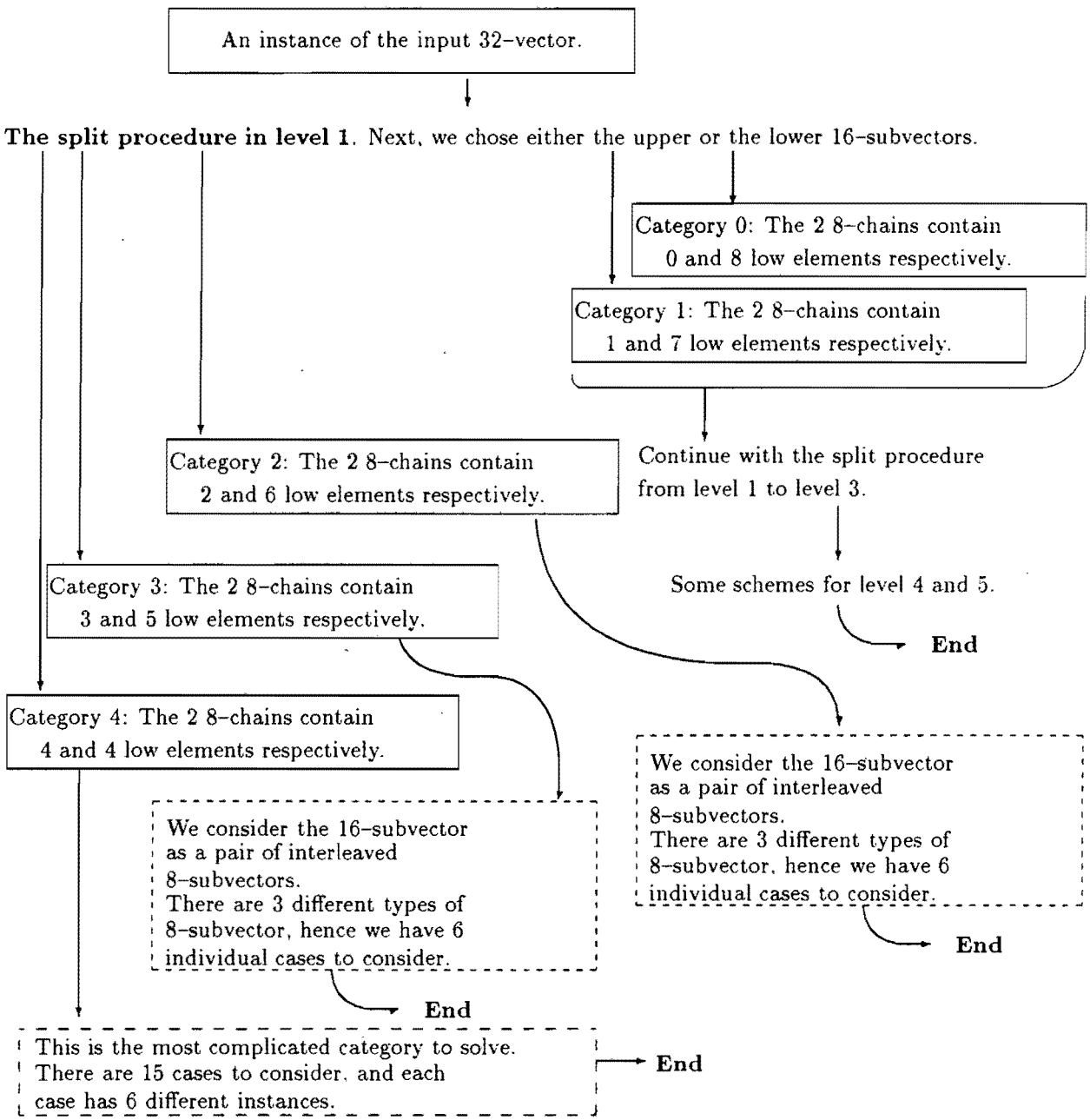

An instance of the input 32-vector.
The split procedure in level 1. Next, we chose either the upper or the lower 16-subvectors.
Category 0: The 2 8-chains contain 0 and 8 low elements respectively.
Category 1: The 2 8-chains contain 1 and 7 low elements respectively.
Category 2: The 2 8-chains contain 2 and 6 low elements respectively.
Continue with the split procedure from level 1 to level 3.
Some schemes for level 4 and 5.
End
Category 3: The 2 8-chains contain 3 and 5 low elements respectively.
Category 4: The 2 8-chains contain 4 and 4 low elements respectively.
We consider the 16-subvector as a pair of interleaved 8-subvectors. There are 3 different types of 8-subvector, hence we have 6 individual cases to consider.
End
We consider the 16-subvector as a pair of interleaved 8-subvectors. There are 3 different types of 8-subvector, hence we have 6 individual cases to consider.
End
This is the most complicated category to solve. There are 15 cases to consider, and each case has 6 different instances.
End

Figure 5.9:

way of searching for method for each possible combination will be too complicated to be extended to larger vectors. we are hoping that by clarifying some aspects of the rearrangeability problem. we will be able to apply some heuristic approach to solve the problem. Especially the idea of the matching subvectors and the iterative definition of the recursive balanced vectors might be useful to achieving this goal. Further work is still needed to solve this problem fully.

Chapter 6

Conclusion

Since the inception of computers in the late 1940's, there have been huge advances in computer technology. There has been unprecedented progresses in computer design and memory technology, where the development of VLSI circuits has played an integral part in this technological advancement. As circuit capacity increases and becomes more integrated, we are able to fit more transistors and more processors onto a single circuit. This means, in practical terms, that the chips are becoming more capable of handling tasks that demand extensive parallel processing. Furthermore, in the industrial environment, it has become the norm to expect a circuit to execute numerous tasks in the most efficient method, often performing the tasks in a parallel manner. To fully harvest the potential capability of such a chip, we will need to incorporate parallel execution more efficiently. Hence it is of vital importance that we understand the properties of a parallel network and are able to work on complex networks. In this work, we gave a lengthy exploration on two applications of parallel networks: sorting on higher dimensional networks and a routing problem on the Shuffle/Exchange network.

The first issue that we tackled was the problem of sorting in parallel networks. In order to achieve an efficient parallel sorting scheme on parallel networks, we began by introducing a comprehensive framework for modelling sorting algorithms on 2-dimensional grids. To this end we introduced a dual representations for sorting algorithms, namely the line representation and the grid representation. Following this framework, we were able to achieve two goals: we were able to standardize a number of existing algorithms under our proposed merge paradigm and we were able to construct efficient sorting algorithms that are also conducive to modularization into subsystems (packages). This merge paradigm enabled us to give a homogeneous evaluation of a host of merge and sorting algorithms on comparator networks. In the process, we were able to improve on the sorting techniques and merge schemes, so that we can utilize circuits that are capable of extensive parallel processing more efficiently.

Under our proposed merge paradigm, we found that it is feasible and advantageous to restrict the algorithms to use a single size comparator, the k-comparator. Thus all the sorting networks introduced in this work use one type of simple processor. The advantage of using only k-comparators is that it alleviates the problem of packaging the circuits. The problem of partitioning a system into physical sub-

systems is a cardinal task, when building large scale digital systems. It is desirable to have few different types of sub-systems, especially when the sub-systems are manufactured by mass production techniques, and when the sub-systems are not general-purpose ones. For a given physical size of a sub-system, it is usually preferable to reduce the total number of sub-systems in a system, rather than the total amount of logic in the system.

Taking into account all the considerations stated above, all the merge algorithms constructed under our merge scheme satisfy the three guidelines stated on page 21 explicitly: All the algorithms can be implemented on parallel networks that use a single sized k–comparator. This implies that we are able to implement the algorithm using one type of processor. Under our merge algorithm, the data and control flow are simple and regular. This is demonstrated when we implement the merge algorithms on various networks with local and regular interconnections to connect the processors - the grid networks, the k–comparator networks and the mesh interconnection networks. Furthermore, the sorting algorithms introduced in this work are merge algorithms, and to utilize the full potential of merge-type algorithms efficiently, we need to execute tasks through extensive parallel processing.

Following the merge paradigm, we constructed numerous merge algorithms that generalized various well-known merge algorithms, which are implemented using 2–comparator networks. In Chapter 3, we introduced the first class of merge algorithms that are constructed from the multi–way multi-merge procedure, where four different underlying grid merge algorithms were introduced. Two of those grid algorithms are from the family of q–way 2–merge algorithms. The q–way 2–merge bitonic algorithm and the q–way 2–merge modulo algorithm will need $2\lceil \log_n(N) \rceil - 3$ time cycles to merge 2 sorted $\frac{N}{2}$–vectors on a k–comparator network. The other two grid algorithms are from the family of q–way r–merge algorithms. The q–way r–merge r–tonic algorithm and the q–way r–merge modulo algorithm will need $2 \log_q k \log_k N - O(\log_q k)$ time cycles to merge r sorted $\frac{N}{r}$–vectors on a k–comparator network. We gave a summary of the time complexities of various sorting networks constructed from the multi-way multi-merge procedure in a table on page 73. Furthermore, if we restrict the size of the comparator to $(q-2)q \leq k \leq q^2$, then when $q \geq 16$ the q–way r–merge sorting network is more efficient than the q–way 2–merge sorting network. However, if $r < 16$, then the k–way 2–merge sorting network will need less depth than the q–way r–merge sorting network of the same size (input/output of size N). Also the q–way r–merge sorting network still uses fewer comparators than the k–way 2–merge sorting network (if $r \leq 4$).

To improve the time complexity of our algorithm, we then investigated the combined sorting algorithm. Under the assumption that N is a power of k, $r = q$ and $k = q^2$, the time complexity of an r–merge sorting network derived from the combined sorting procedure has $\lceil \frac{\log_2 k}{2} \rceil (\log_k N)^2 - O(\log_k N)$ time delay. This time complexity has the same leading term as the q–way 2–merge sorting network, however, it has a smaller the lower order terms. Furthermore, it requires fewer k–comparators.

The q–way r–merge r–tonic algorithm and the q–way r–merge modulo algorithm are very close to being optimal when implemented on planar interconnection networks. The time lower bound for sorting

on an $N^{\frac{2}{3}} \times N^{\frac{1}{3}}$ planar interconnection network is shown to be $2N^{\frac{2}{3}} + N^{\frac{1}{3}}$ [89], and the time upper bound of these two algorithms is shown to be $2.5N^{\frac{2}{3}} + O(N^{\frac{1}{3}})$, which is $0.5N^{\frac{2}{3}} + O(N^{\frac{1}{3}})$ more than the time lower bound.

Another advantage of using our merge paradigm is the fact that it enables one to focus more on improving the sorting technique instead of the implementation of the algorithm. By the fact that we are able to understand the sorting technique better, we are in a better position to find more suitable models to implement the sorting algorithm. Hence we are sometimes able to improve on the time complexity of an algorithm. For example from Theorem 3.9.1, the modified algorithm that follows our merge paradigm improves the time complexity of the original column sort algorithm introduced in Leighton [56].

In Chapter 4, we introduce the second class of merge algorithms that are constructed from the G-merge procedure, where four different underlying grid merge algorithms are introduced. The cochain merge algorithm, the bichain merge algorithm, the alt-q-bichain merge algorithm and the modified bichain merge algorithm, where they belong to the family of q-way r-merge algorithms and they have good packaging properties. These algorithms are implemented following the scheme of our merge paradigm. However, in these cases the algorithms can be implemented without rewiring. The approach that we took is as follows: we first explored different ways of methodically partitioning a $p \times q$ grid (where p is a power of q) into subvectors of equal size, where the size of a vector is the number of its components. We then implemented the sorting algorithm using the prescribed decomposition chains. In this way the sorting algorithms generated can be better partitioned into physical sub-systems. This is desirable when building large scale digital systems, since in practice the networks will be packaged. The merge algorithms derived from the G-merge procedure have the same time delay as the q-way r-merge algorithms derived from the multi-way multi-merge algorithm. However, they are not as efficient as the q-way r-merge algorithms derived from the multi-way multi-merge procedure, when implemented on planar interconnection networks.

The fact that these latter algorithms are more readily modularized into sub-systems (since rewiring is not needed) implies the algorithms can be more readily converted to sort on higher dimensional interconnection networks. We demonstrated how some of the algorithms could sort 3-D interconnection networks with only minor modifications. Similar ideas can also apply to the cochain merge algorithm, where we sorted the diagonal planes of the 3-D interconnection networks.

As we demonstrated in the exploratory section of Chapter 4, there are numerous other possibilities that we have not investigated, where new types of transformer or new types of m-g-chains can be constructed and used in our class of merge algorithms.

One aspect that has not been investigated fully is how to construct a k-comparator. As we have demonstrated in Chapter 3, the time complexity of a k-comparator network derived from our sorting algorithm depends on the size of the k-comparators used. Since the parameter k is bounded by the parameters q and r of the underlying merge algorithms used, the time complexity of various k-comparator

networks can be effected by varying the values of the parameters q and r used. Furthermore, how the k–comparator is constructed will also effect the time complexity of the algorithm. As we have shown in Chapter 3, if we are to use 2–comparators to implement the required k–comparators, then the time complexity of our algorithms will need to be adjusted accordingly. Suppose we need the k–comparator to function as a k–sorter, where we choose to use the Batcher's bitonic 2–comparator sorting network to implement such a k–sorter. Then the k–comparator constructed will induce a delay of $\frac{1}{2}(\log_2(k))(\log_2(k)+1)$ multiplied by the delay of a single 2–comparator (since the bitonic 2–comparator sort of a k–vector used $\frac{1}{2}(\log_2(k))(\log_2(k)+1)$ layers of 2–comparators). However, it is not always the case that one needs to use a k–sorter to emulate a k–comparator. For example, as we have demonstrated the k–comparator network derived from the q–way bitonic merge algorithm or the q–way 2–merge modulo algorithm will need k–comparators to perform as k–merge operations not as k–sorters. Suppose we use the bitonic merger of size k to implement the required k–comparators. Then the k–comparators used in this algorithm will only be delayed by a factor of $\log_2(k)$ (since the bitonic merge uses $\log_2(k)$ layers of 2–comparators to merge). However, this discussion will become irrelevant, if we are able to construct efficient k–comparators without using the 2–comparators. For this case our algorithms will outperform most of the sorting algorithms.

In Chapter 5, we discussed solving the routing problem on the double butterfly network (the Shuffle Exchange network). We constructed a control algorithm that routes every possible combination of signals from the input terminals to the output terminals of a double butterfly network for $N \leq 32$. Furthermore, our approach deviated drastically from the existing approaches presented in the literature, which means we are able to bring many new insights into this problem. Unfortunately, we where not able to generalize this solution for $N > 32$. However, we think that enough new results have been derived from our method to demonstrate that our results can be useful for finding solutions the general cases where $N > 32$.

The only drawback of our algorithm is that we need to arrange the vectors into a set of very specific patterns, and the algorithm does not recognize other possible solutions that are not in this solution set. The restriction of finding solutions from one particular set of patterns may cause the problem to become more complicated as N gets larger, since we lose the freedom to look for solutions from other possible solution sets. On the other hand, the reason we chose to be restricted to the use of this pattern only, is that we want the algorithm to work on a smaller domain space, so that the algorithm is more precise and easy to comprehend. One can incorporate other possible patterns that can also be used to derive recursively balanced vectors into the algorithm. However, by increasing the number of case patterns that an algorithm needs to search through may cause it to become complicated and its time complexity may increase. One of the possible approaches is to use heuristic methods to solve the problem. The idea of a heuristic approach seems suitable, since we can assign a weight function to some preferred patterns, and then by examining the pattern of a given vector, we assign a weight function to each feasible pattern. In such a way, it is possible to reduce the time to search for possible solutions, from a much larger domain space.

Bibliography

[1] D. P. Agrawal (1983), Graph Theoretical Analysis and Design of Multistage Interconnection Networks. *IEEE Transactions on Computers*, **C-32**, 637–648.

[2] A. Agrawal (1991), Limits on Interconnection Network Performance. *IEEE Transactions Parallel and Distributed Systems*, **2**, 398–412.

[3] M. Ajtai, J. Komlos, and E. Szemeredi (1983), An $O(n \log n)$ sorting network. *Combinatorica*, **3**, 1–19.

[4] M. Ajtai, J. Komlos and E. Szemeredi (1992), Halvers and Expanders. *STOC 1992*, 686–689.

[5] S. B. Akers and B. Krishnamurthy (1989), A Group-Theoretic Model for Symmetric Interconnection Networks.

[6] A. Andersson, T. Hagerup, S. Nilsson and R. Raman (1995), Sorting in linear time. *27th Annual ACM Syposium on Thorey of Computing*, 427–436.

[7] A. Bar-Noy and D. Peleg (1991), Square Meshes are not always Optimal. *IEEE Transactions on Computers*, **40**, 196–204.

[8] K.E. Batcher (1968), Sorting Networks and their application. *Proc. AFIPS 1968 Spring Joint Computer Conf*, **32**, 307–314.

[9] R. Becker, D. Nassimi and Y. Perl (1998), The New Class of g–chain Periodic Sorters. *Journal of Parallel and Distributed Computing*, **54**, 206–222.

[10] R. Becker and A. Litman (1999), Multicomparator Sorting Networks with Better Packaging. In preparation.

[11] R. Becker, D. Nassimi and Y. Perl, The generalized class of g-chain periodic sorting networks. Research Report, Department of Mathematics and Applied Mathematics, University of Cape Town.

[12] R. Beigel and J. Gill (1990), Sorting n Objects With a k–Sorter. *IEEE Transactions on Computers*, **39**, 714–716.

[13] V. E. Benes (1964), Permutation Groups, Complexes and Rearrangeable Connecting Networks. *The Bell System Technical Journal*, **43**, 1619–1640.

[14] V. E. Benes (1964), Optimal Rearrangeable Multistage Connecting Networks. *The Bell System Technical Journal*, **43**, 1641–1656.

[15] V. E. Benes (1975), Proving the Rearrangeability of Connecting Networks by group Calculations. *The Bell System Technical Journal*, **54**, 421–434.

[16] G. Berman and K.D. Frayer (1972), Introduction to Combinatorics. Academic Press.

[17] J.C. Bermond, J.M. Fourneau and A. Jean-Marie (1987/88), Equivalence of Multistage Interconnection Networks. *Information Processing Letters*, **26**, 45–50.

[18] S. N. Bhatt, F. R. K. Chung, J-W. Hong, F. T. Leighton, B. Obrenic and E. J. Schwabe (1996), Optimal Emulations by Butterfly-Like Networks. *Journal of the ACM*, **43**, 293–330.

[19] G. Bilardi and F.P. Preparata (1984), An Architecture for Bitonic Sorting with Optimal VLSI Performance. *IEEE Transactions on Computers*, **C-33**, 646–651.

[20] G. Bilardi and F.P. Preparata (1985), A Minimum Area VLSI Network for $O(\log n)$ Time Sorting. *IEEE Transactions on Computers*, **C-34**, 336–343.

[21] G. Bilardi (1989), Merging and Sorting Networks with the Topology of the Omega Network. *IEEE Transactions on Computers*, **38**, 1396–1403.

[22] G. Bilardi and A. Nicolau (1989), Adaptive Bitonic Sorting: An Optimal Parallel Algorithm For Shared-Memory Machines. *SIAM Journal Comput.*, **18**, 216–228.

[23] G. Broomell and J. R. Heath (1995), Classification Categories and Historical Development of Circuit Switching Topologies. *Computing Surveys*, **15**, 95–133.

[24] H. Cam and J. A. B. Fortes (1990), Rearrangeability of Shuffle-Exchange Networks. *In 3rd Symposium on the Frountiers of Massively Parallel Computation*, 303–314.

[25] G. E. Carlsson, J.E. Cruthirds, H. B. Sexton and C. G. Wright (1985), Interconnection Networks Based on a Generalization of Cube-Connected Cycles. *IEEE Transactions on Computers*, **C-34**, 769–772.

[26] P-Y. Chen, D. H. Lawrie, P-C. Yew and D. A. Padua (1981), Interconnection Networks Using Shuffles. *IEEE Computer*, **14**, 55–64.

[27] Y. Chiang (1999), The Double Butterfly is Rearrangeable for $N = 32$. (In preparation).

[28] R. J. Cole (1988), An Optimally Efficient Selection Algorithm. *Information Processing Letters*, **26**, 295–299.

[29] R. J. Cole (1988), Parallel Merge Sort. *SIAM Journal Comput.*, **17**, 770–785.

[30] R. Cole and A. Siegel (1988), Optimal VLSI Circuits for Sorting. *JACM*, **35**, 777–809.

[31] T. H. Corman, C. E. Leiserson and R. L. Rivest (1990), Introduction to Algorithms. The MIT Press. Cambridge, Massachusettes.

[32] R. Cypher and C. G. Plaxton (1993), Deterministic Sorting in Nearly Logarithmic Time on the Hypercube and Related Computers. *Journal of Computer and System Sciences*, **47**, 501–548.

[33] N. Das and J. Dattagupta (1996), Two-Pass Rearrangeability in Faulty Benes Networks. *Journal of Parallel and Distributed Computing*, **35**, 191–198.

[34] M. Dowd, Y. Perl, L. Rudolph and M. Saks (1989), The Periodic Balanced Sorting Network. *JACM*. **36**. 738–757.

[35] V. Estivill-Costro and D. Wood (1992), A Survey of Adoptive Sorting Algorithms. *ACM Computing Surveys*, **24**, 441–476.

[36] S. Even and R. Kupershtok (1994), The Double Baseline is Rearrangeable. *Research Report of Israel Institute of Technology*, 1–10.

[37] T-Y. Feng (1981), A Survey of Interconnection Networks. *IEEE Computer*, **14**, 12–27.

[38] T-Y. Feng and S-W. Seo (1994), A new Routing Algorithm for a Class of Rearrangeable Networks. *IEEE Transactions on Computers*, **C-43**, 1270–1280.

[39] B. M. Gocal and K. E. Batcher (1996), Bitonic Sorting on Benes Networks. *IEEE Proceeding of IPPS'96*. 749–753.

[40] T. Hagerup and C. Rüb (1989), Optimal Merging and Sorting on the EREW PRAM. *Information Processing Letters*, **33**, 181–185.

[41] A. R. Karlin, G. Nelson and H. Tamaki (1994), On the Fault Tolerance of the Butterfly. *26th Annual ACM Syposium on Thorey of Computing*, 125–133.

[42] T. Hayashi, K. Nakano and S. Olariu (1998),Work-Time Optimal k–merge Algortihms on the PRAM. *IEEE Transactions on Parallel and Distributed Systems*, **9**, 275–282.

[43] Y. Hen and Y. Igarashi (1988), Time Lower Bounds for Parallel Sorting on A Mesh-Connected Processor Array. *Proceeding 3rd Aegean Workshop on Computing: VLSI Algorithms and Architectures*, Lecture Notes in Comput. Sci., 319, Springer, Berlin, 434–443.

[44] J. Ja'Ja' and R. M. Owens (1984), VLSI Sorting with Reduced Hardware. *IEEE Transactions on Computers*, **C-33**, 668–671.

[45] J-W Jang and V. K. Prasanna (1995), An Optimal Sorting Algorithm on Reconfigurable Mesh. *Journal of Parallel and Distributed Computing*, **25**, 31–41.

[46] N. Kahale, T. Leighton, Y. Ma, C. G. Plaxton, T. Suel and E. Szemeredi (1995). Lower Bounds for Sorting Networks. *27th Annual ACM Syposium on Thorey of Computing*, 437–446.

[47] M. Kaufmann and J. F. Sibegn (1997), Rondomized Multipacket Routing and Sorting on Meshes. *Algorithica*, **17**, 224–244.

[48] W. H. Kautz, K. N. Levitt and A. Waksman (1968), Cellular Interconnection Arrays. *IEEE Transactions on Computers*, **C-17**, 443–451.

[49] D. E. Knuth (1973), The Art of Computer Programming, Vol. 3. Addison Wesley.

[50] M. Kumar and D. S. Hirschberg (1983), An Efficient Implementation of Batcher's Odd-Even Merge Algorithm and Its Application in Parallel Sorting Schemes. *IEEE Transactions on Computers*, **C-32**, 254–264.

[51] V. P. Kumar and S. M. Reddy (1987), Augmented Shuffle-Excahnge Multistage Interconnection Networks. *IEEE Computer*, **20**, 30–40.

[52] M. Kunde (1987), Lower Bounds for Sorting on Mesh-Connected Architectures. *Acta Informatica*, **24**, 121–130.

[53] H-W. Lang, M. Schimmler, H. Schmeck and H. Schroder (1985), Systolic Sorting on a Mesh-Connected Network. *IEEE Transactions on Computers*, **C-34**, 652–658.

[54] D. H. Lawrie (1975), Access and Alignment of Data in a Array Processor. *IEEE Transactions on Computers*, **C-30**, 324–332.

[55] K. Y. Lee (1985), On the Rearrangeability of $2(\log_2 N) - 1$ Stage Permutation Networks. *IEEE Transactions on Computers*, **C-34**, 118–131.

[56] F. T. Leighton (1985), Tight Bounds on the Complexity of Parallel Sorting. *IEEE Transactions on Computers*, **C-34**, 344–354.

[57] F. M. Leighton and M. Maggs (1992), Fast Algorithms for Routing Around Faults in Multibutterflies and Randomly-Wired Splitter Networks. *IEEE Transactions on Computers*, **41**, 578–587.

[58] F. T. Leighton (1992), Introduction to Parallel Algorithms and Architectures: Arrays, Trees and Hypercubes. Morgan-Kaufmann, San Mateo, Ca.

[59] T. Leighton, Y. Ma and C. G. Plaxton (1997), Breaking the $\Theta(n \log^2 n)$ Barrier for Sorting with Faults. *Journal of Computer and System Sciences*, **54**, 265–304.

[60] T. Leighton and C. G. Plaxton (1998), Hypercubic Sorting Networks. *SIAM Journal Comput.*, **27**, 1–47.

[61] K. Li and K-H. Cheng (1991), A Two-Dimensional Buddy System for Dynamic Resource Allocation in A Partitionable Mesh Connected System. *Journal of Parallel and Distributed Computing*, **12**, 79–83.

[62] K. J. Liszka and K. E. Batcher (1992), A Modulo Merge Sorting Network. *Frontiers*.

[63] A. Litman, Parceling the butterfly and the Batcher sorting Network. Preprint, CS Department, Technion, Haifa, Israel.

[64] T. Nakatani, S-T Huang, B. W. Arden and S. K. Tripathi (1989), k–Way Bitonic Sort. *IEEE Transactions on Computers*, **38**, 283–288.

[65] D. Nassimi, R. Becker, and Y. Perl (1994), The generalized class of g-chain periodic sorting networks. *Proceedings 8th International Parallel Processing Symposium (IEEE), Cancun, Mexico*, April, 424–432.

[66] D. Nassimi and S. Sahni (1979), Bitonic Sort on a Mesh-Connected Parallel Computer. *IEEE Transactions on Computers*, **C-27**, 2–7.

[67] M. Nigam and S. Sahni (1994), Sorting n Numbers on $n \times n$ Reconfigurable Meshes with Buses. *Journal of Parallel and Distributed Computing*, **23**, 37–48.

[68] S. Olariu and J. Schwing (1996), A Novel Deterministic Sampling Scheme with Applications to Broadcast-Efficient Sorting on the Reconfigurable Mesh. *Journal of Parallel and Distributed Computing*, **32**, 215–222.

[69] S. Olariu, M. C. Pinotti and S. Q. Zheng (1999), How to Sort N items Using a Sorting Network of Fixed I/O Size. *IEEE Transactions on Parallel and Distributed Systems*, **10**, 487–499.

[70] D. C. Opferman and N. T. Tsao-Wu (1971), On a class of Rearrangeable Switching Networks Part 1: Control Algorithm. *The Bell System Technical Journal*, **50**, 1579–1600.

[71] D. C. Opferman and N. T. Tsao-Wu (1971), On a class of Rearrangeable Switching Networks Part 2: Enumeration Studies and Fault Diagnosis. *The Bell System Technical Journal*, **50**, 1601–1618.

[72] A. Y. Oruc and M. Y. Oruc (1987), Programming Cellular Permutation Networks Through Decomposition of Symmetric Groups. *IEEE Transactions on Computers*, **C-36**, 802–809.

[73] B. Parhami and C. Y. Hung (1995), Robust Shearsort on Incomplete Bypass Meshes. *Proceedings of International Conference on Parallel Processing*, 304–311.

[74] D. S. Parker, Jr (1980), Notes on Shuffle/Exchange-Type Switching Networks. *IEEE Transactions on Computers*, **29**, 213–222.

[75] B. Parker and I. Parberry (1989/90), Constructing Sorting Networks from k–sorters. *Information Processing Letters* , **33**, 157–162.

[76] J. H. Patel (1981), Perormance of Processor-Memory Interconnections for Multiprocessors. *IEEE Transactions on Computers*, **C-30**, 771–780.

[77] Y. Perl (1989), Better Understanding of Batcher's Merging Networks. *Discrete Applied Mathematics*, **25**, 257–271.

[78] C. G. Plaxton (1992), A Hypercubic Sorting Network with Nearly Logarithmic Depth. *Proceedings of the* 24^{th} *annual ACM Symposium on the Theory of Computing*, May 4–6, 405–416.

[79] F. P. Preparata (1978), New Parallel-Sorting Schemes. *IEEE Transactions on Computers*, **C-27**, 669–673.

[80] C. S. Raghavendra and R. V. Boppana (1991), On Self-Routing in Benes and Shuffle-Exchange Networks. *IEEE Transactions on Communications*, **40**, 1057–1064.

[81] C. S. Raghavendra and A. Varma (1987), Rearrangeability of the Five-Stage Shuffle-Exchange Network for $N = 8$. *IEEE Transactions on Communications*, **35**, 808–812.

[82] A. Raghunathan and H. Saran (1991), Is the Shuffle-Exchang Better than the Butterfly? *ISA '91 Algorithms*, Lecture Notes in Comput. Sci., 557, Springer, Berlin, 32–41.

[83] H. R. Ramanujam (1973), Decomposition of Permutation Networks. *IEEE Transactions on Computers*, **C-22**, 639–643.

[84] L. Rudolph (1985), A Robust Sorting Network. *IEEE Transactions on Computers*, **C-34**, 326–335.

[85] Y. Saad and M. H. Schultz (1988), Topological Properties of Hypercubes. *IEEE Transactions on Computers*, **37**, 867–872.

[86] I. D. Scherson and S. Sen (1989), Parallel Sorting in Two-Dimensional VLSI Models of Computation. *IEEE Transactions on Computers*, **C-38**, 238–249.

[87] I.D. Scherson (1991), Orthogonal Graphs for The Construction of a Class of Interconnection Networks. *IEEE Transactions on Parallel and Distributed Systems*, **2**, 3–19.

[88] I. D. Scherson and A. S. Youssef (1994), Interconnection Networks for High-Performance Parallel Computers. IEEE Computer Society Press, Los Alamitos, California.

[89] C. P. Schnorr and A. Shamir (1986), An Optimal Sorting Algorithm for Mesh Connected Computers. *Proc.* 18^{th} *ACM Symposium on Theory of Computing*, 255–263.

[90] M. Schimmler and C. Starke (1988), VLSI Algorithms and Architectures. *Lecture Notes in Computer Sciences 319, Springer, New York-Berlin*, 444–455.

[91] J. F. Sibeyn (1998), Row-Major Sorting on Meshes. *SIAM Journal Comput.*, **28**. 847–863.

[92] D. Steinberg (1983), Invariant Properties of the Shuffle-Exchange and a Simplified Cost-Effective Version of the Omega Network. *IEEE Transactions on Communications*, **C-32**, 444–450.

[93] H. S. Stone (1971), Parallel Processing with the Perfect Shuffle. *IEEE Transactions on Computers*, **C-20**, 153–161.

[94] T. H. Szymanski and V. C. Hamacher (1987), On the Permutation Capability of Multistage Interconnection Networks. *IEEE Transactions on Computers*, **C-36**, 810–822.

[95] C. D. Thompson and H. T. Kung (1977), Sorting on a Mesh-Connected Parallel Computer. *Communications of ACM*, **20**, 263–271.

[96] C. D. Thompson (1983), The VLSI Complexity of Sorting. *IEEE Transactions on Computers*, **C-32**, 1171–1184.

[97] J. Ullman (1983), Computational Aspects of VLSI, Rockville, Computer Science Press.

[98] A. Varma and C. S. Raghavendra (1987), Rearrangeability of Multistage Shuffle/Exchange Networks. *In Proc. 14th Annu. Symp. Comput. Architect.*, 154–162.

[99] A. Varma and C. S. Raghavendra (1989), Fault-Tolerant Routing in Multistage Interconnection Networks. *IEEE Transactions on Computers*, **38**, 385–393.

[100] A. Varma and C. S. Raghavendra (1994), Interconnection Networks for Multiprocessors and MultiComputers: Theory and Practice. IEEE Computer Society Press, Los Alamitos, California.

[101] A. Waksman (1968), A permutation Network. *Journal of ACM*, **15**, 159–163.

[102] Z. Wen (1996), Multiway Merging in Parallel. *IEEE Transactions on Parallel and Distributed Systems*, **7**, 11–17.

[103] C-L Wu and T-Y Feng (1980), The Reverse-Exchange Interconnection Network. *IEEE Transactions on Computers*, **C-29**, 801–811.

[104] C-L Wu and T-Y Feng (1980), On a class of Multistage Interconnection Networks. *IEEE Transactions on Computers*, **C-29**, 694–702.

[105] C-L Wu and T-Y Feng (1981), The Universality of the Shuffle-Exchange Network. *IEEE Transactions on Computers*, **C-30**. 324–332.

[106] S. Yalamanchill and J. K. Aggaewal (1987), A Characterization and Analysis of Parallel Processor Interconnection Networks. *IEEE Transactions on Computers*, **C-36**, 680–691.

[107] A. C. Yao and F. F. Yao (1985), Fault Tolerant Sorting Networks. *SIAM J. Comp.* **18**, 120–128.

[108] Y-M. Yeh and T-Y. Feng (1992), On a Class of Rearrangeable Networks. *IEEE Transactions on Computers*, **41**, 1361–1379.

[109] I. Yen, F. Bastani and E. Leiss (1991), An Inherently Fault Tolerant Sorting Algorithm. *Proceedings of International Conference on Parallel Processing*. 37–42.

[110] H. Yoon, K. Y. Lee and M. T. Liu (1990), Performance Analysis of Multibuffered Packet-Switching Networks in Multiprocessor Systems. *IEEE Transactions on Computers*, **39**, 319–327.