Routing and Sorting on Fixed Topologies

by

Torsten Suel, Dipl.-Inform., M.S.C.S

Dissertation

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

Doctor of Philosophy

The University of Texas at Austin

December, 1994

Routing and Sorting on Fixed Topologies

Approved by
Dissertation Committee:

Acknowledgments

First, I would like to thank my advisor Greg Plaxton. Greg showed a lot of patience with me during the first years of my study, and was always available for discussions and encouragement throughout my stay at UT. Thanks also to the other members of my dissertation committee, Tom Leighton, Jay Misra, Vijaya Ramachandran, Robert van de Geijn, and Martin Wong, for their time and efforts.

Over the years, my work has benefited from discussions with a number of people, including Nelson Amaral, Tsan-sheng Hsu, Phil MacKenzie, Rajmohan Rajaraman, Jop Sibeyn, and Rolf Wanka. Thanks also to all friends, officemates, and colleagues that made my stay in Austin a great experience.

Finally, and most importantly, I would like to thank my parents, Anton and Sigrid Suel, without whose constant love and support nothing of this would have been possible.

TORSTEN SUEL

The University of Texas at Austin
December 1994

Routing and Sorting on Fixed Topologies

Publication No. ______________________

Torsten Suel, Ph.D.
The University of Texas at Austin, 1994

Supervisor: Charles Gregory Plaxton

This thesis studies the problems of packet routing and sorting on parallel models of computation that are based on a fixed, bounded-degree topology. It establishes lower bounds for several classes of sorting networks and algorithms, and describes techniques and algorithms for packet routing and sorting on mesh-connected and related networks.

A lower bound of $\Omega(\lg n \lg\lg n / \lg\lg\lg n)$ is established for the depth of shuffle-unshuffle sorting networks, a class of sorting networks that maps efficiently to the hypercube and its bounded-degree variants. A stronger lower bound of $\Omega(\lg^2 n / \lg\lg n)$ is shown for a subclass of the shuffle-unshuffle sorting networks whose structure corresponds to the class of ascend and descend algorithms on the hypercube. These lower bounds also extend to restricted classes of non-oblivious sorting algorithms on hypercubic networks. A lower bound of $\Omega(n \lg^2 n / (\lg\lg n)^2)$ is shown for the size of Shellsort sorting networks, and for the running time of non-oblivious Shellsort algorithms. The lower bound establishes a trade-off between the running time of a Shellsort algorithm and the length of the underlying increment sequence.

For the problems of permutation routing and sorting on meshes and related networks, a set of techniques is proposed that can be used to convert many randomized algorithms into deterministic algorithms with matching running time and queue size. Applications of these techniques lead to a deterministic algorithm for sorting on the two-dimensional mesh that achieves a running time of $2n + o(n)$, and a fairly simple deterministic algorithm for routing with a running time of $2n + o(n)$ and very small queue size. Some other applications of the techniques are also described. Finally, the thesis gives algorithms and lower bounds for routing and sorting on multi-dimensional meshes and meshes with bus connections.

Contents

Chapter 1

Introduction

Among the many theoretical models of parallel computation that have been proposed, the *Parallel Random Access Machine* (PRAM) is undoubtedly the most highly studied. In the PRAM model, a set of sequential processors communicates via a shared memory. In a single step of the computation, each processor can perform a bounded amount of internal computation, and access an arbitrary location in the shared memory. In particular, this also means that any pair of processors can communicate in a single step. While this assumption of a shared memory allows for a succinct statement of many parallel algorithms, it also makes the PRAM model somewhat unrealistic.

In more realistic models of parallel computation, the memory is distributed among the processing devices, and each processing device can only communicate with a small, fixed set of neighbors in a single step, while several communication steps are necessary in order to send a message between processing devices that are not directly connected to each other by a communication link. We refer to such a model of computation as having a fixed topology.

As a result of this restriction to communication among neighboring processing devices, the need arises for efficient algorithms that implement a variety of communication patterns between processing devices not directly connected to each other. Such algorithms are commonly referred to as *routing algorithms*, and they have been studied extensively in the last two decades. Another problem that is closely related to the routing problem, but also important in its own right, is the problem of parallel sorting. Both routing and sorting are important subroutines in many parallel algorithms, and various models of specialized hardware devices for these problems have been proposed.

1.1 Model of Computation

In this thesis, we consider the problems of routing and sorting on parallel models of computation that are based on a fixed topology. We focus our attention on the following two classes of machines, which we refer to as *fixed-connection networks*, and *circuits*. A *fixed-connection network* consists of a collection of sequential processors connected by a sparse system of communication links. The processors operate in a synchronous fashion, and communicate by sending messages over the communication links. In a single step, a processor can read a constant number of the messages that were sent to it in the previous step, perform some fixed amount of internal computation, and send a constant number of messages across its communication links to neighboring processors.

By a *circuit*, we understand a collection of specialized hardware devices arranged in the form of a directed acyclic graph. Each hardware device can perform a fixed elementary operation on a set of input values supplied by its incoming edges, and output the results of this operation on its outgoing edges in the following step. The computation of a circuit is started by supplying a set of input values to a set of special devices with in-degree zero called *input nodes*. The results of the computation appear at the *output nodes*, which are special devices with out-degree zero. Examples of *circuits* are sorting networks built from comparators, or Boolean circuits composed of *AND*, *OR*, and *NOT* gates.

The main differences between fixed-connection networks and circuits are as follows. The fixed-connection network is usually considered as a model for a general-purpose parallel computer, where each processor in the network is equivalent in power to a sequential *Random Access Machine*. The system of communication links should be chosen in such a way that the resulting topology of the network is simple and allows for an elegant and efficient solution of a variety of algorithmic problems. In contrast, circuits are designed as special-purpose hardware for one particularly important application, such as sorting, routing, or the computation of a fixed arithmetic or Boolean expression. The devices used in the circuit are usually extremely simple, and perform a fixed operation on the input values. The topology of the circuit, on the other hand, may be complicated and irregular, as it does not have to support any other applications. (Of course, due to layout constraints or other considerations, it may still be advantageous to have a simple topology.)

Another difference between the two models is in the possible initial distribution of the input values and the structure of the computation. Under the fixed-connection network model, every processor can initially contain one or even a large number of input values, while in a circuit every input node can have at most one input. On the other hand, the circuit model is especially suitable for pipelined computations. In particular, if the circuit is *leveled*, then a new set of input values can enter the network in every step. Here, we say that a circuit is *leveled*, if and only if there exists a labeling of the nodes with integer values such that all input nodes are labeled with 0, all output nodes are labeled with some value l, and every edge goes from a node with label i to a node with label $i+1$, for some i.

We remark at this point that the distinction between fixed-connection networks and circuits is not always as clear as may be suggested by the above presentation. The reader should think about these two classes as the two extremes in a spectrum of possibilities, rather than as two unrelated and completely disjoint classes. In fact, there are numerous relationships between the two classes. For example, a number of sorting and switching circuits, such as the bitonic sorter or the Beneš permutation routing network, can be efficiently implemented on many important fixed-connection networks. As another example, a switching circuit composed of 2-input 2-output switches usually requires some additional processor power and memory space in its nodes, for example, to control the sequence of switch settings, or to buffer packets that are delayed. Also, many systolic algorithms are suitable for implementation on fixed-connection networks as well as circuits of specialized hardware devices.

In the following, the number of processors of a fixed-connection network is denoted by N. We use the symbol $\mathbf{N}$ to denote the set of natural numbers, and $[n]$ to denote the set $\{0, \ldots, n-1\}$. Given $x, y \in \mathbf{N}^k$, we define the *Hamming distance* between x and y as $\mathrm{ham}(x, y) \stackrel{\mathrm{def}}{=} \sum_{i=1}^{k} |x_i - y_i|$, where x_i and y_i denote the ith components of x and y, respectively. We also define $\mathrm{ham}_n(x, y) \stackrel{\mathrm{def}}{=} \sum_{i=1}^{k} \min\{|x_i - y_i|, n - |x_i - y_i|\}$.

In the next section, we describe the classes of mesh-connected and hypercubic fixed-connection networks, and define the problems of routing and sorting on these networks. Section 1.3 describes sorting circuits based on comparator and switching elements. Finally, Section 1.4 contains an overview of the main contributions of this thesis. Detailed descriptions of previous results can be found in the introductory sections of the subsequent chapters.

1.2 Fixed-Connection Networks

A fixed-connection network can be described by an undirected graph $G = (V, E)$, where each vertex v_i corresponds to a processor p_i, and each edge (v_i, v_j) corresponds to a communication link between processors p_i and p_j. Unless explicitely stated otherwise, we assume that all communication links are bidirectional, and that a bounded amount of information can be transmitted in either direction in a single step. Note that this definition does not take the possible existence of buses in the network into account, as these are usually connected to more than two processors. Such networks with buses are discussed in Chapter 6.

A variety of different fixed-connection networks have been proposed in the literature; for further references and an overview of the most important classes of networks, we refer the reader to Leighton's text [66]. In this thesis, we restrict our attention to the families of mesh-connected and hypercubic networks, which are probably the most important and most extensively investigated classes of fixed-connection networks. They have a simple structure and admit elegant and efficient implementations of a variety of parallel algorithms.

Aside from their practical importance, we believe that the mesh-connected and hypercubic networks also deserve further attention due to the interesting relationship between the two classes. On the one hand, the mesh-connected networks and the hypercube are closely related in structure. On the other hand, the two-dimensional mesh and the hypercubic networks are on opposite ends of the spectrum with respect to diameter and layout area, and hence many of the techniques developed for one of the two classes are not suitable for the other. This raises the question of whether we can obtain algorithms that achieve good performance on both the two-dimensional mesh and the hypercubic networks, as well as on the entire spectrum of multi-dimensional meshes in between.

1.2.1 Mesh-Connected Networks

The d-dimensional mesh-connected network of side length n (or d-dimensional mesh for short) is the network $M_{n,d} = (V, E)$ with $V = [n]^d$ and $E = \{(x, y) \in V^2 \mid \mathrm{ham}\,(x, y) = 1\}$. By adding wrap-around edges to this network, we obtain the d-dimensional torus of side length n, formally defined as $T_{n,d} = (V, E)$ with $V = [n]^d$ and $E = \{(x, y) \in V^2 \mid \mathrm{ham}_n(x, y) = 1\}$. Other closely related networks are meshes and tori with diagonal edges, trigonal and hexagonal meshes, and the various classes of meshes with buses discussed in Chapter 6.

The one-dimensional mesh and torus networks are also often referred to as the *linear array* and *ring*, respectively. Many problems related to routing and sorting on these networks are already fairly well understood, and hence we restrict our attention to networks of dimension at least 2. We only remark at this point that a good understanding of these one-dimensional networks is very important in the study of routing and sorting algorithms for networks of dimension 2 and higher, as many of these algorithms use routing and sorting on linear arrays and rings as subroutines.

Of particular practical importance are the cases of the two-dimensional mesh and torus. A number of parallel machines have been designed and built based on these topologies (e.g., see [3]), and numerous algorithmic problems have been studied both in theory and practice. In contrast, the meshes and tori of dimension $d \geq 3$, also called *multi-dimensional meshes*, have received somewhat less attention, and many problems related to routing and sorting on these networks remain open.

1.2.2 Hypercubic Networks

The second family of fixed-connection networks that we consider in this thesis are the *hypercubic networks*. For $d > 0$, the *d-dimensional hypercube* is defined as $H_d = (V, E)$ with $V = \{0,1\}^d$ and $E = \{(x,y) \in V^2 \mid \mathrm{ham}\,(x,y) = 1\}$. Thus, a d-dimensional hypercube has $N = 2^d$ processors that can be labeled with the 2^d bit strings of length d in such a way that two processors are connected by a communication link if and only if their labels differ in exactly one bit position. Note that the d-dimensional hypercube is nothing more than a d-dimensional mesh of side length 2. Thus, we can regard the two-dimensional mesh and the hypercube as being on opposite ends of the spectrum of mesh-connected networks.

One disadvantage of the hypercube is its non-constant degree. In a hypercube of dimension d, each of the $N = 2^d$ processors is connected to d other processors; for larger N this quickly becomes impractical. However, many important algorithms for the hypercube have a very regular structure that does not require the full connectivity afforded by the network. To formalize this claim, we say that a processor communicates across dimension i if it sends a message to the neighbor whose label differs in the ith bit position from its own label. An algorithm for the d-dimensional hypercube is called *normal* if we can assign a label $l(i) \in \mathbf{N}$ to the ith step of its computation, for all $i \in \mathbf{N}$, such that $|l(i+1) - l(i)| \leq 1$ and every processor only communicates across dimension $l(i) \bmod d$ in the ith step of the computation. Two natural subclasses of the normal algorithms are obtained by imposing the conditions $l(i+1) = l(i) + 1$ and $l(i+1) = l(i) - 1$, respectively, on the labeling of the computation steps in the above definition; the resulting classes of algorithms are known as the *ascend* and *descend* classes, respectively

Examples of important normal algorithms are the Fast Fourier Transform, prefix computations, and bitonic merging and sorting. (We remark that all of these algorithms can also be efficiently implemented as ascend or descend algorithms.) The simple and regular structure of this class of algorithms has motivated the definition of several bounded-degree variants of the hypercube, called *hypercubic* networks, which can efficiently execute these algorithms. Examples of hypercubic networks are the butterfly, shuffle-exchange, and cube-connected cycles. For a formal definition of these networks, we refer the reader to [66]. In the present context, it suffices to know that any hypercubic network can simulate an arbitrary normal algorithm on a hypercube of the same size with only constant slowdown. Like the two-dimensional mesh and the torus, the hypercubic networks have served as the basis for a number of actual parallel machines [3].

1.2.3 Routing and Sorting on Fixed-Connection Networks

In our study of fixed-connection networks, we focus on the problems of routing and sorting. We restrict our attention to *store-and-forward* routing techniques; see [67] for an overview of other message routing methods. We define the *packet routing problem* on a fixed-connection network as the problem of rearranging a set of *packets* in the network such that every packet ends up at the processor specified in its destination address. Here, the address of a processor is determined by some fixed bijection $\mathcal{I} : V \mapsto [|V|]$ called an indexing scheme. A 1–1 routing problem, or *permutation routing problem*, is a routing problem in which each processor initially holds at most one packet, and each processor receives at most one packet. A routing problem in which each processor is the source of at most k_1 packets and the destination of at most k_2 packets is called a k_1–k_2 routing problem, or a k_1–k_2 relation.

We distinguish between *off-line* and *on-line* routing problems. In an off-line routing problem, the sources and destinations of all the packets are known in advance, and an appropriate schedule for the movement of the packets in the network can be computed before the start of the actual routing process. The problem of off-line routing is then to prove the existence of a schedule that can be executed within a certain time bound, and to compute such a schedule efficiently. In an on-line routing problem, the sources and destinations of the packets are not known in advance, and thus each processor has to decide its next action based only on the local information it has accumulated up to the current time. A special case of on-line routing is the *dynamic routing problem*, in which packets are continuously generated during a computation of the network.

In the *1–1 sorting problem*, we assume that every processor initially holds a single packet, where each packet contains a key drawn from a totally ordered set. Our goal is to rearrange the packets in such a way that the packet with rank i (i.e., with key of rank i) is moved to the processor with index i, for all i. Similarly, we can define the k–k sorting problem, where each processor initially holds k packets, and the packet with rank i has to be moved to the processor with rank $\lfloor i/k \rfloor$.

Note that a trivial lower bound for both routing and sorting is given by the diameter of the network. This implies a lower bound of $r(n-1)$ for the r-dimensional mesh, and of $\Omega(\lg n)$ for the hypercubic networks. A lower bound for k–k routing and sorting is given by the bisection width of the network. For the r-dimensional mesh, this bisection width is n^{r-1}, and hence at least $\frac{kn}{2}$ steps are needed in the case where all kn^r packets have to cross the bisection. In the remainder of this thesis, a routing or sorting algorithm for a hypercubic network is called *optimal* if its running time matches the lower bound within a constant factor. In contrast, an algorithm for a mesh-connected network is called *optimal* if its running time matches the lower bound within an additive lower order term.

Finally, we remark that there is a close relationship between the problems of routing and sorting on parallel machines. For example, a routing problem in which each processor is the source and destination of exactly one packet can be solved by sorting the packets with respect to their destination addresses. More generally, many routing algorithms for fixed-connection networks involve the sorting of small subsets of the packets, while many sorting algorithms use off-line routing in intermediate steps of the computation.

1.3 Sorting Networks

A circuit can be described by a directed acyclic graph $D = (V, E)$, where each vertex $v_i \in V$ corresponds to a hardware device g_i, and each directed edge $(v_i, v_j) \in E$ corresponds to a wire from an output of g_i to an input of g_j. In the case where there are several different kinds of hardware devices in the circuit, we assume an appropriate labeling of the set of vertices. If a device produces several different outputs, or if it performs a non-commutative operation on several inputs, then an additional labeling of the edges is needed to specify the order of the input and output values of the device.

Among the number of different classes of circuits that have been studied, the class of Boolean circuits built from *AND*, *OR*, and *NOT* gates has probably received the most attention. In this thesis, we are not concerned with Boolean circuits, but instead we focus on the class of *comparator* and *switching* circuits. The circuits in this class are built from *comparators* and *switches*, and have been extensively studied in the context of routing and sorting. In this thesis, we limit our attention to the problem of constructing efficient sorting circuits, or *sorting networks*. For an introduction to routing circuits (also often referred to as *interconnection networks*) and a survey of results, we refer the reader to [51, 87, 108].

A 2-input comparator is a device that compares two integer values supplied on its input wires, and then outputs the larger of the two values on the output wire labeled as the *maximum* output, and the smaller value on the output wire labeled as the *minimum* output. A 2-input switch is a device with two inputs x_0 and x_1 and two outputs y_0 and y_1. Depending on a special internal state called the *switch position*, two packets arriving on the inputs x_0 and x_1 are either directly moved to the corresponding outputs (that is, the packet on x_0 is moved to y_0, and the packet on x_1 is moved to y_1), or they are exchanged and then moved to the corresponding outputs. The switch position can either be set by some external process, or it can be locally computed in the switch. Thus, a comparator can be viewed as a switch in which the switch position is determined by a comparison of two values contained in the incoming packets. These definitions of comparators and switches can be generalized to the case of k inputs and outputs, $k \geq 2$, in a natural way.

Let C be a circuit of comparators and switches with n input nodes x_i, $0 \leq i < n$, and n output nodes y_i, $0 \leq i < n$. Then C is called a *sorting network* if, for any assignment of input values to the input nodes, the values will eventually appear in sorted order at the output nodes. Similarly, we can define *merging networks* that merge two or more sorted lists, or *selection networks* that select the input value of a specified rank. The *size* of a sorting network is given by the number of comparators, while the *depth* of a sorting network is defined as the length of the longest path from an input node to an output node. It follows from simple information-theoretic arguments that every sorting circuit has $\Omega(n \lg n)$ size and $\Omega(\lg n)$ depth. For a detailed introduction to the theory of sorting networks, and a survey of early results, we refer the reader to Knuth's text [50]. Some references to more recent work can be found in Section 2.1.

Most of the literature on sorting networks assumes that the network consists entirely of comparator elements, and that no copying of values is possible (i.e., no fan-out is allowed at the nodes). However, sometimes it may be convenient to relax some of these conditions. For example, Muller and Preparata [80] have designed a small-depth sorting circuit consisting of comparators and Boolean components. As another example, it is possible to design fault-tolerant circuits by allowing a constant fan-out at the nodes [5, 68]. In this thesis, we allow both comparators and switches to appear in a sorting network. As we will see in Chapter 2, this allows for an elegant definition of several natural classes of sorting networks that can be efficiently emulated on mesh-connected and hypercubic fixed-connection networks.

1.4 Summary of Thesis Results

In the following we give an overview of the main contributions of this thesis.

In Chapter 2, we prove lower bounds on the depth of *shuffle-unshuffle* sorting networks, a class of sorting networks whose structure corresponds to the class of normal algorithms on the hypercube. We first show a lower bound of $\Omega(\lg^2 n / \lg\lg n)$ for the special case of *shuffle-based* sorting networks, which correspond to the ascend and descend algorithms on the hypercube. Through a generalization of the proof technique, we then establish a lower bound of $\Omega\left(\frac{\lg n \lg\lg n}{\lg\lg\lg n}\right)$ for the entire class of shuffle-unshuffle sorting networks. The only previously known lower bound for these classes of networks was the trivial $\Omega(\lg n)$ bound. We also describe extensions of our lower bounds to restricted classes of non-oblivious sorting algorithms on hypercubes and multi-dimensional meshes. The results of this chapter are joint work with my advisor Greg Plaxton, and preliminary versions of the material have appeared in [90] and [91].

Shellsort is a well known sequential sorting paradigm that has also been used in the design of small depth sorting networks. While Shellsort algorithms have a very simple structure, it is often very difficult to analyze or bound their performance. In Chapter 3, we present general lower bounds on the running time of Shellsort networks and algorithms. We first give a fairly simple proof of a lower bound of $\Omega(n \lg^2 n/(\lg\lg n)^2)$ on the size of any Shellsort sorting network. This bound is then extended to the running time of non-oblivious Shellsort algorithms. The lower bounds establish a trade-off between the running time of a Shellsort algorithm and the length of the underlying increment sequence. This chapter also represents joint work with Greg Plaxton; a preliminary version was published in [89].

Chapter 4 considers the problems of permutation routing and sorting on meshes and tori. Over the last few years, a number of authors have proposed randomized algorithms for these problems that achieve a smaller running time or queue size than the best deterministic solutions. The main contribution of Chapter 4 is a technique that allows us to convert many of these randomized algorithms into deterministic algorithms with matching running times and queue sizes (within a lower order additive term). Using this technique, we derive a new deterministic routing algorithm for the two-dimensional mesh with a running time of $2n + o(n)$ and a queue size of 5, and a sorting algorithm with a running time of $2n + o(n)$ and a queue size of around 25. We also point out some other applications of this technique. This chapter describes joint work with Michael Kaufmann and Jop Sibeyn, and preliminary versions of this material have appeared in [47] and [111].

For routing and sorting on multi-dimensional meshes and tori, the running times of the fastest algorithms known are about a factor of 2 away from the diameter lower bound. In Chapter 5, we reduce this gap by giving improved upper and lower bounds for sorting in the *multi-packet* model of the mesh. For networks of sufficiently high constant dimension, our bounds are nearly tight. We also describe algorithms for permutation routing that nearly match the diameter bound. A preliminary version of these results has appeared in [112].

Chapter 6 studies the problems of permutation routing and sorting on several models of meshes with fixed and reconfigurable buses. Part of this material has been published in [113]. Finally, Chapter 7 contains some concluding remarks and directions for further research.

Chapter 2

Lower Bounds for Shuffle-Unshuffle Sorting Networks

This chapter considers *shuffle-unshuffle* sorting networks, a class of comparator networks whose structure maps efficiently to the hypercube and any of its bounded degree variants. Leighton and Plaxton [71, 88] have recently discovered a family of n-input shuffle-unshuffle sorting networks with depth $2^{O(\sqrt{\lg\lg n})}\lg n$; these networks are the only known sorting networks of depth $o(\lg^2 n)$ that are not based on expanders. In this chapter, we present a lower bound of $\Omega(\lg^2 n/\lg\lg n)$ for the subclass of *shuffle-based* networks. In addition, we establish a lower bound of $\Omega\left(\frac{\lg n \lg\lg n}{\lg\lg\lg n}\right)$ for the entire class of shuffle-unshuffle sorting networks, thus ruling out the existence of optimal, $O(\lg n)$-depth sorting networks in this class. We also describe a restricted class of non-oblivious sorting algorithms on the hypercube that is covered by our lower bounds.

2.1 Introduction

A variety of different classes of sorting networks have been described in the literature. Of particular interest here are the so-called AKS network [2] discovered by Ajtai, Komlós, and Szemerédi, and the sorting networks proposed by Batcher [7]. While the AKS network is the only known sorting network with $O(\lg n)$ depth, it also suffers from two significant shortcomings. First, the multiplicative constant hidden by the O-notation is impractically large. Through a series of improvements [18, 86], this constant has been reduced to below 2000, but remains impractical. Second, the structure of the network is highly irregular, and does not seem to map efficiently to any of the common interconnection schemes. For example, Cypher [25] has shown that any emulation of the AKS network on the cube-connected cycles requires $\Omega(\lg^2 n)$ time. (A sorting algorithm emulates the AKS network if it performs the same sequence of comparisons on any input.)

In contrast, the networks proposed by Batcher have a relatively simple structure and a small associated constant, and can be efficiently implemented on many common interconnection schemes, including meshes and hypercubic networks. This makes them the networks of choice in many practical applications, even though they have depth $\Theta(\lg^2 n)$ and are thus asymptotically inferior to AKS. This situation has motivated a number of attempts to construct $O(\lg n)$-depth sorting networks with simpler, more regular topologies, and/or a considerably smaller constant. Three classes of networks that have received particular attention are *Shellsort* networks, *periodic* sorting networks, and *shuffle-unshuffle* sorting networks.

Shellsort networks have a very simple structure that is based on the sequential *Shellsort* sorting algorithm. A class of Shellsort networks with depth $\Theta(\lg^2 n)$ was given by Pratt [95]. For Shellsort networks based on monotonically decreasing increment sequences, Cypher [24] has established a lower bound of $\Omega(\lg^2 n/\lg\lg n)$. A more general lower bound that holds for all Shellsort networks, and even non-oblivious Shellsort algorithms, is established in Chapter 3 of this thesis; similar bounds have also been established by Poonen [92]. These results answer in the negative the longstanding open question of whether a running time of $O(n \lg n)$ can be achieved by any Shellsort algorithm.

A comparator network is called a *periodic* sorting network if every input permutation can be sorted by repeatedly passing it through the network. The primary motivation for such periodic networks is the reduction in hardware cost achieved by applying the same network repeatedly to the input. A periodic sorting network of depth $O(\lg n)$ and running time $O(\lg^2 n)$ was given by Dowd, Perl, Rudolph, and Saks [28]. Very recently, Kutyłowski, Loryś, Oesterdiekhoff, and Wanka [60] have shown the existence of periodic networks of depth 5 and running time $O(\lg^2 n)$ based on expanders. No non-trivial lower bounds for periodic sorting networks are currently known.

In this chapter, we focus on the class of *shuffle-unshuffle* sorting networks, a notion that is formalized below. We establish a depth lower bound of $\Omega(\lg^2 n / \lg \lg n)$ for the subclass of *shuffle-based* sorting networks, and a lower bound of $\Omega\left(\frac{\lg n \lg \lg n}{\lg \lg \lg n}\right)$ for arbitrary *shuffle-unshuffle* sorting networks. In fact, our lower bound argument can be extended to certain restricted classes of non-oblivious sorting algorithms on hypercubic networks and multi-dimensional meshes. Before elaborating any further on these results, we will briefly describe the comparator network model, and define several classes of sorting networks.

2.1.1 Shuffle-Unshuffle Sorting Networks

In Section 1.3, a comparator network was defined as an acyclic circuit of comparator elements, each having two input wires and two output wires. We will use this model throughout most of this chapter, but will also briefly consider the following alternative model.

In this model, a comparator network on n registers is determined by a sequence of pairs $(\Pi_i, \vec{x}_i)$, $0 \leq i < \ell$, where Π_i is a permutation of $\{0, \ldots, n-1\}$ and $\vec{x}_i$ is a vector of length $\lfloor n/2 \rfloor$ over $\{+, -, 0, 1\}$. The network receives as input a permutation of $\{0, \ldots, n-1\}$ that is initially stored in the registers, and then operates on the input in ℓ consecutive steps. In step i, $0 \leq i < \ell$, the register contents are permuted according to Π_i, and then the operation stored in the kth component of $\vec{x}_i$ is applied to registers $2k$ and $2k+1$. In a "+" operation, the values stored in the two registers are compared, and the smaller of the values is stored in register $2k$, the larger one in $2k+1$. In a "−" operation, the values are stored in the opposite order. A "0" means that no operation takes place on the corresponding pair of registers. A "1" operation simply exchanges the values of the two registers. A comparator network is called a sorting network if it maps every possible input permutation to the same output permutation.

It is well known that these two models of comparator networks are equivalent. (That is, given any network in one model, there exists a network in the other model with the same size and depth that implements the same mapping from inputs to outputs.) While the first model often appears more intuitive, we can use the second one to define some interesting special classes of networks by restricting the possible choices for the permutations Π_i.

The *shuffle permutation* π_{sh} on $n = 2^d$ inputs may be defined as follows. If $j_{d-1} \cdots j_0$ denotes the binary representation of some integer j, $0 \leq j < n$, then $\pi_{sh}(j)$ has binary representation $j_{d-2} \cdots j_0 j_{d-1}$. A sorting network is called *shuffle-unshuffle* if $\Pi_i = \pi_{sh}$ or $\Pi_i = \pi_{sh}^{-1}$ holds for all i. A natural subclass of the shuffle-unshuffle sorting networks can be obtained by requiring $\Pi_i = \pi_{sh}$ for all i; we say that a network satisfying this condition is *shuffle-based*. Similarly, if $\Pi_i = \pi_{sh}^{-1}$ for all i, then the network is *unshuffle-based*.

The primary motivation for the definition of these two classes of networks is given by the fact that they can be efficiently implemented on any of the hypercubic fixed-connection networks (i.e., the hypercube, butterfly, cube-connected cycles, or shuffle-exchange). More precisely, the structure of the shuffle-unshuffle sorting networks corresponds exactly to the class of *normal* algorithms on the hypercube, while the structures of the shuffle-based and unshuffle-based networks correspond to the classes of *descend* and *ascend* algorithms, respectively (see Subsection 1.2.2 for a definition of these classes). Most of the algorithms that have been proposed for the hypercube are normal; important examples are Fast Fourier Transform, parallel prefix, bitonic merging and sorting. In fact, it can be argued that the primary motivation for the study of the bounded-degree variants of the hypercube (i.e., the butterfly, cube-connected cycles, and shuffle-exchange) has been the capability of these networks to efficiently implement the class of normal algorithms.

The study of shuffle-based sorting networks was proposed by Knuth [50, Exercise 5.3.4.47]. The best upper bound for this class is given by Batcher's bitonic sort [7], with a depth of $O(\lg^2 n)$.

The class of shuffle-unshuffle sorting networks was defined by Leighton and Plaxton [71, 88], who show the existence of a family of shuffle-unshuffle sorting networks with depth $2^{O(\sqrt{\lg\lg n})}\lg n$. The construction of these networks is based on a "probabilistic" sorting network described in [70], which sorts all but a superpolynomially small fraction of the possible input permutations. We point out that the depth of the above networks is $o(\lg^{1+\epsilon} n)$, for all $\epsilon > 0$, and that they represent the only known sorting networks of depth $o(\lg^2 n)$ that are not based on expanders. Naturally, this raises the question of whether a depth of $O(\lg n)$ can be achieved by any shuffle-unshuffle sorting network.

2.1.2 Overview of this Chapter

In the following sections, we resolve this question by showing a lower bound of $\Omega\left(\frac{\lg n \lg\lg n}{\lg\lg\lg n}\right)$ on the depth of any shuffle-unshuffle sorting network. We also show a stronger lower bound of $\Omega(\lg^2 n/\lg\lg n)$ for the class of shuffle-based sorting networks, thus establishing a separation between the power of strictly shuffle-based (or unshuffle-based) networks, and networks in which both shuffling and unshuffling is allowed. Our lower bounds also extends to certain restricted classes of non-oblivious sorting algorithms on hypercubic machines and multi-dimensional meshes. However, our lower bound argument does not allow the copying of elements by the algorithm. Thus, the *Sharesort* sorting algorithm of Cypher and Plaxton [26], which achieves a running time of $O(\lg n \lg\lg n)$ (with preprocessing) on any of the hypercubic machines, is not subject to our lower bound. Nonetheless, we believe that our present results are already interesting in their own right, and that they may constitute an important step towards more general lower bounds for sorting on hypercubic machines.

The remainder of this chapter is organized as follows. Section 2.2 describes the basic ideas underlying our proof technique. Section 2.3 contains some useful definitions and lemmas. Section 2.4 proves the lower bound for shuffle-based networks. Section 2.5 then shows the lower bound for arbitrary shuffle-unshuffle sorting networks. Some extensions and limitations of our proof technique are discussed in Section 2.6. Finally, Section 2.7 lists some open questions for future research.

2.2 Proof Ideas

In this section, we give a very informal description of the most important ideas in the proofs of the lower bounds. We first outline the lower bound argument for shuffle-based networks given in Section 2.4. We then explain why this relatively simple argument does not extend to the more general class of shuffle-unshuffle sorting networks, and describe the additional ideas that are needed in order to get a lower bound for this class.

2.2.1 A Naive Proof Idea

A simple observation concerning comparator networks is that a sorting network must perform a comparison on every pair of adjacent values in every input, that is, every pair of values $\{m, m+1\}$ must appear on the input wires of some comparator element. (We assume the inputs to be permutations of $\{0, \ldots, n-1\}$.) Thus, one might attempt to prove a lower bound of ℓ for the depth of a class of comparator networks by showing, for all networks in the class, the existence of an input permutation π, and of a set of adjacent values $\{m, \ldots, m+i\}$ in π, such that no two elements of the set are compared up to level ℓ of the network. In the following, we will call such a set an *incomparable set*. If we apply this proof idea to a shuffle-unshuffle network, starting out with the set of all values as our incomparable set, and, whenever two elements of the set get compared, removing one of them from the set, then we might lose up to half of the elements in any given level. So using this simple approach, we could only show the trivial lower bound of $\Omega(\lg n)$ for the depth of a sorting network.

2.2.2 The Proof for Shuffle-Based Sorting Networks

The key idea to overcome this problem is to modify the proof technique in a way that allows us to exploit the structural properties of the particular class of networks that we are studying. To explain this idea, we consider the case of the shuffle-based networks; the case of the unshuffle-based networks is symmetric. Note that a shuffle-based network can be seen as a concatenation of a number of butterfly networks of depth $\lg n$ each. Thus, if we can show that the size of our incomparable set decreases by at most a polylogarithmic factor in each butterfly, then at least $\Omega(\lg n / \lg\lg n)$ consecutive butterflies are needed in order to bring the size of the incomparable set down to 1; this directly implies the $\Omega(\lg^2 n / \lg\lg n)$ lower bound for shuffle-based sorting networks.

The following recursive definition of a butterfly is crucial for understanding our proof technique: A butterfly with 2^d inputs and depth d consists of two parallel 2^{d-1}-input butterflies of depth $d-1$, followed by a final level of up to 2^{d-1} comparators. Every comparator in the final level takes one input from the outputs of each of the two 2^{d-1}-input subnetworks. Finally, a 1-input butterfly is just a wire. This "tournament-like" structure leads to the following important property of a butterfly: An observer of a 2^d-input butterfly tournament who sees the outcomes of all comparisons in the two 2^{d-1}-input subnetworks, but not the outcomes of the final level of comparisons, will not be able to say anything about the relative ordering of any two items taken from different subnetworks. In other words, the observer will not be able to say anything about the relative strength of the two "subtournaments" before the final stage. This "disjointness property" of the subnetworks plays a crucial role in the lower bound argument.

Instead of maintaining only a single incomparable set, we now maintain a collection of incomparable sets in each recursive subnetwork. More precisely, after entering a new butterfly of depth $\lg n$, we partition our current incomparable set into $n \lg^3 n$ disjoint incomparable sets, most of which are empty, with $\lg^3 n$ sets entering on each wire (recall that a single wire is a 1-input butterfly). Thus, every 2-input butterfly has two different collections of $\lg^3 n$ incomparable sets arriving on its two input wires. It is now possible to recombine these sets to get a new collection of roughly $\lg^3 n$ incomparable sets, containing all of the elements of the two collections.

More generally, due to the recursive structure of a butterfly, in every level we recursively have two different collections of $\Theta(\lg^3 n)$ incomparable sets coming from two disjoint subnetworks. We show that there exists a partial matching between these two collections of sets such that, if we combine the sets according to the matching and remove one element from every pair of elements from the same set that gets compared, we obtain a new collection of incomparable sets while losing only a very small fraction of our elements. The number of sets in this new collection is only slightly larger than the number of sets in either of the two previous collections. The aforementioned "disjointness property" of the two subnetworks is needed at this point to make sure that the new sets in the collection each contain adjacent elements, under some appropriately chosen set of input permutations.

If we repeat this process over all $\lg n$ levels of the butterfly, then we end up with a single collection of $\Theta(\lg^3 n)$ incomparable sets. The total number of elements in the sets is only a constant factor smaller than it was when we entered the butterfly. If we pick the largest of the $\Theta(\lg^3 n)$ sets as our new incomparable set, then we only lose a polylogarithmic factor in the size of the set.

To formalize this proof idea, Section 2.3 introduces the notion of an *input pattern* representing a class of similar inputs. A class of inputs with the desired property (existence of a large incomparable set) is then constructed in Section 2.4 by stepwise *refinement* of a given input pattern in every level of the network.

2.2.3 The Proof for Shuffle-Unshuffle Sorting Networks

The above argument does not work for arbitrary shuffle-unshuffle networks, as they do not satisfy the "disjointness property" of the two subnetworks used in the argument. To overcome this obstacle and prove a lower bound for arbitrary shuffle-unshuffle sorting networks, we introduce the class of shuffle-unshuffle networks with "bounded overlap".

Assume we are given an arbitrary shuffle-unshuffle network Λ with ℓ levels $(\Pi_i, \vec{x}_i)$, $0 \leq i < \ell$, as described in the register model of a comparator network. In order to define the "span" and "overlap" of Λ, it is convenient to introduce a number of auxiliary variables. Let $a_i = 1$ if $\Pi_i = \pi_{sh}$ and $a_i = -1$ if $\Pi_i = \pi_{sh}^{-1}$, $0 \leq i < \ell$. (We remark that the value of a_0 has no impact on the definitions that follow.) Let $b_i = \sum_{1 \leq j \leq i} a_j$, $0 \leq i < \ell$. The *span* of Λ may now be defined as

$|\{b_i : 0 \leq i < \ell\}|$. The *overlap* of Λ is the minimum integer $r \geq 0$ such that either: (i) $b_i \leq b_j + r$ for all $0 \leq i < j < \ell$, or (ii) $b_i \geq b_j - r$ for all $0 \leq i < j < \ell$. Note that a network has overlap 0 iff $\Pi_i = \Pi_j$ for all $1 \leq i < j < \ell$. Furthermore, the span of a network is always at least as large as its overlap, with equality occurring only in the case $\ell = 0$, where the span and overlap are both 0.

The proof of the general lower bound in Section 2.5 is based on two main additional ideas. First, we show in Subsection 2.5.1 how the lower bound argument for shuffle-based networks can be modified to handle shuffle-unshuffle networks with small overlap. The overall structure of this proof is very similar to that for shuffle-based networks. However, a number of subtle changes are required in order to extend the argument to networks with non-zero overlap. The modified proof is based on the observation that, informally speaking, a shuffle-based network with small overlap still satisfies some relaxed version of the "disjointness property". More precisely, we obtain a trade-off between the overlap of the network and the lower bound that can be shown.

Second, we show in Subsection 2.5.2 that any shuffle-unshuffle network can be partitioned into a number of consecutive shuffle-unshuffle networks such that the overlap of each network in the partition is sufficiently smaller than its depth.

2.3 Definitions and Basic Lemmas

This section contains a number of definitions and lemmas that are needed for the proof of the lower bound. In the first subsection, we introduce the concepts of *input patterns* and *input pattern refinement.* Subsection 2.3.2 defines our notion of a comparator network and its action on an input pattern, and introduces the class of reverse delta network. Finally, Subsection 2.3.3 contains a few basic lemmas.

In the following, unless explicitly stated otherwise, the set of *input wires* of a comparator network is denoted W. An *input* to a comparator network is a total mapping from W to a set V of possible *input values*. We will restrict our attention to inputs π that are permutations of $[n]$, i.e., where $|W| = n$, $V = [n]$, and π is one-to-one. The set of all one-to-one functions from a set A to a set B will be denoted by $(A \mapsto B)$, and so the set of all inputs of a given comparator network may be written as $(W \mapsto V)$. Furthermore, for a function f on a set A and a subset B of A, let $f_{|_B}$ denote the functional restriction of f to B. For two functions f_0 and f_1 on disjoint sets A_0 and A_1, we write $f_0 \oplus f_1$ for the *union* of f_0 and f_1:

$$(f_0 \oplus f_1)(x) \stackrel{\text{def}}{=} \begin{cases} f_0(x) & \text{for all } x \text{ in } A_0, \text{ and} \\ f_1(x) & \text{for all } x \text{ in } A_1. \end{cases}$$

2.3.1 Input Patterns and Refinement

In the following definitions, we introduce the notions of *input patterns* and *input pattern refinement*, which are fundamental to our proof technique. Informally, an input pattern describes a set of inputs with certain common properties. Input pattern refinement is the process of imposing additional constraints on such a set of inputs.

Definition 2.3.1 *Let P be a set and $<_P$ be a total ordering on P.*

(a) An input pattern *is a total mapping from W to P.*

(b) Let p_0, p_1 be two input patterns. We say that p_0 can be refined to p_1 (written $p_0 \supset_W p_1$) if $(p_0(w) <_P p_0(w')) \Rightarrow (p_1(w) <_P p_1(w'))$ holds for all w and w' in W.

(c) Let p be an input pattern and π be an input. We say that p can be refined to π (written $p \supset_W \pi$) if $(p(w) <_P p(w')) \Rightarrow (\pi(w) < \pi(w'))$ holds for all w and w' in W.

The set P will be referred to as the *pattern alphabet*, and the elements of P are called *pattern symbols*. Throughout this chapter, pattern symbols are denoted by script letters.

Example 2.3.1 *Let* $W \stackrel{def}{=} \{w_0, \ldots, w_{n-1}\}$*,* $P \stackrel{def}{=} \{\mathcal{S}, \mathcal{M}, \mathcal{L}\}$*, and let the ordering* $<_P$ *on* P *be given by* $\mathcal{S} <_{\mathcal{P}} \mathcal{M} <_{\mathcal{P}} \mathcal{L}$*. (Informally, the symbols* $\mathcal{S}$*,* $\mathcal{M}$*, and* $\mathcal{L}$ *may be interpreted as "Small", "Medium", and "Large", respectively.) Then the input pattern* p *assigning* $\mathcal{L}$ *to* w_0 *and* w_1 *and* $\mathcal{M}$ *to all other wires can be refined to all inputs that assign the two largest values to* w_0 *and* w_1*. We could also refine* p *to other input patterns, for example to a pattern* p' *such that* $\mathcal{L}$ *is assigned to* w_0 *and* w_1*,* $\mathcal{S}$ *is assigned to* w_2*, and* $\mathcal{M}$ *is assigned to all other wires. The new pattern* p' *can itself be refined to all inputs that assign the largest values to* w_0 *and* w_1*, and the smallest value to* w_2*.*

The relation $\supset_W$ defined above is a partial ordering on the set of input patterns. Note that the set V of input values can be regarded as a special case of a pattern alphabet with the ordering of the natural numbers. Every pattern can be refined to some input, and we could assume that the pattern alphabet P is always a subset of V. The pattern-to-pattern refinement in Part (b) of Definition 2.3.1 would then become a special case of the pattern-to-input refinement in Part (c). However, in the following we will not restrict our choice of P to subsets of V. We will see that this gives us more power of expression and, thus, simplifies the presentation of the proof.

We may think of an input pattern p as a description of the set of inputs to which p can be refined. This set is denoted $p[V] \stackrel{\text{def}}{=} \{\pi : \pi \text{ is an input such that } p \supset_W \pi\}$. When we refine a pattern p_0 to p_1, then we are imposing additional constraints on this set of inputs. Formally, we have $(p_0 \supset_W p_1) \Leftrightarrow (p_0[V] \supseteq p_1[V])$. Alternatively, the reader may also view an input pattern p as a shorthand for a logical predicate that holds for exactly the inputs in $p[V]$.

Definition 2.3.2 *Let* p *and* q *be input patterns on* W*, and let* U *be a subset of* W*.*

(a) *The input pattern* $p_{|U}$ *on* U *is the* restriction *of* p *to* U*.*

(b) *We say that* p *can be* U*-refined to* q *(written* $p \supset_U q$*) if* $p \supset_W q$ *and* $p(w) = q(w)$ *holds for all* w *in* $W \setminus U$*.*

Definition 2.3.3 *Let* U_0 *and* U_1 *be disjoint subsets of* W*,* p_0 *be an input pattern on* U_0*, and* p_1 *be an input pattern on* U_1*. Then* $q = p_0 \oplus p_1$ *is the input pattern on* $U_0 \cup U_1$ *such that* $q_{|U_0} = p_0$ *and* $q_{|U_1} = p_1$*.*

If for two patterns p_0 and p_1 both $p_0 \supset_W p_1$ and $p_1 \supset_W p_0$ hold, then we say that p_0 and p_1 are *equivalent*. In this case, we have $p_0[V] = p_1[V]$, and the refinement steps from p_0 to p_1 and vice versa can be achieved by simply renaming the pattern symbols in a way that preserves the ordering $<_P$. Hence, we call this special case of a refinement step an *order-preserving renaming*.

Example 2.3.2 *Let* $W \stackrel{def}{=} \{w_0, \ldots, w_{n-1}\}$ *and* $P \stackrel{def}{=} \{\mathcal{P}_i : i \geq 0\}$ *with* $\mathcal{P}_i <_P \mathcal{P}_{i+1}$ *for all* $i \geq 0$*. Then any input pattern* p *is equivalent to the input pattern* p_k*,* $k \geq 0$ *obtained from* p *by substituting every pattern symbol* $\mathcal{P}_i$ *in* p *by* $\mathcal{P}_{i+k}$*, for all* i*.*

2.3.2 Comparator Networks

We now further formalize our notion of a comparator network, and explain how its domain of operation can be extended from the set of inputs to the set of input patterns.

In the following, a comparator network is interpreted as a mapping from a set of possible inputs to a set of possible outputs. More precisely, a comparator network Λ on input wires W and output wires W' defines a mapping (which we also denote by Λ) from $(W \mapsto V)$ to $(W' \mapsto V)$ such that every input $\pi : W \mapsto V$ is mapped to an output $\pi' : W' \mapsto V$ that is a "permutation" of π. By this we mean that there exists a bijection $\rho : W \mapsto W'$ such that $\pi(w) = \pi'(\rho(w))$ holds for all w in W.

Let Λ_0^*, Λ_1^* be two sets of n-input comparator networks. Then $\Lambda_0^* \otimes \Lambda_1^*$, the *serial composition* of Λ_0^* and Λ_1^*, denotes the set of all networks Λ that can be obtained by connecting the output wires of a network from Λ_0^* to the input wires of a network from Λ_1^*. In some cases, we may want to impose certain special conditions on this connection between the output wires of the first network and the input wires of the second network. If no conditions are stated, then the connections can be made according to an arbitrary one-to-one mapping. As it happens, we often make use of the serial composition operator in the context of singleton sets Λ_0^* and Λ_1^*. In such a case, we may write, for example, $\Lambda_0 \otimes \Lambda_1$ (where Λ_0, Λ_1 are networks) rather than $\{\Lambda_0\} \otimes \{\Lambda_1\}$.

Given two comparator networks Λ_0 and Λ_1 on disjoint sets of input and output wires, we obtain the *parallel composition* of Λ_0 and Λ_1 as the union of the two networks, written $\Lambda_0 \oplus \Lambda_1$. The set of input (output) wires of $\Lambda_0 \oplus \Lambda_1$ is the union of the sets of input (output) wires of Λ_0 and Λ_1. Given these definitions, we can now formally define the class of reverse delta networks.

Definition 2.3.4 *A 2^s-input comparator network Δ is called an s-level reverse delta network if*

- *$s = 0$ and Δ contains no comparator elements, or*
- *$s > 0$ and Δ is an element of $(\Delta_0 \oplus \Delta_1) \otimes \Gamma_s$, where*

  (i) Δ_0 and Δ_1 are $(s-1)$-level reverse delta networks, and

  (ii) Γ_s consists of one level with at most 2^{s-1} comparator elements,

  such that every comparator in Γ_s takes one input from an output wire of Δ_0 and the other input from an output wire of Δ_1.

Note that we do not require the ith level to have exactly 2^{i-1} comparator elements. This corresponds to allowing the reverse delta network to contain "0" (do nothing) and “1” (exchange) circuit elements, as introduced in the “register model” of a comparator network.

We call a network Δ an (l, s)-iterated reverse delta network if it consists of l consecutive s-level reverse delta networks, or, formally, if Δ belongs to $\Delta_0 \otimes \cdots \otimes \Delta_{l-1}$ where every Δ_i is an s-level reverse delta network. It should be pointed out that this definition allows an arbitrary fixed permutation between any two consecutive reverse delta networks, due to our definition of serial composition. Recall that we allowed both comparators and switching elements in our network. For this model it has been shown that any permutation on $n = 2^d$ inputs can be routed by a shuffle-exchange network with $3d - 4$ levels [85, 76, 118]. Thus, eliminating the permutations between the reverse delta networks would only increase the depth of the circuit by at most a constant factor.

A comparator network Λ was identified with a mapping from the set of inputs to the set of outputs. The following definition extends Λ to a function from the set of input patterns to the set of output patterns. (An output pattern is a mapping from the set of output wires to the set of pattern symbols.)

Definition 2.3.5 *Given a comparator network Λ, an input pattern p_0, and an output pattern p_1 such that $p_1(W) = p_0(W)$, we define*

$$\Lambda(p_0) = p_1 \Leftrightarrow \Lambda(p_0[V]) = p_1[V].$$

Note that this definition characterizes the behavior of a comparator network on an input pattern in the way we would expect: If two pattern symbols $\mathcal{P}_0$ and $\mathcal{P}_1$ arrive on the input wires of a comparator gate, then the symbol that is larger according to the ordering $<_P$ will appear on the max-output of the gate, and the smaller one will appear on the min-output. This implies that any set of inputs that can be expressed by an input pattern will produce a set of outputs that can be expressed by an output pattern.

Definition 2.3.6 *We say that two input wires w_0 and w_1 collide in a network Λ under an input π if the input values $\pi(w_0)$ and $\pi(w_1)$ are compared in Λ when π is given as input.*

According to the above definition, two wires whose respective values meet in a noncomparator element, that is, a "0" (do nothing) or "1" (exchange) switch, are not regarded as colliding. In the rest of the chapter, we do not have to distinguish between the different circuit elements any more, since the entire lower bound argument is based on the notion of collision introduced above and extended to input patterns in the following.

Given a network Λ and an input π, we can always determine whether two input values are compared or not. (Recall that we only consider inputs that are permutations.) This is not the case for input patterns, since an input pattern can contain several occurences of the same pattern symbol. This motivates the following definition of collision for input patterns:

Definition 2.3.7 *Let Λ be a comparator network, let p be an input pattern for Λ, and let w_0 and w_1 be two input wires of Λ.*

(a) We say that w_0 and w_1 collide *in Λ under p if they collide in Λ under every input in $p[V]$.*

(b) We say that w_0 and w_1 can collide *in Λ under p if there exists an input in $p[V]$ such that w_0 and w_1 collide in Λ.*

(c) We say that w_0 and w_1 cannot collide *in Λ under p if there is no input in $p[V]$ such that w_0 and w_1 collide in Λ.*

(d) A set $U \subset W$ is called non-colliding *in Λ under p if any two wires in U cannot collide in Λ under p.*

Example 2.3.3 *Let* $W \stackrel{def}{=} \{w_0, w_1, w_2, w_3\}$*,* $P \stackrel{def}{=} \{\mathcal{S}, \mathcal{M}, \mathcal{L}\}$*, and let the ordering* $<_P$ *on* P *be given by* $\mathcal{S} <_P \mathcal{M} <_P \mathcal{L}$*. Let the network* Λ *consist of a comparator between* w_1 *and* w_2*, followed by a comparator between* w_2 *and* w_3*, followed by a comparator between* w_0 *and* w_3*, where all comparators are directed towards the wire with the larger index. Then the following holds under the input pattern* p *that maps* w_0 *to* $\mathcal{S}$*,* w_1 *and* w_2 *to* $\mathcal{M}$*, and* w_3 *to* $\mathcal{L}$*:*

(1) Wires w_1 *and* w_2 *collide in* Λ *under* p *since the very first comparator is between these two wires.*

(2) Wires w_1 *and* w_3 *can collide in* Λ *under* p*, since we can refine* p *to an input* π *that assigns a larger value to* w_1 *than to* w_2*. In that case, the input value assigned to* w_1 *will be compared to that of* w_3 *in the second comparator. Similarly,* w_2 *can collide with* w_3 *in* Λ *under* p*.*

(3) Wires w_0 *and* w_3 *collide in* Λ *under* p*, since no exchange can occur in the second comparator of the network under any input* π *with* $p \supset_W \pi$*. Also,* w_0 *and* w_1 *(resp.* w_2*) cannot collide in* Λ *under* p*.*

Note that, if two wires collide (cannot collide) in some network Λ under an input pattern p, then they also collide (cannot collide) in Λ under any refinement p' of p. Similarly, if a set U is non-colliding in Λ under p, then it is also non-colliding in Λ under p'. The property *can collide* is not preserved under arbitrary refinement.

In the following we restrict our attention to a fixed pattern alphabet P which is used throughout the lower bound argument:

$$P \stackrel{\mathrm{def}}{=} \{\mathcal{S}_i, \mathcal{X}_{i,j}, \mathcal{M}_i, \mathcal{L}_i : i, j \geq 0\}.$$

The ordering $<_P$ on P is defined by

$$\begin{aligned} \mathcal{S}_i &<_P \mathcal{S}_{i+1}, \\ \mathcal{S}_i &<_P \mathcal{X}_{0,0}, \end{aligned}$$

$$\begin{array}{rcl} \mathcal{X}_{i,j} & <_P & \mathcal{X}_{i,j+1}, \\ \mathcal{X}_{i,j} & <_P & \mathcal{M}_i, \\ \mathcal{M}_i & <_P & \mathcal{X}_{i+1,0}, \\ \mathcal{M}_i & <_P & \mathcal{L}_j, \text{ and} \\ \mathcal{L}_{i+1} & <_P & \mathcal{L}_i, \end{array}$$

for all nonnegative integers i, j.

Definition 2.3.8 *For a pattern p and a pattern symbol $\mathcal{P}$ we define the [$\mathcal{P}$]-set of p as the set $\{w \in W : p(w) = \mathcal{P}\}$.*

Definition 2.3.9 *We say that a comparator network Λ has an* incomparable set *of size m if there exists an input pattern p and an integer i such that the [$\mathcal{M}_i$]-set of p is of size m and is non-colliding in Λ under p.*

2.3.3 Basic Lemmas

The following lemmas will be used in our lower bound argument. Their proofs are fairly straightforward and we will only sketch some of the proof ideas.

Lemma 2.3.1 *Let p be an input pattern on W such that only the pattern symbols $\mathcal{S}_0$, $\mathcal{M}_0$, and $\mathcal{L}_0$ appear in p. Let W_0 and W_1 be disjoint subsets of W with $W = W_0 \cup W_1$ and let A be the [$\mathcal{M}_0$]-set of p. Let q_0 and q_1 be input patterns on W_0 and W_1, respectively, with $\mathcal{S}_0 <_P q_0(w), q_1(w) <_P \mathcal{L}_0$ for all w in A. Then from $p_{|W_0} \supset_{A \cap W_0} q_0$ and $p_{|W_1} \supset_{A \cap W_1} q_1$, we can infer $p \supset_A q_0 \oplus q_1$.*

This lemma ensures that, given an input pattern p for a network $\Lambda = \Lambda_0 \oplus \Lambda_1$, we obtain a refinement of p if we separately refine the input patterns $p_{|W_0}$ for Λ_0 and $p_{|W_0}$ for Λ_1 according to the above rules, where W_0 and W_1 are the sets of input wires of Λ_0 and Λ_1, respectively.

Lemma 2.3.2 *Let Λ be a comparator network, p be an input pattern for Λ, and A be the $[\mathcal{M}_i]$-set of p. If A is non-colliding in Λ under p, then for every input wire w in A there exists a unique output wire w' such that $\pi(w) = \Lambda(\pi)(w')$ holds for all π in $p[V]$.*

Informally, the above lemma states that an input value on a wire w in a non-colliding $[\mathcal{M}_i]$-set follows the same "path" through the network under all inputs in $p[V]$. The proof of the lemma is by a simple induction on the depth of the network. This one-to-one correspondence between the input and output wires of a non-colliding $[\mathcal{M}_i]$-set is also the underlying idea in the next lemma.

Lemma 2.3.3 *Let Λ be a comparator network in $\Lambda_0 \otimes \Lambda_1$, i be a nonnegative integer, and p be an input pattern for Λ_0 such that its $[\mathcal{M}_i]$-set A is non-colliding in Λ_0 under p. Let $q \stackrel{def}{=} \Lambda_0(p)$ be an input pattern for Λ_1 and B be the $[\mathcal{M}_i]$-set of q. Then for every q' with $q \supset_B q'$ there exists a p' with $p \supset_A p'$ such that $q' = \Lambda_0(p')$. Furthermore, if the $[\mathcal{M}_i]$-set of q' is non-colliding in Λ_1 under q', then the $[\mathcal{M}_i]$-set of p' is non-colliding in Λ under p'.*

To verify the validity of the final lemma, note that the paths taken by the $\mathcal{M}_i$-symbols through a network are not changed if we rename the rest of the symbols in the way described in the lemma.

Lemma 2.3.4 *Let Λ be a comparator network, p be an input pattern for Λ, and A be the $[\mathcal{M}_i]$-set of p. Let $\rho_i(p)$ be the input pattern obtained from p by changing all pattern symbols $\mathcal{P}$ with $\mathcal{P} <_P \mathcal{M}_i$ to $\mathcal{S}_0$, all pattern symbols $\mathcal{P}$ with $\mathcal{M}_i <_P \mathcal{P}$ to $\mathcal{L}_0$, and all pattern symbols $\mathcal{M}_i$ to $\mathcal{M}_0$. If A is non-colliding in Λ under p, then A is also non-colliding in Λ under $\rho_i(p)$.*

2.4 Bounds for Shuffle-Based Networks

This section contains the proof of the lower bound for sorting on shuffle-based networks. Before giving the formal proof, we briefly describe the proof strategy in terms of the definitions of the previous section.

2.4.1 Proof Strategy

To prove that a network Λ is not a sorting network, we will show that the network has an incomparable set of size at least 2. The input pattern p associated with the incomparable set can then be refined to an input such that the wires in the $[\mathcal{M}_i]$-set contain adjacent input values; this implies that Λ does not sort all inputs in $p[V]$. The input pattern p will be constructed using stepwise refinement, starting out with a pattern containing only the symbol $\mathcal{M}_0$.

In general, we will assume that whenever we enter a new reverse delta network the current pattern p only contains the pattern symbols $\mathcal{M}_0$, $\mathcal{S}_0$, and $\mathcal{L}_0$, with the latter two symbols marking the input wires carrying values that are smaller and larger, respectively, than those of the wires in the $[\mathcal{M}_0]$-set.

We then split up the pattern p into n patterns p_i, $0 \leq i < n$, of size 1, with one p_i corresponding to each input wire (1-input reverse delta network). Every pattern p_i can be interpreted as having $\lg^3 n$ non-colliding sets $M_0, \ldots, M_{\lg^3 n-1}$, where M_j is the $[\mathcal{M}_j]$-set of p_i, for $0 \leq j < \lg^3 n$. Except for M_0, all of these sets will be empty at this point.

Thus, every 2-input reverse delta network will have two collections of $[\mathcal{M}_i]$-sets, denoted by $M_{0,0}, \ldots, M_{0,t-1}$ and $M_{1,0}, \ldots, M_{1,t-1}$, where $t = \lg^3 n$, entering on the first and second input wire, respectively. In general, in every level of the recursive definition of a reverse delta network we will have two collections of $\Theta(\lg^3 n)$ non-colliding $[\mathcal{M}_j]$-sets coming from each of the two disjoint subnetworks. We will be able to recombine these collections to obtain a single collection of non-colliding

$[\mathcal{M}_j]$-sets such that this single collection still contains nearly all of the input wires that were in either of the two collections, while the number of sets will only increase marginally. Hence, on average, the new sets will contain roughly twice as many elements as the old sets.

This proof step is performed by showing the existence of an appropriate matching between the two collections, and refining the two input patterns according to this matching. After the last level of the reverse delta network, we will have a collection of $\Theta(\lg^3 n)$ non-colliding sets containing only a constant factor fewer elements than the "original" $[\mathcal{M}_0]$-set before the current reverse delta network. We can choose the largest of these sets as our new non-colliding $[\mathcal{M}_0]$-set by performing an order-preserving renaming of the pattern p, mapping the wires in this set to $\mathcal{M}_0$ and all of the wires in the other sets to some $\mathcal{S}_i$ or $\mathcal{L}_i$. This procedure is iterated over $\Theta(\frac{\lg n}{\lg \lg n})$ consecutive reverse delta networks.

2.4.2 The Proof

The proof is divided into several steps: First, Lemma 2.4.1 establishes the existence of a pattern p with a "large" $[\mathcal{M}_0]$-set that is non-colliding in a single reverse delta network under p. This is the main part of our proof, and also the one that contains the novel proof ideas. This lemma is used by Lemmas 2.4.2 and 2.4.3 to show that a fairly large incomparable set can be maintained over several consecutive reverse delta networks in an iterated reverse delta network. Finally, a corollary establishes the lower bound.

We point out that Lemma 2.4.1 is actually a special case of Lemma 2.5.1, which will be established in the next section. Nonetheless, we have chosen to give a complete proof of the lemma at this point. We believe that the special case treated in Lemma 2.4.1 is somewhat simpler and more intuitive than Lemma 2.5.1, and that it may help the reader in understanding the more general results of the next section.

Lemma 2.4.1 *Let Δ be an s-level reverse delta network, $s \geq 0$, and let p be an input pattern for Δ such that only the pattern symbols $\mathcal{S}_0$, $\mathcal{L}_0$, and $\mathcal{M}_0$ occur in p. Let A be the $[\mathcal{M}_0]$-set of p, and let k be any positive integer. Then there exists an input pattern q with $p \supset_A q$ and $t(s) \stackrel{def}{=} k^3 + sk^2$ sets $M_0, \ldots, M_{t(s)-1}$ of input wires such that the following properties hold, where $B \stackrel{def}{=} \bigcup_{0 \leq i < t(s)} M_i$:*

(1) Every M_i is the $[\mathcal{M}_i]$-set of q,

(2) Every M_i is non-colliding in Δ under q,

(3) $B \subset A$, and

(4) $|B| \geq |A| - \frac{s \cdot |A|}{k^2}$.

Proof: We will prove the lemma by induction over s, the number of levels in the reverse delta network.

Base Case: $s = 0$

We define the sets $M_0, \ldots, M_{t(0)-1}$ by setting M_0 to A and all M_i, $1 \leq i < t(0)$, to the empty set. If we set $q = p$, then Properties (1) to (4) are satisfied. In particular, Property (2) is satisfied since a 0-level reverse delta network does not contain any comparators, and hence every set is non-colliding in the network under every input pattern.

Induction Step: $s > 0$

An s-level reverse delta network Δ consists of two $(s-1)$-level reverse delta networks Δ_0 and Δ_1, and an sth level Γ_s satisfying the conditions of Definition 2.3.4. The input wires W of Δ can be partitioned into the sets W_0 and W_1 of input wires of Δ_0 and Δ_1, respectively. Let $p_0 \stackrel{\mathrm{def}}{=} p_{|W_0}$ and $p_1 \stackrel{\mathrm{def}}{=} p_{|W_1}$. Then $A_0 \stackrel{\mathrm{def}}{=} A \cap W_0$ is the $[\mathcal{M}_0]$-set of p_0 and $A_1 \stackrel{\mathrm{def}}{=} A \cap W_1$ is the $[\mathcal{M}_0]$-set of p_1.

Applying the induction hypothesis to Δ_0, p_0, and A_0 we can infer the existence of an input pattern q_0 with $p_0 \supset_{A_0} q_0$, and of $t(s-1)$ disjoint sets $M_{0,i}$, $0 \leq i < t(s-1)$, such that

- every $M_{0,i}$ is the $[\mathcal{M}_i]$-set of q_0,
- every $M_{0,i}$ is non-colliding in Δ_0 under q_0,
- $B_0 \subset A_0$, and
- $|B_0| \geq |A_0| - \frac{(s-1)\cdot|A_0|}{k^2}$,

where $B_0 \stackrel{\text{def}}{=} \bigcup_{0 \leq i < t(s-1)} M_{0,i}$.

Correspondingly, for Δ_1, p_1, and A_1 we get an input pattern q_1, disjoint sets $M_{1,i}$, $0 \leq i < t(s-1)$, and a set B_1, with the same properties.

We will now construct the sets M_i, $0 \leq i < t(s)$, by combining the sets $M_{0,i}$ of Δ_0 with the sets $M_{1,j}$ of Δ_1, according to some partial matching to be determined in the following.

Note that, due to the topology of a reverse delta network, no element of a set $M_{0,i}$ can collide in Δ with any element of a set $M_{1,j}$ before level s. Also, because of Lemma 2.3.2, any two elements w_0 in $M_{0,i}$ and w_1 in $M_{1,j}$ either collide in level s of Δ under $q_0 \oplus q_1$, or they cannot collide in that level.

For $0 \leq i, j < t(s-1)$, we define $C_{i,j}$ as the set of all w_0 in $M_{0,i}$ such that w_0 collides with some w_1 in $M_{1,j}$ in level s of Δ under $q_0 \oplus q_1$.

For $0 \leq i < k^2$ and $0 \leq j < t(s)$, we define

$$M(i,j) \stackrel{\text{def}}{=} \begin{cases} M_{0,j} & 0 \leq j < i, \\ (M_{0,j} \setminus C_{j,j-i}) \cup M_{1,j-i} & i \leq j < t(s-1), \\ M_{1,j-i} & t(s-1) \leq j < t(s-1)+i, \text{ and} \\ \emptyset & t(s-1)+i \leq j < t(s). \end{cases}$$

By their construction, the sets $M(i,j)$ are non-colliding in Δ under $q_0 \oplus q_1$. If we let $L_i \stackrel{\text{def}}{=} \bigcup_{i \leq j < t(s-1)} C_{j,j-i}$ for $0 \leq i < k^2$, then

$$\bigcup_{0\le j<t(s)} M(i,j) = (B_0 \setminus L_i) \cup B_1.$$

The $C_{i,j}$ are pairwise disjoint and contained in B_0. Thus, the L_i's are also pairwise disjoint and contained in B_0. Hence, by averaging there exists an i_0, $0 \le i_0 < k^2$, such that $|L_{i_0}| \le \frac{|B_0|}{k^2}$. We use this i_0 to determine the partial matching between the $M_{0,i}$ and the $M_{1,j}$.

More precisely, for all j with $0 \le j < t(s)$, we match the set $M_{0,j}$ with the set $M_{1,j-i_0}$ to obtain a new set $M_j \stackrel{\text{def}}{=} M(i_0, j)$ (here we assume $M_{0,i}$ and $M_{1,i}$ to be the empty set for $i < 0$ and $i \ge t(s-1)$). Thus, the new set M_j is obtained by removing the wires in $C_{j,j-i_0}$ from $M_{0,j}$, and merging the resulting set with $M_{1,j-i_0}$. We now show that this choice of M_j satisfies Properties (3) and (4). We have

$$\begin{aligned} B &\stackrel{\text{def}}{=} \bigcup_{0\le j<t(s)} M_j \\ &= (B_0 \setminus L_{i_0}) \cup B_1 \\ &\subset B_0 \cup B_1 \\ &\subset A_0 \cup A_1 \\ &= A. \end{aligned}$$

This establishes Property (3). Verifying Property (4) is also straightforward:

$$\begin{aligned} |B| &= |B_0| + |B_1| - |L_{i_0}| \\ &\ge |A_0| - \frac{(s-1)\cdot|A_0|}{k^2} + |A_1| - \frac{(s-1)\cdot|A_1|}{k^2} - |L_{i_0}| \\ &= (|A_0| + |A_1|)\left(1 - \frac{s-1}{k^2}\right) - |L_{i_0}| \\ &\ge |A| - \frac{(s-1)\cdot|A|}{k^2} - \frac{|B_0|}{k^2} \\ &\ge |A| - \frac{(s)\cdot|A|}{k^2}. \end{aligned}$$

To complete our proof, we have to construct a refinement q of p such that Properties (1) and (2) hold for q and the sets M_j. We do this by A_0-refining q_0 to some q'_0 and A_1-refining q_1 to some q'_1. Then $p_0 \supset_{A_0} q'_0$ and $p_1 \supset_{A_1} q'_1$, and by Lemma 2.3.1 the pattern $q \stackrel{\text{def}}{=} q'_0 \oplus q'_1$ is an A-refinement of p.

We refine q_0 to q'_0 in the following steps:

1. First change all pattern symbols $\mathcal{M}_i$ and $\mathcal{X}_{i,j}$ with $i \geq t(s-1)$ to $\mathcal{M}_{i+k^2}$ and $\mathcal{X}_{i+k^2,j}$, respectively.
2. Then change the pattern symbols of all wires in $C_{i,i-i_0}$ with $i_0 \leq i < t(s-1)$ to $\mathcal{X}_{i,j_0}$, where j_0 is chosen such that before this step only symbols $\mathcal{X}_{i,j}$ with $j < j_0$ appear in the pattern.

The steps for the refinement of q_1 to q'_1 are:

1′. First change all pattern symbols $\mathcal{M}_i$ and $\mathcal{X}_{i,j}$ with $i \geq t(s-1)$ to $\mathcal{M}_{i+k^2}$ and $\mathcal{X}_{i+k^2,j}$, respectively.

2′. Then change all pattern symbols $\mathcal{M}_i$ and $\mathcal{X}_{i,j}$ with $0 \leq i < t(s-1)$ to $\mathcal{M}_{i+i_0}$ and $\mathcal{X}_{i+i_0,j}$, respectively.

All refinement steps described above are order-preserving renamings and, thus, valid refinement steps. Steps 1 and 1′ remove all symbols $\mathcal{M}_i$ and $\mathcal{X}_{i,j}$ with $t(s-1) \leq i < t(s)$ from the patterns. Then Steps 2 and 2′ can be executed to perform the matching between the sets $M_{0,i}$ and $M_{1,j}$. Note that Steps 1 and 1′ are not really necessary since we can assume that the patterns q_0 and q_1 themselves have been constructed using the above refinement steps, and that, therefore, no symbols $\mathcal{M}_i$ and $\mathcal{X}_{i,j}$ with $i \geq t(s-1)$ exist in the pattern. However, in order to simplify our induction hypothesis, we have chosen not to make this assumption.

The pattern $q = q'_0 \oplus q'_1$ has been constructed such that the sets M_i are the $[\mathcal{M}_i]$-sets of q, so Property (1) is satisfied.

To see that Property (2) holds, note that $C_{i,j}$, the set of input wires of $M_{0,i}$ that collide with an input wire of $M_{1,j}$ in Γ_s under $q_0 \oplus q_1$, also contains the same colliding wires with respect to $q = q_0' \oplus q_1'$. The sets $M_{0,i}$ are non-colliding in Δ_0 under q_0' and, thus, also non-colliding in Δ under q. Similarly, the sets $M_{1,j}$ are non-colliding in Δ under q. Hence,

$$M_j = (M_{0,j} \setminus C_{j,j-i_0}) \cup M_{1,j-i_0}$$

is non-colliding in Δ under q.

□

Lemma 2.4.2 *Let Λ be an (l,s)-iterated reverse delta network with $l > 0$ and $s = \lg n$. Let W be the set of input wires of Λ, and let $n = |W| \geq 8$ be the number of input wires of Λ. Then there exists an input pattern p such that the following properties hold, where D is the [$\mathcal{M}_0$]-set of p:*

(1) Only the symbols $\mathcal{S}_0$, $\mathcal{M}_0$, and $\mathcal{L}_0$ occur in p,

(2) D is non-colliding in Λ under p, and

(3) $|D| \geq n/\lg^{4l} n$.

Proof: We will prove the lemma by induction over l, the number of consecutive reverse delta network in Λ.

Induction Start: $l = 0$

Choose $D = W$ and p such that $p(w) = \mathcal{M}_0$ for all w in W.

Induction Step: $l > 0$

A (l,s)-iterated reverse delta network Λ consists of an $(l-1,s)$-iterated reverse delta network Λ_0 followed by a single s-level reverse delta network Δ, or, formally, $\Lambda \in \Lambda_0 \otimes \Delta$.

By the induction hypothesis there exists a pattern p' such that the following properties hold, where D' is the $[\mathcal{M}_0]$-set of p':

- Only the symbols $\mathcal{S}_0$, $\mathcal{M}_0$, and $\mathcal{L}_0$ occur in p',
- D' is non-colliding in Λ_0 under p', and
- $|D'| \geq n/\lg^{4(l-1)} n$.

Then the input pattern $q' \stackrel{\text{def}}{=} \Lambda_0(p')$ for Δ contains only the symbols $\mathcal{S}_0$, $\mathcal{M}_0$, and $\mathcal{L}_0$. The $[\mathcal{M}_0]$-set B' of q' has size $|B'| = |D'| \geq n/\lg^{4(l-1)} n$.

We can now apply Lemma 2.4.1 with Δ, q', and $s = k = \lg n$. By the lemma, there exists an input pattern q'' with $q' \supset_{B'} q''$ and $t(\lg n) = 2\lg^3 n$ disjoint sets $M_0, \ldots, M_{t(\lg n)-1}$ of input wires of Δ such that

- every M_i is the $[\mathcal{M}_i]$-set of q'',
- every M_i is non-colliding in Δ under q'',
- $B'' \subset B'$, and
- $|B''| \geq |B'| - \frac{|B'| \cdot \lg n}{\lg^2 n} \geq \frac{n}{\lg^{4(l-1)} n}(1 - \frac{1}{\lg n})$,

where $B'' \stackrel{\text{def}}{=} \bigcup_{0 \leq i < t(\lg n)} M_i$.

By averaging, there exists a set M_{i_0}, $0 \leq i_0 < 2\lg^3 n$, of size at least

$$\frac{n}{2\lg^{4l-1} n}\left(1 - \frac{1}{\lg n}\right) \geq \frac{n}{\lg^{4l} n},$$

where the last inequality follows from the fact that $\frac{1}{2}(1 - 1/\lg n) \geq 1/\lg n$ for all $n \geq 8$.

By Lemma 2.3.3, there exists an input pattern p'' for Λ with $p' \supset_{D'} p''$ such that $q'' = \Lambda_0(p'')$. The set M_{i_0} is non-colliding in Δ, hence the $[\mathcal{M}_{i_0}]$-set D of p'' is non-colliding in $\Lambda \in \Lambda_0 \otimes \Delta$ under p''.

Then, by Lemma 2.3.4, there exists an input pattern p such that

- only the symbols $\mathcal{S}_0$, $\mathcal{M}_0$, and $\mathcal{L}_0$ occur in p, and

- D is non-colliding in Λ under p.

Furthermore, we have $|D| = |M_{i_0}| \geq n/\lg^{4l} n$. This concludes the induction step.

□

Due to Definition 2.3.9, this directly implies the following lemma.

Lemma 2.4.3 *Let Λ be an (l,s)-iterated reverse delta network with $l > 0$ and $s = \lg n$. Then Λ has an incomparable set of size at least $n/\lg^{4l} n$.*

Corollary 2.4.0.1 *All n-input sorting networks with iterated delta topology have depth $\Omega\left(\frac{\lg^2 n}{\lg\lg n}\right)$.*

Proof: Let Λ be an $(l, \lg n)$-iterated reverse delta network with $l < \frac{\lg n}{4 \lg\lg n}$. By Lemma 2.4.3, Λ has an incomparable set of size at least

$$\frac{n}{\lg^{4l} n} > \frac{n}{\lg^{\left(\frac{\lg n}{\lg\lg n}\right)} n} = 1.$$

Thus, Λ cannot be a sorting network. Note that the constant $1/4$ obtained in this proof can be improved to $1/(2+\epsilon)$ by a sharper analysis in Lemmas 2.4.1 and 2.4.2.

□

2.5 Bounds for Shuffle-Unshuffle Networks

In this section, we extend the ideas of the previous section to the case of arbitrary shuffle-unshuffle sorting networks. We first show in Subsection 2.5.1 that a large incomparable set can be effectively maintained over the levels of any shuffle-unshuffle network with sufficiently small overlap. The main result of this section is Lemma 2.5.2, which bounds the decrease in the size of the incomparable set that can occur in any 2^d-input shuffle-unshuffle network with span $s \leq d$ and overlap

r. This lemma is then used in Subsection 2.5.2 to establish our lower bound for arbitrary shuffle-unshuffle sorting networks.

2.5.1 Networks with Small Overlap

The actual argument addressing the size of the incomparable set in a shuffle-unshuffle network with small overlap is contained in the proof of Lemma 2.5.1, and is described with respect to a more general class of networks, called (d, s, r)*-hypercubic* networks.

We now give an inductive definition of the class of (d, s, r)*-hypercubic* networks, which properly contains the class of 2^d-input shuffle-unshuffle networks with span $s \leq d$ and overlap r. Note that the 2^d output wires of a (d, s, r)-hypercubic network are partitioned into 2^{d-r} *output groups* of size 2^r.

Definition 2.5.1 *For $r \leq s \leq d$, a 2^d-input comparator network Δ is called a* (d, s, r)-hypercubic *network if:*

(a) $s - r = 0$, Δ is a network containing no comparators at all (i.e., the 2^d input wires are directly connected to the 2^d output wires), and the output wires of Δ have been partitioned into 2^{d-r} output groups of size 2^r, or

(b) $s - r > 0$ and Δ is an element of $(\Delta_0 \oplus \Delta_1) \otimes \Lambda$, where

- *Δ_0 and Δ_1 are $(d-1, s-1, r)$-hypercubic networks, and*
- *Λ is the parallel composition of 2^{d-r-1} disjoint 2^{r+1}-input comparator networks Λ_i, $0 \leq i < 2^{d-r-1}$, of arbitrary size and depth, such that: (i) the 2^{r+1} input wires of each network Λ_i are connected to one output group of size 2^r of Δ_0 and one output group of size 2^r of Δ_1, and (ii) the 2^{r+1} output wires of each network Λ_i are partitioned to form two of the 2^{d-r} output groups of network Δ.*

The following Lemma 2.5.1 is actually a generalization of Lemma 2.4.1, and the proof also has a very similar structure. However, a number of often subtle changes are needed to establish the result.

Lemma 2.5.1 *Let Δ be a (d,s,r)-hypercubic network with $r \leq s \leq d$, and p be an input pattern for Δ such that only the pattern symbols $\mathcal{S}_0$, $\mathcal{L}_0$, and $\mathcal{M}_0$ occur in p. Let A be the $[\mathcal{M}_0]$-set of p, and k be any positive integer. Then there exists an input pattern q with $p \supset_A q$ and $t(s) \stackrel{def}{=} 2^r \cdot k^3 + (s-r) \cdot 2^r \cdot k^2$ sets M_i, $0 \leq i < t(s)$, of input wires such that the following properties hold, where $B \stackrel{def}{=} \bigcup_{0 \leq i < t(s)} M_i$:*

(1) Every M_i is the $[\mathcal{M}_i]$-set of q.

(2) Every M_i is non-colliding in Δ under q.

(3) $B \subset A$.

(4) $|B| \geq |A| - \frac{(s-r)\cdot|A|}{k^2}$.

(5) No two elements of any $[\mathcal{M}_i]$-set of $\Delta(q)$ are located in the same output group of Δ.

Proof: The proof is by induction on $s - r$.

Base Case: $s - r = 0$

In this case the network Δ does not contain any comparator elements. We define the sets M_i, $0 \leq i < t(0)$, by partitioning A into $2^r \cdot k^3$ sets such that no two elements in any set are located in the same output group. (Each output group has size $2^r \leq 2^r \cdot k^3$, so this is clearly possible.) If we define q as the pattern obtained from p by relabeling each wire in set M_i with $\mathcal{M}_i$, for $0 \leq i < t(0)$, then Properties (1) to (5) are satisfied.

Induction Step: $s - r > 0$

A (d,s,r)-hypercubic network consists of two $(d-1, s-1, r)$-hypercubic networks Δ_0 and Δ_1, and a network Λ satisfying the conditions of Definition 2.5.1.

The input wires W of Δ can be partitioned into the sets W_0 and W_1 of input wires of Δ_0 and Δ_1, respectively. Let $p_0 \stackrel{\text{def}}{=} p_{|W_0}$ and $p_1 \stackrel{\text{def}}{=} p_{|W_1}$. Then $A_0 \stackrel{\text{def}}{=} A \cap W_0$ is the $[\mathcal{M}_0]$-set of p_0 and $A_1 \stackrel{\text{def}}{=} A \cap W_1$ is the $[\mathcal{M}_0]$-set of p_1.

Applying the induction hypothesis to Δ_0, p_0, and A_0, we can infer the existence of an input pattern q_0 with $p_0 \supset_{A_0} q_0$, and of $t(s-1)$ disjoint sets $M_{0,i}$, $0 \le i < t(s-1)$, such that

- every $M_{0,i}$ is the $[\mathcal{M}_i]$-set of q_0,
- every $M_{0,i}$ is non-colliding in Δ_0 under q_0,
- $B_0 \subset A_0$,
- $|B_0| \ge |A_0| - \frac{(s-r-1)\cdot|A_0|}{k^2}$, and
- no two elements of any $[\mathcal{M}_i]$-set of $\Delta_0(q_0)$ are located in the same output group of Δ_0,

where $B_0 \stackrel{\text{def}}{=} \bigcup_{0 \le i < t(s-1)} M_{0,i}$.

Correspondingly, for Δ_1, p_1, and A_1, we get an input pattern q_1, disjoint sets $M_{1,i}$, $0 \le i < t(s-1)$, and a set B_1, with the same properties.

We will now construct the sets M_i, $0 \le i < t(s)$, by combining the sets $M_{0,i}$ of Δ_0 with the sets $M_{1,j}$ of Δ_1, according to some partial matching to be determined in the following.

Because no $[\mathcal{M}_i]$-set of $\Delta_0(q_0)$ (resp., $\Delta_1(q_1)$) contains any two elements that are located in the same output group of Δ_0 (resp., Δ_1), no element of any set $M_{0,i}$ (resp., $M_{1,i}$) can collide with any other element of the same set in Δ.

Also, due to the topology of a (d, s, r)-hypercubic network, no element of a set $M_{0,i}$ can collide in $\Delta_0 \oplus \Delta_1$ with any element of a set $M_{1,j}$. By Lemma 2.3.2, we can determine for each w in a set $M_{0,i}$ (resp., $M_{1,j}$) the output wire w' of Δ_0 (resp., Δ_1) that receives the value $\pi(w)$ under all π in $q_0[V]$ (resp., $q_1[V]$). Thus, for any such w we can determine the subnetwork Λ_ν (where ν is some function f of w) of Λ that will receive $\pi(w)$ as an input value under all π in $q_0[V]$ (resp., $q_1[V]$).

For $0 \leq i, j < t(s-1)$, we define $C_{i,j}$ as the set of all wires w_0 in $M_{0,i}$ such that $f(w_0) = f(w_1)$ holds for some w_1 in $M_{1,j}$. Note that the $C_{i,j}$'s are not pairwise disjoint. However, since each subnetwork Λ_ν receives only 2^r input values from Δ_1, every element w_0 in $M_{0,i}$ is contained in at most 2^r sets $C_{i,j}$. Also, each $C_{i,j}$ contains all wires in $M_{0,i}$ that can collide in Λ with some wire in $M_{1,j}$.

For $0 \leq i < 2^r \cdot k^2$ and $0 \leq j < t(s)$, we define

$$M(i,j) \stackrel{\text{def}}{=} \begin{cases} M_{0,j} & 0 \leq j < i, \\ (M_{0,j} \setminus C_{j,j-i}) \cup M_{1,j-i} & i \leq j < t(s-1), \\ M_{1,j-i} & t(s-1) \leq j < t(s-1)+i, \text{ and} \\ \emptyset & t(s-1)+i \leq j < t(s). \end{cases}$$

By their construction, the sets $M(i,j)$ are non-colliding in Δ under $q_0 \oplus q_1$. If we let $L_i \stackrel{\text{def}}{=} \bigcup_{i \leq j < t(s-1)} C_{j,j-i}$ for $0 \leq i < 2^r \cdot k^2$, then

$$\bigcup_{0 \leq j < t(s)} M(i,j) = (B_0 \setminus L_i) \cup B_1.$$

Since every element of B_0 can occur at most 2^r times in the sets $C_{i,j}$, every element of B_0 can occur at most 2^r times in the sets L_i. Hence, by averaging there exists an i_0, $0 \leq i_0 < k^2 \cdot 2^r$, such that $|L_{i_0}| \leq \frac{|B_0|}{k^2}$. We use this i_0 to determine the partial matching between the $M_{0,i}$'s and the $M_{1,j}$'s.

More precisely, for all j such that $0 \leq j < t(s)$, we match the set $M_{0,j}$ with the set $M_{1,j-i_0}$ to obtain a new set $M_j \stackrel{\text{def}}{=} M(i_0, j)$. (Here we assume $M_{0,i}$ and $M_{1,i}$ to be the empty set for $i < 0$ and $i \geq t(s-1)$.) Thus, the new set M_j is obtained by removing the wires in $C_{j,j-i_0}$ from $M_{0,j}$, and merging the resulting set with $M_{1,j-i_0}$. We now show that this choice of M_j satisfies

Properties (3) and (4). We have

$$
\begin{aligned}
B &\stackrel{\text{def}}{=} \bigcup_{0\le j<t(s)} M_j \\
&= (B_0 \setminus L_{i_0}) \cup B_1 \\
&\subset B_0 \cup B_1 \\
&\subset A_0 \cup A_1 \\
&= A.
\end{aligned}
$$

This establishes Property (3). Verifying Property (4) is also straightforward:

$$
\begin{aligned}
|B| &= |B_0| + |B_1| - |L_{i_0}| \\
&\ge |A_0| - \frac{(s-r-1)\cdot|A_0|}{k^2} + |A_1| - \frac{(s-r-1)\cdot|A_1|}{k^2} - |L_{i_0}| \\
&= (|A_0| + |A_1|)\left(1 - \frac{(s-r-1)}{k^2}\right) - |L_{i_0}| \\
&\ge |A| - \frac{(s-r-1)\cdot|A|}{k^2} - \frac{|B_0|}{k^2} \\
&\ge |A| - \frac{(s-r)\cdot|A|}{k^2}
\end{aligned}
$$

To complete our proof, we construct a refinement q of p such that Properties (1), (2), and (5) hold for q and the sets M_j. We do this by A_0-refining q_0 to some q_0' and A_1-refining q_1 to some q_1'. Then $p_0 \supset_{A_0} q_0'$ and $p_1 \supset_{A_1} q_1'$, and by Lemma 2.3.1 the pattern $q \stackrel{\text{def}}{=} q_0' \oplus q_1'$ is an A-refinement of p.

We refine q_0 to q_0' in the following steps:

1. First change all pattern symbols $\mathcal{M}_i$ and $\mathcal{X}_{i,j}$ with $i \ge t(s-1)$ to $\mathcal{M}_{i+2^r\cdot k^2}$ and $\mathcal{X}_{i+2^r\cdot k^2,j}$, respectively.
2. Then change the pattern symbols of all wires in $C_{i,i-i_0}$ with $i_0 \le i < t(s-1)$ to $\mathcal{X}_{i,j_0}$, where j_0 is chosen such that before this step only symbols $\mathcal{X}_{i,j}$ with $j < j_0$ appear in the pattern.

The steps for the refinement of q_1 to q_1' are:

1′. First change all pattern symbols $\mathcal{M}_i$ and $\mathcal{X}_{i,j}$ with $i \geq t(s-1)$ to $\mathcal{M}_{i+2^r \cdot k^2}$ and $\mathcal{X}_{i+2^r \cdot k^2, j}$, respectively.

2′. Then change all pattern symbols $\mathcal{M}_i$ and $\mathcal{X}_{i,j}$ with $0 \leq i < t(s-1)$ to $\mathcal{M}_{i+i_0}$ and $\mathcal{X}_{i+i_0,j}$, respectively.

All refinement steps described above are order-preserving renamings and, thus, valid refinement steps. Steps 1 and 1′ remove all symbols $\mathcal{M}_i$ and $\mathcal{X}_{i,j}$ with $t(s-1) \leq i < t(s)$ from the patterns. Then Steps 2 and 2′ can be executed to perform the matching between the sets $M_{0,i}$ and $M_{1,j}$. Note that Steps 1 and 1′ are not really necessary since we can assume that the patterns q_0 and q_1 themselves have been constructed using the above refinement steps, and hence that no symbols $\mathcal{M}_i$ and $\mathcal{X}_{i,j}$ with $i \geq t(s-1)$ exist in the pattern. However, in order to simplify our induction hypothesis, we have chosen not to make this assumption.

The pattern $q = q_0' \oplus q_1'$ has been constructed such that the sets M_i are the $[\mathcal{M}_i]$-sets of q, so Property (1) is satisfied.

To see that Property (2) holds, note that the set $C_{i,j}$, which contains all input wires of $M_{0,i}$ that can collide with an input wire of $M_{1,j}$ in Λ under $q_0 \oplus q_1$, also contains the same colliding wires with respect to $q = q_0' \oplus q_1'$. The sets $M_{0,i}$ are non-colliding in Δ_0 under q_0' and, thus, also non-colliding in Δ under q. Similarly, the sets $M_{1,j}$ are non-colliding in Δ under q. Hence,

$$M_j = (M_{0,j} \setminus C_{j,j-i_0}) \cup M_{1,j-i_0}$$

is non-colliding in Δ under q.

Finally, due to the definition of the sets $C_{i,j}$ that were removed from the matched sets, no two elements of any $[\mathcal{M}_i]$-set of $\Delta(q)$ are in the same output group of Δ. This establishes Property (5).

□

Lemma 2.5.2 *Let Δ be a 2^d-input shuffle-unshuffle network with span $s \le d$ and overlap r, and let Λ be an arbitrary comparator network with an incomparable set of size ν. Then any network in $\Lambda \otimes \Delta$ has an incomparable set of size $\nu' \ge \nu/(s^4 \cdot 2^r)$.*

Proof: According to Definition 2.3.9, there exists an input pattern p_0 such that some $[\mathcal{M}_{i_0}]$-set C of p_0 is of size ν and is non-colliding in Λ under p_0. By Lemma 2.3.4, we can assume that $i_0 = 0$, and that p_0 contains only the symbols $\mathcal{S}_0$, $\mathcal{M}_0$, and $\mathcal{L}_0$.

Every 2^d-input shuffle-unshuffle network with span $s \le d$ and overlap r is equivalent to a (d, s, r)-hypercubic network. Hence, we can apply Lemma 2.5.1 to Δ. Let $k = s$, $p = \Lambda(p_0)$, and A be the $[\mathcal{M}_0]$-set of p. Then by Lemma 2.5.1, there exists an input pattern q with $p \supset_A q$ and $t(s) \le 2s^3 \cdot 2^r$ disjoint sets M_i, $0 \le i < t(s)$ of input wires of Δ such that

- every M_i is the $[\mathcal{M}_i]$-set of q,
- every M_i is non-colliding in Δ under q,
- $B \subset A$, and
- $|B| \ge \nu \cdot (1 - 1/s)$,

where $B \stackrel{\text{def}}{=} \bigcup_{0 \le i < t(s)} M_i$. By averaging, there exists a set M_{j_0}, $0 \le j_0 < t(s)$, of size at least

$$\frac{|B|}{2s^3 \cdot 2^r} \ge \frac{\nu}{s^4 \cdot 2^r},$$

where the inequality follows from the fact that $\frac{1}{2}(1-1/s) \ge 1/s$ for $s \ge 3$. (For $s < 3$, the claim follows from $\nu' \ge \nu/2^s$.) By Lemma 2.3.3, there exists an input pattern q_0 with $p_0 \supset_C q_0$ such that $q = \Lambda(q_0)$ and the $[\mathcal{M}_{j_0}]$-set of q_0 is non-colliding in $\Lambda \otimes \Delta$ under q_0. Since $q = \Lambda(q_0)$, the $[\mathcal{M}_{j_0}]$-set of q_0 also contains at least $\nu/(s^4 \cdot 2^r)$ elements.

□

The following lemma can be established by partitioning a shuffle-unshuffle network of overlap r and depth ℓ into $\lceil \ell/d \rceil$ consecutive shuffle-unshuffle networks of overlap r and depth at most d, and applying Lemma 2.5.2 to each of the networks.

Lemma 2.5.3 *Let Λ be an n-input shuffle-unshuffle network with depth ℓ and overlap $r \leq d = \lg n$. Then Λ has an incomparable set of size at least*

$$\frac{n}{(d^4 \cdot 2^r)^{\lceil \ell/d \rceil}}.$$

Lemma 2.5.3 immediately implies the following lower bound for shuffle-unshuffle networks with bounded overlap. Note that for the special case $r = 0$, we obtain the result in Section 2.4. However, if the overlap is $\Theta(d)$, we only get the trivial $\Omega(\lg n)$ lower bound.

Theorem 2.5.1 *Any n-input shuffle-unshuffle sorting network with overlap r has depth $\Omega\left(\frac{\lg^2 n}{r + \lg\lg n}\right)$.*

2.5.2 Networks with Arbitrary Overlap

In this subsection we establish the main result of this chapter, a lower bound on the depth of arbitrary shuffle-unshuffle sorting networks. In order to prove the result, we need one more lemma. Informally, Lemma 2.5.4 below states that we can maintain a fairly large incomparable set over the levels of any shuffle-unshuffle network of span at most d. The proof of the lemma is based on the idea that any shuffle-unshuffle network with depth ℓ either has a small overlap relative to ℓ, or can be (recursively) partitioned into several consecutive networks satisfying this property. In the first case, we can use Lemma 2.5.2 to bound the size of the incomparable set. The second case is handled by induction.

Lemma 2.5.4 *Let Δ be a shuffle-unshuffle network with depth ℓ and span $s \leq d$, let $\alpha(\ell, s) \stackrel{def}{=} (\ell - s/2)/(\lg s / \lg\lg s)$, and let Λ be an arbitrary comparator network* *with an incomparable set of size ν. Then any network in $\Lambda \otimes \Delta$ has an incomparable set of size ν', where*

$$\frac{\nu}{\nu'} \geq s^4 \cdot 2^{9 \cdot \alpha(\ell, s)}.$$

Proof: The proof is by induction on the depth ℓ of the network.

Base Case: $\ell \leq 2^{16}$

Using $s \leq \ell \leq 2^{16}$, we obtain $\lg s / \lg\lg s \leq 4$ and

$$9 \cdot \alpha(\ell, s) \ \geq \ \frac{9 \cdot \ell/2}{4} \ \geq \ \ell \ \geq r.$$

Then the claim follows by a simple application of Lemma 2.5.2.

Induction Step: $\ell > 2^{16}$

For the induction step, we assume a shuffle-unshuffle network Δ with depth ℓ, overlap r, and span $s \leq d$. Now suppose that $r \leq 9 \cdot \alpha(\ell, s)$. In this case, the claim follows by a simple application of Lemma 2.5.2.

Hence, in the following we assume that

$$r \ > \ 9 \cdot \alpha(\ell, s) \ \geq \ \frac{9s}{2 \lg s / \lg\lg s}. \tag{2.1}$$

Note that $s \geq r > 9 \cdot \alpha(\ell, s)$ and $\ell > 2^{16}$ imply $s > 2^{16}$ and $\lg\lg s / \lg s < 1/4$.

Due to the definition of overlap, there exist shuffle-unshuffle networks Δ_i, $0 \leq i < 2$, with depth ℓ_i and span s_i, such that Δ belongs to $\Delta_0 \otimes \Delta_1$, $\ell_0 + \ell_1 = \ell$, and $s_0 + s_1 = s + r$. By applying the induction hypothesis first to Λ and Δ_0 , and then to $\Lambda \otimes \Delta_0$ and Δ_1, we obtain

$$\begin{aligned} \frac{\nu}{\nu'} \quad &\leq \quad s_0^4 \cdot 2^{9 \cdot \alpha(\ell_0, s_0)} \cdot s_1^4 \cdot 2^{9 \cdot \alpha(\ell_1, s_1)} \\ &= \quad s_0^4 \cdot s_1^4 \cdot 2^{9x}, \end{aligned}$$

where $x \stackrel{\text{def}}{=} \alpha(\ell_0, s_0) + \alpha(\ell_1, s_1)$. Using $\min\{s_0, s_1\} \geq r$, $\max\{s_0, s_1\} \leq s$, and Equation (2.1) we obtain

$$\min\left\{\frac{\lg s_0}{\lg\lg s_0}, \frac{\lg s_1}{\lg\lg s_1}\right\} \quad \geq \quad \frac{\lg r}{\lg\lg s}$$

$$
\begin{aligned}
&\geq \frac{1}{\lg\lg s} \cdot \lg\left(\frac{9s}{2\lg s/\lg\lg s}\right) \\
&\geq \frac{1}{\lg\lg s} \cdot \lg\left(\frac{s}{\lg s}\right) \\
&= \frac{\lg s}{\lg\lg s} \cdot \left(1 - \frac{\lg\lg s}{\lg s}\right).
\end{aligned}
$$

Using this bound, and the fact that $1/(1-\epsilon) \leq 1+2\epsilon$ holds for $\epsilon = \lg\lg s/\lg s < 1/2$, we obtain

$$
\begin{aligned}
x &\leq \left(\frac{\ell_0 - s_0/2}{\lg s/\lg\lg s} + \frac{\ell_1 - s_1/2}{\lg s/\lg\lg s}\right)\left(\frac{1}{1-\lg\lg s/\lg s}\right) \\
&\leq \left(\frac{\ell_0 - s_0/2}{\lg s/\lg\lg s} + \frac{\ell_1 - s_1/2}{\lg s/\lg\lg s}\right)(1 + 2\lg\lg s/\lg s) \\
&= \frac{\ell_0 - s_0/2}{\lg s/\lg\lg s} + \frac{\ell_1 - s_1/2}{\lg s/\lg\lg s} + (\ell_0 - s_0/2 + \ell_1 - s_1/2) \cdot 2\left(\frac{\lg\lg s}{\lg s}\right)^2 \\
&= \frac{\ell - s/2 - r/2}{\lg s/\lg\lg s} + (\ell - s/2 - r/2) \cdot 2\left(\frac{\lg\lg s}{\lg s}\right)^2 \\
&\leq \alpha(\ell, s) - \frac{r}{2\lg s/\lg\lg s} + (\ell - s/2) \cdot 2\left(\frac{\lg\lg s}{\lg s}\right)^2.
\end{aligned}
$$

Note that Equation (2.1) implies

$$
\ell - s/2 < \frac{r\lg s}{9\lg\lg s},
$$

and hence

$$
\begin{aligned}
x &\leq \alpha(\ell, s) - \frac{r}{2\lg s/\lg\lg s} + \frac{2r}{9\lg s/\lg\lg s} \\
&= \alpha(\ell, s) - \frac{5r}{18\lg s/\lg\lg s} \\
&\leq \alpha(\ell, s) - \frac{5s}{4(\lg s/\lg\lg s)^2} \\
&\leq \alpha(\ell, s) - \frac{4\lg s}{9},
\end{aligned}
$$

where the last two inequalities follow from Equation (2.1) and $s > 2^{16}$, respectively. Using $\max\{s_0, s_1\} \leq s$ we obtain

$$
\frac{\nu}{\nu'} \leq s_0^4 \cdot s_1^4 \cdot 2^{9x}
$$

$$
\begin{aligned}
&\leq s_0^4 \cdot s_1^4 \cdot 2^{9\cdot\alpha(\ell,s)-4\lg s} \\
&\leq s^4 \cdot 2^{4\lg s} \cdot 2^{9\cdot\alpha(\ell,s)-4\lg s} \\
&= s^4 \cdot 2^{9\cdot\alpha(\ell,s)}.
\end{aligned}
$$

□

Theorem 2.5.2 *Any shuffle-unshuffle sorting network has depth* $\Omega\left(\frac{\lg n \lg\lg n}{\lg\lg\lg n}\right)$.

Proof: Let Δ be an n-input shuffle-unshuffle network of depth ℓ, $n = 2^d$. We partition Δ into $k \leq \lceil \ell/d \rceil$ consecutive shuffle-unshuffle networks Δ_i, $0 \leq i < k$, with depth ℓ_i and span d. This can be done by defining Δ_0 as the shortest prefix of the levels of the network Δ with span d, Δ_1 as the shortest prefix of $\Delta - \Delta_0$ with span d, and so on. (At the end, we may have to add some additional levels to the network in order to get a span of exactly d for Δ_{k-1}. Adding such additional levels to the network can certainly not increase the size of the largest incomparable set.)

Let Λ be a network containing no comparator elements at all. Clearly, Δ belongs to $\Lambda \otimes \Delta$, and Λ has an incomparable set of size n. We now apply Lemma 2.5.4 once for each network Δ_i, $0 \leq i < k$. It follows that there exists an incomparable set of size n' in Δ, such that

$$\frac{n}{n'} = \prod_{0 \leq i < k} d^4 \cdot 2^{\left(\frac{9(\ell_i - d/2)}{\lg d/\lg\lg d}\right)} \leq 2^{\left(\frac{9\ell}{\lg d/\lg\lg d}\right)},$$

for d sufficiently large. Hence, if $\ell < 9 \cdot d \lg d/\lg\lg d$, then $n' > 1$, and it follows that Δ cannot be a sorting network.

□

2.6 Extensions and Limitations of the Proof Technique

This section discusses some extensions and limitations of our proof technique. In the first subsection, we describe how our bounds can be generalized to certain classes of non-oblivious sorting algorithms on the hypercube. Subsection 2.6.2 contains some extensions to multi-dimensional meshes. Finally, we explain why our results cannot be extended to the average or randomized case.

2.6.1 Non-Oblivious Sorting Algorithms

The lower bounds for shuffle-based and shuffle-unshuffle sorting networks can be extended to certain restricted classes of non-oblivious sorting algorithms on hypercubic machines. Recall that in the lower bound arguments, it was never assumed that the labeling of the circuit elements with $\{+, -, 0, 1\}$ was fixed beforehand. Instead, in every level, we can allow the network to choose this labeling in an arbitrary, deterministic fashion. (That is, the label of each circuit element may be computed as an arbitrary function of the outcomes of all comparisons previously made throughout the entire network.)

Our lower bounds also extend to the case where a processor can temporarily hold more than one element during the computation, provided that elements cannot be copied. (Here, we assume that a processor can send an element to a neighboring processor, without receiving another element in return.) To prove this claim, we observe that no processor can accumulate more than $\lg^2 n$ elements during any computation of length $\lg^2 n$. We can then model each processor by a "sub-hypercube" of $\lg^2 n$ "sub-processors", each containing at most one element. We assume that in a single step, an arbitrary amount of computation can be performed within each "sub-hypercube"; this gives $\Theta(\lg \lg n)$ additional dimensions in the network, which can be "handled" by adding $\Theta(\lg \lg n)$ to the overlap of the networks occurring in the lower bound argument.

These observations lead us to the following class of non-oblivious sorting algorithms on the hypercube that is covered by our lower bounds:

(1) The algorithm has to be deterministic, normal (resp., ascend/descend), and

comparison-based. (That is, the only way of accessing the value of an element is by means of a comparison with another element.)

(2) No copies of elements can be made. (It is unclear whether our techniques can be extended to a model where copying of elements is allowed.)

(3) The sequence of shuffles and unshuffles in the normal algorithm is oblivious. (In fact, it is clear that this sequence only needs to be fixed $\Theta(\lg n)$ steps in advance, and we believe that it should be possible to remove this restriction.)

(4) Initially, each processor holds a single element.

(5) In a single step, each processor can perform an arbitrary amount of internal computation, and can send one element, plus an arbitrary amount of auxiliary information, to one of its neighbors.

While this class covers a fairly wide range of sorting algorithms, it does unfortunately not include the *Sharesort* algorithm of Cypher and Plaxton [26], which makes copies of some of the elements.

2.6.2 Multi-Dimensional Meshes

We can also extend our lower bounds to some restricted classes of sorting algorithms on multi-dimensional meshes. In [121], Wanka describes the following natural extension of the class of ascend algorithms to multi-dimensional meshes. In an ascend algorithm on a d-dimensional mesh of side length m, the dimensions are visited in strictly ascending order. Whenever we visit a dimension, we perform m steps of communication across this dimension. Thus, in a single visit to a dimension, an algorithm could completely sort the elements in each linear array along that dimension. Note that this class of algorithms corresponds to the class of sorting networks built from m-input comparator gates, where consecutive levels of the network are connected by an m-way unshuffle permutation (as defined in the register model of a comparator network).

An example of an ascend algorithm on the two-dimensional mesh is the *Shearsort* algorithm [100, 101], which alternatingly sorts along the rows and along the columns. Recently, Corbett and Scherson [23] and Wanka [121] have described two different generalizations of this algorithm to meshes of arbitrary dimension. Both of the algorithms can be implemented as ascend algorithms, and they achieve a running time of $O(d^2 m \lg m)$ on the d-dimensional mesh of sidelength m.

Using the techniques in this chapter, we can show an $\Omega(d^2 m \lg m / \lg(dm))$ lower bound for the class of ascend sorting algorithms on multi-dimensional meshes (under the conditions stated in the previous subsection, that is, the algorithms have to be comparison-based and no copying of elements is allowed). For meshes with non-constant dimension, this implies that no ascend algorithm can achieve an asymptotically optimal running time. For constant d and sufficiently large m, we can show that any ascend algorithm requires more than $(d-1) \cdot dm$ steps.

Similarly, we can define natural extensions of the classes of normal algorithms, and normal algorithms with overlap, to multi-dimensional meshes. For normal algorithms, we obtain a lower bound of $\Omega(dm \lg d / \lg \lg d)$.

2.6.3 Average Case and Randomized Algorithms

Our lower bounds do not apply to probabilistic sorting networks, that is, networks that sort the vast majority of input permutations, but are not sorting networks in the strict sense. In fact, Leighton and Plaxton [70] have designed a shuffle-unshuffle comparator network of depth $O(\lg n)$ that sorts all but a super-polynomially small fraction of the inputs. Their techniques can also be used to construct a shuffle-based network of depth $O(\lg n \lg \lg n)$ that sorts all but a polynomially small fraction of the inputs.

Similarly, we cannot hope to extend our lower bounds to "randomized" sorting networks, which may contain additional "randomizing" circuit elements that interchange the input values with probability $1/2$, and leave them unchanged otherwise. In [70], Leighton and Plaxton show how to construct a randomized shuffle-unshuffle network of depth $O(\lg n)$ that sorts every input permutation with high probability. This new element can also be used to construct a shuffle-based randomized sorter of depth $O(\lg n \lg\lg n)$.

2.7 Open Questions

In this chapter, we have established lower bounds on the depth of shuffle-based and shuffle-unshuffle sorting networks. Our techniques also apply to certain restricted classes of non-oblivious sorting algorithms on hypercubes and multi-dimensional meshes. A gap remains between our lower bounds and the best upper bounds known, and it would certainly be an interesting improvement to narrow or close this gap.

An important open question is whether we can extend our lower bounds to more general classes of non-oblivious sorting algorithms on the hypercube. Of particular interest in this respect would be the class of normal comparison-based sorting algorithms, or any other natural class of algorithms that includes the *Sharesort* algorithm of Cypher and Plaxton [26].

Another possible direction for future research would be to consider other restricted classes of sorting networks. As a natural extension of the shuffle-unshuffle networks, we could consider the class of sorting networks whose structure corresponds to the class of "leveled" algorithms on the hypercube, where in each step communication only occurs across a single dimension, but the sequence of dimensions can be arbitrary. (Note that this class of algorithms cannot be emulated with constant slowdown on any of the bounded-degree variants of the hypercube.)

Other classes of interest would be sorting networks based on a single permutation, or periodic sorting networks [28, 48, 60].

Finally, it is an open problem whether our techniques can be applied to obtain lower bounds for shuffle-based or shuffle-unshuffle selection networks.

Chapter 3

Lower Bounds for Shellsort

In this chapter we establish lower bounds on the worst-case complexity of *Shellsort* networks and algorithms. In particular, we give a fairly simple proof of an $\Omega(n \lg^2 n/(\lg\lg n)^2)$ lower bound for the size of Shellsort sorting networks, for arbitrary increment sequences. We also show an identical lower bound for the running time of non-oblivious Shellsort algorithms. Our lower bounds establish an almost tight trade-off between the running time of a Shellsort algorithm and the length of the underlying increment sequence.

3.1 Introduction

Shellsort is a classical sorting algorithm introduced by Shell in 1959 [104]. The algorithm is based on a sequence $H = h_0, \ldots, h_{m-1}$ of positive integers called an *increment sequence*. An input file $A = A[0], \ldots, A[n-1]$ of elements is sorted by performing an h_j-sort for every increment h_j in H, starting with h_{m-1} and going down to h_0. Every h_j-sort partitions the positions of the input array into congruence classes modulo h_j, and then performs Insertion Sort on each of these classes. It is not difficult to see that at least one of the h_j's must be equal to 1 in order for the algorithm to sort all input files properly. Furthermore, once some increment equal to 1 has been processed, the file will certainly be sorted. Hence, we may assume without loss of generality that $h_0 = 1$ and $h_j > 1$ for all $j > 0$.

The running time of Shellsort varies heavily depending on the choice of the increment sequence H. Most practical Shellsort algorithms set H to the prefix of a single, monotonically increasing infinite sequence of integers, using only the increments that are less than n. Shellsort algorithms based on such increment sequences are called *uniform*. In a *nonuniform* Shellsort algorithm, H may depend on the input size n in an arbitrary fashion.

A general analysis of the running time of Shellsort is difficult because of the vast number of possible increment sequences, each of which can lead to a different running time and behavior of the resulting algorithm. Consequently, many important questions concerning general upper and lower bounds for Shellsort have remained open, in spite of a number of attempts to solve them. Apart from pure mathematical curiosity, the interest in Shellsort is motivated by the good performance of many of the known increment sequences. The algorithm is very easy to implement, and outperforms most other sorting methods on small or nearly sorted input files. Moreover, Shellsort is an in-place sorting algorithm, so it is very space-efficient.

3.1.1 Previous Results on Shellsort

The original algorithm proposed by Shell was based on the increment sequence given by $h_{m-1} = \lfloor n/2 \rfloor$, $h_{m-2} = \lfloor n/4 \rfloor, \ldots, h_0 = 1$. However, this choice of H leads to a worst case running time of $\Theta(n^2)$ if n is a power of 2. Subsequently, several authors proposed modifications to Shell's original sequence [63, 34, 50] in the hope of obtaining a better running time. Papernov and Stasevich [84] showed that the sequence of Hibbard [34], consisting of the increments of the form $2^k - 1$, achieves a running time of $O(n^{3/2})$. A common feature of all of these sequences is that they are *nearly geometric*, meaning that they approximate a geometric sequence within an additive constant.

An exception is the sequence designed by Pratt [95], which consists of all increments of the form $2^i 3^j$. This sequence gives a running time of $O(n \lg^2 n)$, which still represents the best asymptotic bound known for any increment sequence. In practice, the sequence is not popular because it has length $\Theta(\lg^2 n)$; implementations of Shellsort tend to use $O(\lg n)$-length increment sequences because these result in better running times for files of moderate size [36]. In addition, there is no hope of getting an $O(n \lg n)$-time algorithm based on a sequence of length $\omega(\lg n)$.

Pratt [95] also showed an $\Omega(n^{3/2})$ lower bound for all nearly geometric sequences. Partly due to this result, it was conjectured for quite a while that $\Theta(n^{3/2})$ is the best worst-case running time achievable by increment sequences of length $O(\lg n)$. However, in 1982, Sedgewick [103] improved this upper bound to $O(n^{4/3})$, using an approximation of a geometric sequence that is not nearly geometric in the above sense. Subsequently, Incerpi and Sedgewick [36] designed a family of $O(\lg n)$-length increment sequences with running times $O(n^{1+\epsilon/\sqrt{\lg n}})$, for all $\epsilon > 0$. Chazelle achieves a similar running time with a class of nonuniform sequences [36]; his construction is based on a generalization of Pratt's sequence.

The sequences proposed by Incerpi and Sedgewick are all within a constant factor of a geometric sequence, that is, they satisfy $h_j = \Theta(\alpha^j)$ for some constant $\alpha > 0$. Weiss [122, 125] showed that all sequences of this type take time $\Omega(n^{1+\epsilon/\sqrt{\lg n}})$, but his proof assumed an as yet unproven conjecture on the number of inversions in the Frobenius pattern. Based on this so-called Inversion Conjecture, he also showed an $\Omega(n^{1+\epsilon/\sqrt{\lg n}})$ lower bound for the $O(\lg n)$-length increment sequences of Chazelle. The question of existence of Shellsort algorithms with running time $O(n \lg n)$ remained unresolved.

The two classes of increment sequences given by Incerpi and Sedgewick and by Chazelle are of particular interest because they not only establish an improved upper bound for sequences of length $O(\lg n)$, but also indicate an interesting trade-off between the running time and the length of an increment sequence. Specifically, using a construction described in [36], it is possible to achieve better asymptotic running times by allowing longer increment sequences.

Another goal in the study of Shellsort is the construction of sorting networks of small depth and size. A Shellsort sorting network of depth $\approx 0.6 \lg^2 n$ based on increments of the form $2^i 3^j$ was given by Pratt [95]. Thus, his network came very close to the fastest known network at that time, due to Batcher [7], with depth $\approx 0.5 \lg^2 n$. In 1983, Ajtai, Komlós, and Szemerédi [2] designed a sorting network of depth $O(\lg n)$; however, their construction suffers from an irregular topology and a large constant hidden by the O-notation. This situation has motivated the search for $O(\lg n)$-depth sorting networks with simpler topologies or a smaller multiplicative constant. Shellsort has been considered a potential candidate for such a network [119], due to the rich variety of possible increment sequences and the lack of non-trivial general lower bounds. The lower bounds of Pratt and Weiss also apply to network size, but they only hold for very restricted classes of increment sequences.

Cypher [24] has established an $\Omega(n \lg^2 n / \lg \lg n)$ lower bound for the size of Shellsort networks. However, his proof technique only works for *monotone* increment sequences, that is, sequences that are monotonically increasing. Though this captures a very general class of sequences, it does not rule out the possibility of an $O(\lg n)$-depth network based on some nonmonotone sequence.

Very recently, and independent of our work, Poonen [92] has shown a lower bound of $\Omega(n \lg^2 n / (\lg \lg n)^2)$ that holds for arbitrary Shellsort algorithms. His lower bound also has the form of a trade-off between the running time of a Shellsort algorithm and the length of the underlying increment sequence. The proof uses techniques from solid geometry and is quite intricate. A comparison of Poonen's results and the results in this chapter will be given in the next subsection.

3.1.2 Overview of this Chapter

In this chapter we show lower bounds on the worst-case complexity of Shellsort. In particular, we give a fairly simple proof of an $\Omega(n \lg^2 n/(\lg\lg n)^2)$ lower bound for the size of Shellsort networks, for arbitrary increment sequences. We also establish an identical lower bound for the running time of Shellsort algorithms, again for arbitrary increment sequences. As in Poonen's paper, our lower bounds establish a trade-off between the running time of an algorithm and the length of the underlying increment sequence. This gives lower bounds for increment sequences of length $O(\lg n)$ that come very close to the best known upper bounds. At the other end of the spectrum, the trade-off implies that no increment sequence can match Pratt's upper bound with significantly fewer increments.

Our proof technique is based on purely combinatorial arguments, and we believe that it is significantly simpler than the technique used by Poonen. The technique also leads to certain improvements in the lower bounds, particularly in the trade-off between the running time and the length of the increment sequence. The result by Poonen, on the other hand, is of independent interest, since it establishes a variant of the Inversion Conjecture of Weiss [125] using a new geometric approach to the Frobenius Problem. The technique used in this chapter is not based on a proof of the Inversion Conjecture. Instead, it shows how to "combine" Frobenius patterns to construct permutations with a large number of inversions. This result, together with the idea of dividing an increment sequence into "stages" (also called "intervals" in [92]), leads to the strong lower bounds of this chapter.

Throughout this chapter, we will limit our attention to increment sequences of length $O(\lg^2 n/(\lg\lg n)^2)$. Lower bounds for longer increment sequences are implied by the fact that Shellsort performs at least $\Omega(n)$ comparisons for every increment less than $n/2$. The results of this chapter are presented in an "incremental" fashion, starting with a very basic argument for a restricted class of algorithms, and extending the lower bounds to more general classes in each of the subsequent sections.

This chapter is organized as follows. Section 3.2 illustrates our proof technique by giving a simple and informal argument showing a lower bound for the depth of Shellsort networks based on monotone increment sequences. Section 3.3 introduces a number of definitions and simple lemmas, and then proceeds to give a formal proof of a general lower bound for the depth and size of arbitrary Shellsort networks. Section 3.4 then establishes a lower bound on the running time of non-oblivious Shellsort algorithms based on arbitrary increment sequences. Section 3.5 contains a discussion of our results and a comparison with the best known upper bounds. Finally, Section 3.6 lists some open questions for future research.

3.2 The Basic Proof Idea

In this section we illustrate our proof idea by giving a very simple and informal argument showing a polylogarithmic lower bound for the depth of any Shellsort network based on a monotone increment sequence of length at most $c \lg^2 n/(\lg\lg n)^2$, for some small c. In the following sections, we will then formalize and extend this technique to obtain more general lower bounds.

Let H be a monotone increment sequence with $m \leq c \lg^2 n/(\lg\lg n)^2$ increments. We now divide the increment sequence H into a number of *stages* $S_0, \ldots, S_{t-1}$. Every stage S_i is a set consisting of all increments h_j of H such that $n_i \geq h_j > n_{i+1}$, where $n_0, \ldots, n_t$ are chosen appropriately. We define the n_i by $n_0 = n$ and $n_{i+1} = n_i/\lg^k n_i$, for $i \geq 0$ and some fixed integer k. In this informal argument, we will not be concerned about the integrality of the expressions obtained. Note that the n_i divide the increment sequence into at least $\frac{\lg n}{k \lg\lg n}$ disjoint stages. There are at least $s \stackrel{\text{def}}{=} \frac{\lg n}{2k \lg\lg n}$ disjoint stages consisting of increments $h_j \geq n^{1/2}$.

By averaging, one of these stages, say S_i, will contain at most $m/s \leq \frac{2ck \lg n}{\lg\lg n}$ increments. Now suppose there exists an input permutation A such that, after sorting A by all increments in stages S_0 to S_i, some element is still $\Omega(n_i)$ positions away from its final position in the sorted file. Since H is monotone, we know that from now on only comparisons over a distance of at most n_{i+1} positions will be performed. Hence, we can conclude that the element has to pass through at least $\Omega(n_i/n_{i+1}) = \Omega(\lg^k n)$ comparators in order to reach its final, correct position.

To complete the proof we have to show the existence of a permutation A such that some element is still "far out of place" after sorting A by all increments in S_0 to S_i. We will only give an informal argument at this point; a formal proof will be given in the next subsection. Consider all permutations of length n of the following form: Every element is in its correct, final position, except for the elements in a block of size n_i, ranging from some position a to position $a + n_i - 1$ in the permutation. The elements in this block are allowed to be scrambled up in an arbitrary way. It is easy to see that a permutation of this form is already sorted by all increments greater than n_i, that is, all increments in stages S_0 to S_{i-1}. Hence, no exchanges will occur during these stages.

We now look at what happens in the block of size n_i during stage S_i. Note that no element outside the block will have an impact on the elements in the block. Thus, when we sort the permutation by some increment h_j with $n_i \geq h_j > n_{i+1}$, the new position of any element only depends on its previous position and on the elements in the at most $\frac{n_i}{h_j} \leq \lg^k n_i$ other positions in the block that are in its h_j-class. By our assumption, there are at most $m/s \leq \frac{2ck \lg n}{\lg \lg n}$ increments in stage S_i. Hence, the position of an element after stage S_i only depends on its position before the stage, which can be arbitrary, and on the elements in at most

$$\left(\lg^k n_i\right)^{m/s} \leq (\lg n_i)^{\frac{2ck^2 \lg n}{\lg \lg n}}$$

other positions. If we choose c such that $4ck^2 < 1 - \epsilon$, for some $\epsilon > 0$, then we get

$$\begin{aligned}(\lg n_i)^{\frac{2ck^2 \lg n}{\lg \lg n}} &\leq 2^{2ck^2 \lg n} \\ &\leq 2^{4ck^2 \lg n_i}\end{aligned}$$

$$= \quad o(n_i).$$

This means that for large n, the position of an element in the block after sorting by all increments in S_i will only depend on the elements in $o(n_i)$ other positions in the block. If we assign the smallest elements in the block to these positions, then an element that is larger than these, but smaller than all other elements will end up in a position close to the largest elements after stage S_i. Hence, this element is $\Omega(n_i)$ positions away from its final position. All in all, we get the following result:

Theorem 3.2.1 *Let H be a monotone increment sequence of length at most $c \lg^2 n/(\lg\lg n)^2$, and let k be such that $4ck^2 < 1 - \epsilon$, for some $\epsilon > 0$. Then any sorting network based on H has depth $\Omega\left(\lg^k n\right)$.*

The above argument is quite informal and does not make use of the full potential of our proof technique; it has mainly been given to illustrate the basic proof idea and to demonstrate its simplicity. The above result implies that we cannot hope to match the $O(\lg^2 n)$-depth upper bound of Pratt [95] with any increment sequence of fewer than $\frac{(1-\epsilon)\lg^2 n}{16(\lg\lg n)^2}$ increments, thus answering a question left open by Cypher's lower bound [24]. It also implies that we cannot achieve polylogarithmic depth with increment sequences of length $o(\lg^2 n/(\lg\lg n)^2)$.

By extending the argument we will be able to show much stronger lower bounds for shorter increment sequences. More precisely, we can get a trade-off between depth and increment sequence length by choosing appropriate values for the integers n_i that divide the increment sequence into stages. We can also extend the result to non-oblivious Shellsort algorithms by showing the existence of an input such that not just one, but "a large number" of elements are "far out of place" after the sparse stage S_i.

3.3 Lower Bounds for Networks

In this section, we will show general lower bounds for the depth and size of Shellsort sorting networks. We start off by giving a number of definitions and simple lemmas. We then show how to formalize and generalize the argument of Section 3.2 to obtain a trade-off between the depth of a Shellsort network and the length of the underlying increment sequence. Next, we explain how the results on network depth imply lower bounds on the size of Shellsort networks. We conclude this section by extending our results to nonmonotone increment sequences.

3.3.1 Definitions and Simple Lemmas

This section contains a number of basic definitions and associated lemmas. All of the lemmas are quite straightforward and so their proofs have been omitted.

We will use $\Pi(n)$ to denote the set of $n!$ permutations over $\{0, \ldots, n-1\}$. A 0-1 permutation of length n is an n-tuple over $\{0, 1\}$. Thus $\{0, 1\}^n$ denotes the set of 2^n 0-1 permutations.

Throughout this chapter we will assume that the input files are drawn from $\Pi(n)$. We will use the letters A, B, and C to denote elements from $\Pi(n)$, and we will use X, Y, and Z to denote 0-1 permutations. We say that a file A is *h-sorted* if $A[i] \leq A[i+h]$, for $0 \leq i < n-h$. The following trivial lemma arises as a special case of the last definition.

Lemma 3.3.1 *Every file of length n is h-sorted for any $h \geq n$.*

In the following, let $H = h_0, \ldots, h_{m-1}$ be an increment sequence of length $m \geq 1$. Let $\min(H)$ denote the smallest increment in H. We say that a file is *H-sorted* if and only if it is h_i-sorted for all i such that $0 \leq i < m$.

Definition 3.3.1 *Let template(H, n) denote the 0-1 permutation X obtained by setting $X[i]$ to 1 if and only if there exist nonnegative integers $a_0, \ldots, a_{m-1}$ such that*

$$i = \sum_{0 \leq j < m} a_j \cdot h_j.$$

Lemma 3.3.2 *The 0-1 permutation template(H, n) is H-sorted.*

Lemma 3.3.3 *The number of 1's in the 0-1 permutation template(H, n) is at most*

$$\left\lceil \frac{n}{\min(H)} \right\rceil^m.$$

Definition 3.3.2 *For any 0-1 permutation X of length n' with $0 \leq n' \leq n$, let pad(X, n) denote the 0-1 permutation Y of length n obtained by setting*

$$Y[i] = \begin{cases} X[i] & 0 \leq i < n', \text{ and} \\ 1 & n' \leq i < n. \end{cases}$$

Lemma 3.3.4 *Let X be an arbitrary 0-1 permutation of length n' with $0 \leq n' \leq n$. Then X is H-sorted if and only if pad(X, n) is H-sorted.*

Definition 3.3.3 *For any 0-1 permutation X of length $n \geq 0$, and any integer k, let shift(X, k) denote the 0-1 permutation Y obtained by setting*

$$Y[i] = \begin{cases} 0 & \text{if } i - k < 0, \\ X[i-k] & \text{if } 0 \leq i - k < n, \text{ and} \\ 1 & \text{if } i - k \geq n. \end{cases}$$

Lemma 3.3.5 *For any 0-1 permutation X and any integer k, if X is H-sorted, then shift(X, k) is also H-sorted.*

In the following definition, a Boolean expression is assumed to represent 1 if it is true, and 0 if it is false.

Definition 3.3.4 *For any 0-1 permutation X of length $n \geq 0$, let perm(X) denote the permutation Y in $\Pi(n)$ obtained by setting*

$$Y[i] = \sum_{0 \leq j < i} (X[j] = X[i]) + \sum_{0 \leq j < n} (X[j] < X[i]),$$

Lemma 3.3.6 *Let X be an arbitrary 0-1 permutation. Then X is H-sorted if and only if perm(X) is H-sorted.*

We say that an element $A[j]$ of a permutation A in $\Pi(n)$ is k places out of position if $|A[j] - j| \geq k$. The following lemma will be used in Section 3.3.2 to obtain a simple lower bound on the depth of any Shellsort sorting network.

Lemma 3.3.7 *Let X denote any H-sorted 0-1 permutation of length n, let i denote the number of 1's in X, and let j denote the least index such that $X[j] = 1$ (if $i = 0$ then set $j = n$). Then element perm$(X)[j]$ is $n - i - j$ places out of position.*

3.3.2 A More General Lower Bound

We will now generalize the proof technique presented in the previous section to obtain a trade-off between the length of an increment sequence H and the lower bound for the depth of a sorting network based on H. For the sake of simplicity, we assume H to be monotone. It will be shown later that this assumption is not really necessary.

As before, we divide the increment sequence into stages $S_0, \ldots, S_{t-1}$, such that stage S_i contains all increments h_j with $n_i \geq h_j > n_{i+1}$. We define the n_i by $n_0 = n$ and $n_{i+1} = \left\lfloor n_i / \lg^k n_i \right\rfloor$, but we now assume k to be a function of the input size n and the increment sequence length m. Note that the number of stages t is determined by our choice of k. In particular, if we choose k such that

$$\left(\lg^k n\right)^s \leq n^{1/2},$$

then we get at least s stages that contain only elements greater $n^{1/2}$. Solving this inequality we get

$$k = \left\lfloor \frac{\lg n}{2s \lg\lg n} \right\rfloor . \tag{3.1}$$

as a possible choice of k. We will now formalize our earlier observation that an element can be "far out of place" after sorting by all increments up to stage S_i, provided that S_i contains "few" increments.

Lemma 3.3.8 *Let H be an increment sequence for permutations of length n, and suppose that for some integers ν, ν' with $0 < \nu' < \nu \leq n$ there are at most μ increments h_j with $\nu' < h_j \leq \nu$ in H. If $\lceil \frac{\nu}{\nu'} \rceil^{\mu} = o(\nu)$, then there exists an input file A such that: (i) A is sorted by all $h_j > \nu'$, and (ii) there exists an element in A that is $\Omega(\nu)$ places away from its final position.*

Proof: Let H' denote the subsequence of H consisting of all increments h_j such that $h_j > \nu'$, let H'' denote the subsequence of H' consisting of all increments h_j such that $h_j \leq \nu$, and let $X = \mathit{template}(H'', \nu)$. We know that X is H''-sorted by Lemma 3.3.2. Lemma 3.3.1 then implies that X is also H'-sorted. Note that $|H''| \leq \mu$ and $\min(H'') > \nu'$. Hence, by Lemma 3.3.3, the number of 1's in X is at most

$$\left\lceil \frac{\nu}{\nu'} \right\rceil^{\mu} = o(\nu).$$

Now let $Y = \mathit{pad}(X, n)$. By Lemma 3.3.4, the 0-1 permutation Y is H'-sorted. Furthermore, since the number of 1's in Y is exactly $n - \nu$ greater than the number of 1's in X, and $X[0] = Y[0] = 1$, Lemma 3.3.6 implies that the permutation $A \stackrel{\text{def}}{=} \mathit{perm}(Y)$ is H'-sorted, and by Lemma 3.3.7 some element of A is $\Omega(\nu)$ places out of position.

□

In the preceding argument, we could also have defined Y as $\mathit{shift}(\mathit{pad}(X, n), j)$ for any integer j with $0 \leq j \leq n - \nu$. We will make use of this observation to establish Corollary 3.3.1.1 below.

Theorem 3.3.1 *Any Shellsort sorting network based on a monotone increment sequence of length m has depth*

$$\Omega\left(2^{\frac{\lg n}{(2+\epsilon)\sqrt{m}}}\right),$$

for all $\epsilon > 0$.

Proof: We will partition the increment sequence H into at least $s = (1+\epsilon_0)\sqrt{m}$ disjoint stages consisting of increments h_j with $h_j \geq n^{1/2}$, for some $\epsilon_0 > 0$. By averaging, one of these stages, say S_i, will contain at most

$$\mu \stackrel{\text{def}}{=} \left\lfloor \frac{m}{s} \right\rfloor = \left\lfloor \frac{\sqrt{m}}{1+\epsilon_0} \right\rfloor$$

increments. Using Equation 3.1 we determine k as:

$$k = \left\lfloor \frac{\lg n}{2(1+\epsilon_0)\sqrt{m}\lg\lg n} \right\rfloor$$

Define $\nu = n_i$ and $\nu' = n_{i+1}$. We now have

$$\begin{aligned}
\left\lceil \frac{\nu}{\nu'} \right\rceil^{\mu} &\leq \left\lceil \lg^k \nu \right\rceil^{\frac{\sqrt{m}}{1+\epsilon_0}} \\
&\leq (\lg n)^{\frac{\lg n}{2(1+\epsilon_0)^2 \lg\lg n}} \\
&= 2^{\frac{\lg n}{2(1+\epsilon_0)^2}} \\
&\leq 2^{\frac{\lg \nu}{(1+\epsilon_0)^2}} \\
&= o(\nu).
\end{aligned}$$

Thus, we can apply Lemma 3.3.8. According to the lemma, there exists a permutation such that an element is $\Omega(n_i)$ positions away from its final position after stage S_i. Since all subsequent increments are less than or equal to n_{i+1}, this element must pass through at least $\Omega(n_i/n_{i+1})$ comparators. We have

$$\begin{aligned}
\frac{n_i}{n_{i+1}} &\geq \lg^k n_i \\
&= (\lg n_i)^{\left\lfloor \frac{\lg n}{2(1+\epsilon_0)\sqrt{m}\lg\lg n} \right\rfloor} \\
&\geq (\lg n_i)^{\frac{\lg n}{2(1+\epsilon_0)(1+\epsilon_1)\sqrt{m}\lg\lg n}}
\end{aligned}$$

$$
\begin{aligned}
&\geq \left(2^{\frac{\lg\lg n_i}{\lg\lg n}}\right)^{\frac{\lg n}{2(1+\epsilon_0)(1+\epsilon_1)\sqrt{m}}} \\
&\geq \left(2^{\frac{1}{1+\epsilon_2}}\right)^{\frac{\lg n}{2(1+\epsilon_0)(1+\epsilon_1)\sqrt{m}}} \\
&= 2^{\frac{\lg n}{(2+\epsilon)\sqrt{m}}},
\end{aligned}
$$

where ϵ is chosen to satisfy the inequality $(2+\epsilon) \geq 2(1+\epsilon_0)(1+\epsilon_1)(1+\epsilon_2)$.

□

3.3.3 A Lower Bound for Network Size

The depth lower bound of Theorem 3.3.1 also implies a lower bound on the size of any Shellsort network based on a monotone increment sequence. We will not give a formal proof of this result, since it arises as a special case of the lower bound for the running time of non-oblivious Shellsort algorithms established in the next section. Instead, we will briefly describe the main idea.

Lemma 3.3.8 shows how to construct an input file A that is sorted under all increments in stages S_0 to S_i of an increment sequence H such that one element $A[z]$ in A is "far out of place". In fact, as discussed immediately after the proof of Lemma 3.3.8, we can use the method of the lemma to construct a set of $n - n_i$ "shifted" versions of such an input file A. In particular, let A_j, $0 \leq j < n-n_i$, denote the input file obtained by setting Y to *shift*$(pad(X, n), j)$ instead of $pad(X, n)$. Note that $A_0 = A$. Let $A_0[z]$ be the element proven to be far out of place in A_0. By construction, the element $A_j[z+j]$ is far out of place in A_j. Due to the common structure of the input files, element $A_j[z+j]$ in file A_j will never pass through the same comparator as element $A_k[z+k]$ in A_k, for any $j \neq k$. Instead, the two elements will always be exactly $k - j$ positions apart at each level of the sorting network. This implies the result.

Corollary 3.3.1.1 *Any sorting network based on a monotone increment sequence* *of length m has size*

$$\Omega\left(n \cdot 2^{\frac{\lg n}{(2+\epsilon)\sqrt{m}}}\right),$$

for all $\epsilon > 0$.

We can now compare our result to the lower bound of $\Omega(n \lg^2 n / \lg\lg n)$ for network size given by Cypher [24]. The main difference between the two results is that Cypher gets a lower bound that is independent of the length of the increment sequence, while we get a trade-off between network size and increment sequence length. This makes our lower bound much stronger for short increment sequences. Our method also implies a lower bound of $\Omega(n \lg^2 n / (\lg\lg n)^2)$ for increment sequences of arbitrary length, since every increment increases the size of a Shellsort network by at least n. This is slightly weaker than Cypher's lower bound. However, Cypher's bound only applies to monotone increment sequences, while our result also holds for nonmonotone sequences, as will be shown in the next subsection. Another strength of our method is its simplicity and flexibility, which will makes it possible to extend our lower bound to non-oblivious Shellsort algorithms and certain variations of Shellsort.

3.3.4 Nonmonotone Increment Sequences

So far, we have restricted our attention to monotone increment sequences. We will now show that this restriction is really unnecessary, and that the same lower bounds also apply to nonmonotone sequences. Recall that we obtained the depth lower bound by showing the existence of an input permutation such that an element is "far out of place" after the "sparse" stage S_i. More precisely, Lemma 3.3.8 showed the existence of a permutation A that is already sorted by all increments in stages S_0 through S_i and that contains such an element. Thus, no exchanges are performed by the increments in stages S_0 through S_i on input A, and the lower bound follows. We will make use of the following well-known lemma (see, for example, [95]) in order to extend this argument to nonmonotonic increment sequences.

Lemma 3.3.9 *For any two increments h, h', if we h'-sort an h-sorted file, it stays h-sorted.*

Now suppose we have a nonmonotone increment sequence H. We can divide H into stages $S_0, \ldots, S_{t-1}$ as before, with stage S_i containing all increments h_j with $n_i \geq h_j > n_{i+1}$. Again, there exists a "sparse" stage S_i with few increments, and a permutation sorted by all increments in $S \stackrel{\text{def}}{=} S_0 \cup \cdots \cup S_i$ such that some element is "far out of place". If we take A as the input permutation, then by Lemma 3.3.9 A will stay sorted by all increments in S throughout the network. Hence, no exchanges will take place during the applications of Insertion Sort corresponding to increments in S. This implies that all of the exchanges needed to move the "out-of-place" element to its final position are performed by increments $h_j \leq n_{i+1}$, and the lower bound follows. The same reasoning also applies to the lower bound for network size, and to the results obtained in the next section. This gives us the following result:

Corollary 3.3.1.2 *Any sorting network based on an increment sequence of length m has size*

$$\Omega\left(n \cdot 2^{\frac{\lg n}{(2+\epsilon)\sqrt{m}}}\right),$$

for all $\epsilon > 0$.

Note that this result does not rule out the existence of nonmonotone increment sequences that perform better than the "corresponding" monotone sequences (that is, the sequences obtained by sorting the nonmonotone sequences into increasing order). It is an open question whether such sequences exist.

3.4 Non-Oblivious Shellsort Algorithms

The results obtained so far all rely on the fact, established in Lemma 3.3.8, that we can construct an input file such that one element is "far away" from its final position in the sorted file. We were able to extend the lower bounds to network size due to the oblivious nature of sorting networks. However, the results for network size do not imply a lower bound for the running time of Shellsort algorithms that are non-oblivious.

In this subsection, we will establish such a lower bound. The high-level structure of the proof is the same as that of the depth lower bound in the last section; we only have to substitute Lemma 3.3.8 by a stronger lemma showing that there exists an input file A such that not just one, but "a large number" of the elements in A are "far away" from their final position. This result is formalized in the following lemma, which we will prove later in this subsection.

Lemma 3.4.1 *Let H be an increment sequence applied to input files of length n, and suppose that for some integers ν, ν' with $4 \leq \nu' < \nu \leq n$ there are at most μ increments h_j with $\nu' < h_j \leq \nu$ in H. If $\lceil \frac{\nu}{\nu'} \rceil^{\mu} \leq \nu / \lg^3 \nu$, then there exists an input file A such that: (i) A is sorted by all $h_j > \nu'$, and (ii) there exist $\Omega(n / \lg^3 \nu)$ elements in A that are $\Omega(\nu / \lg^2 \nu)$ places away from their final position.*

Given an increment sequence H, we can establish the lower bound for non-oblivious Shellsort algorithms by dividing H into stages in the same way as in the proof of Theorem 3.3.1, and then applying the above Lemma 3.4.1 instead of Lemma 3.3.8. The lower bound obtained is slightly weaker than the one for network size, since Lemma 3.4.1 only shows that a polylog fraction of the elements are a polylog fraction of n_{i-1} out of place. This gives the following theorem:

Theorem 3.4.1 *Any Shellsort algorithm based on an increment sequence of length* *m has running time*

$$\Omega\left(\frac{n}{\lg^5 n}\cdot 2^{\frac{\lg n}{(2+\epsilon)\sqrt{m}}}\right),$$

for all $\epsilon > 0$.

We remark that the exponent "5" in the preceding theorem is not the best possible. It results from summing the exponents "3" and "2" appearing in the statement of Lemma 3.4.1, which can be improved to "2" and "1", respectively. We have chosen to weaken these constants in order to simplify the proof of Lemma 3.4.1.

Comparing the bound of Theorem 3.4.1 to previous results we note that the lower bounds of Pratt [95] and Weiss [122] only hold for increment sequences approximating a geometric sequence, while the lower bound of Theorem 3.4.1 applies to all increment sequences. Also, the bound given by Weiss, which holds for a more general class than Pratt's bound, is based on an unproven conjecture about the number of inversions in certain input files.

The remainder of this subsection contains the proof of Lemma 3.4.1. To establish the result, we will need a few technical lemmas. The first two lemmas are straightforward and their proofs will be omitted. In particular, Lemma 3.4.2 is a straightforward generalization of Lemma 3.3.7.

Lemma 3.4.2 *Let X denote any H-sorted 0-1 permutation of length n, let i denote the number of 1's in X, let n' be such that $0 \le n' < n-2i$, and let $j = \sum_{0\le k<n'} X[k]$. Then at least j elements of $perm(X)$ are $n - n' - i$ places out of position.*

Definition 3.4.1 *For any 0-1 permutation X of length n' such that $0 \le n' \le n$, let $perm^*(X, n)$ denote the permutation Y in $\Pi(n)$ obtained from $Z \stackrel{def}{=} perm(X)$ by setting*

$$Y[i] = Z[i \bmod n'] + \left\lfloor \frac{i}{n'} \right\rfloor \cdot n'.$$

Lemma 3.4.3 *Let X be any 0-1 permutation of length n' such that $0 \le n' \le n$. Then X is H-sorted if and only if perm$^*(X, n)$ is H-sorted. If i elements of perm(X) are j places out of position, then at least $i \cdot \lfloor n/n' \rfloor$ elements of perm$^*(X, n)$ are j places out of position.*

In the following, let H be an arbitrary increment sequence. Let ν be any integer with $\nu \ge 4$, and define $\alpha \stackrel{\text{def}}{=} \nu - 2\nu/\lg^2 \nu$ and $\beta \stackrel{\text{def}}{=} \nu - \nu/\lg^2 \nu$.

Lemma 3.4.4 *Let X denote a 0-1 permutation of length $\nu \ge 4$ with $X[0] = 1$ and $\sum_{0 \le i < \nu} X[i] \le \nu/\lg^3 \nu$. Then there exists an integer k, $0 \le k \le (\nu - \alpha)\lfloor \lg \nu \rfloor$, such that the 0-1 permutation $Y \stackrel{\text{def}}{=}$ shift(X, k) satisfies*

$$\sum_{0 \le i < \alpha} Y[i] \ge \sum_{\alpha \le i < \nu} Y[i].$$

Proof: Suppose, for the sake of contradiction, that

$$\sum_{0 \le i < \alpha} \textit{shift}(X,k)[i] < \sum_{\alpha \le i < \nu} \textit{shift}(X,k)[i]$$

holds for all k with $0 \le k \le (\nu - \alpha)\lfloor \lg \nu \rfloor$. This implies that $\sum_{0 \le i < \alpha - k} X[i] < \sum_{\alpha - k \le i < \nu - k} X[i]$. Using $R_k \stackrel{\text{def}}{=} \sum_{0 \le i < \alpha - k} X[i]$, this can be rewritten as $R_k < R_{k-(\nu-\alpha)} - R_k$, or $R_k < \frac{1}{2} R_{k-(\nu-\alpha)}$. Hence,

$$R_{(\nu-\alpha)\lfloor \lg \nu \rfloor} < 2^{-\lfloor \lg \nu \rfloor} R_0,$$

and from $R_0 \le \sum_{0 \le i < \nu} X[i] \le \nu/\lg^3 \nu$ we get

$$R_{(\nu-\alpha)\lfloor \lg \nu \rfloor} < \frac{2}{\lg^3 \nu} < 1.$$

This is clearly a contradiction, since $X[0] = 1$ implies $R_{(\nu-\alpha)\lfloor \lg \nu \rfloor} \ge 1$.

□

In the next lemma, given 0-1 permutations X and Y, we will use $or(X, Y)$ to denote the 0-1 permutation Z obtained by setting bit $Z[i]$ to the logical OR of bits $X[i]$ and $Y[i]$, $0 \le i < n$. Clearly, if X and Y are H-sorted, then $or(X, Y)$ is also H-sorted.

Lemma 3.4.5 *Let X be an H-sorted 0-1 permutation of length ν with $x \stackrel{def}{=} \sum_{0 \le i < \nu} X[i] \le \nu/\lg^3 \nu$. Let $x' \stackrel{def}{=} \sum_{0 \le i < \alpha + k} X[i]$ where $(x \lg \nu)/2 \le k \le \nu/(2 \lg^2 \nu)$. Then there exists an H-sorted 0-1 permutation Y of length ν with $\sum_{0 \le i < \nu} Y[i] \le 2x$ and*

$$\sum_{0 \le i < \alpha + 2k} Y[i] \ge 2\left(1 - \frac{1}{\lg \nu}\right) x'.$$

Proof: We will set Y to $Y_j \stackrel{\mathrm{def}}{=} or(X, \mathit{shift}(X, j))$ for some appropriately chosen integer j, $1 \le j \le k$. Note that by Lemma 3.3.5, any such 0-1 permutation Y_j is H-sorted, and it is easy to see that $\sum_{0 \le i < \nu} Y_j[i] \le 2x$ holds. Let $y'_j = \sum_{0 \le i < \alpha + 2k} Y_j[i]$. It remains to show the existence of an integer j_0, $1 \le j_0 \le k$, such that $y'_{j_0} \ge 2(1 - 1/\lg \nu) x'$. We will accomplish this by means of an averaging argument. We have

$$\sum_{1 \le j \le k} y'_j \ge 2kx' - \binom{x'}{2}.$$

Hence, there exists a j_0, $1 \le j_0 \le k$, such that

$$\begin{aligned} y'_{j_0} &\ge \frac{2kx' - \binom{x'}{2}}{k} \\ &\ge 2x' - \frac{x' - 1}{\lg \nu} \\ &\ge 2\left(1 - \frac{1}{\lg \nu}\right) x'. \end{aligned}$$

Now choose $Y = Y_{j_0}$.

□

Lemma 3.4.6 *Let Y be an H-sorted 0-1 permutation of length ν with $\sum_{0 \le i < \nu} Y[i] \le \nu/\lg^3 \nu$ and*

$$\sum_{0 \le i < \alpha} Y[i] \ge \sum_{\alpha \le i < \nu} Y[i].$$

Then there exists an H-sorted 0-1 permutation Z of length ν such that $\sum_{0 \le i < \nu} Z[i] \le \nu/\lg^3 \nu$ and

$$\sum_{0 \le i < \beta} Z[i] = \Omega(\nu/\lg^3 \nu).$$

Proof: We will "transform" the given 0-1 permutation Y into a 0-1 permutation of the desired form by a sequence of applications of Lemma 3.4.5. Let $Y_0 \stackrel{\text{def}}{=} Y$. The jth application of Lemma 3.4.5 will be used to obtain Y_j from Y_{j-1}, $j \geq 1$. Let $y_j = \sum_{0 \leq i < \nu} Y_j[i]$, and let $y'_j = \sum_{0 \leq i < \alpha + 2^j y_0 \lg \nu} Y_j[i]$. Note that $y'_0 \geq \sum_{0 \leq i < \alpha} Y[i] \geq y_0/2$. Then Lemma 3.4.5 implies that $y_j \leq 2^j y_0$, and

$$y'_j \geq \left[2\left(1 - \frac{1}{\lg \nu}\right)\right]^j y'_0$$

for $j \leq \lg \nu - \lg y_0 - 3 \lg \lg \nu$ (the latter inequality ensures that $\alpha + 2^j y_0 \lg \nu \leq \beta$). Setting j_0 to $\lfloor \lg \nu - \lg y_0 - 3 \lg \lg \nu \rfloor$, and making use of the inequality $y'_0 \geq y_0/2$, we find that $y_{j_0} \leq \nu / \lg^3 \nu$ and

$$y'_{j_0} = \Omega(y_{j_0}) = \Omega(\nu / \lg^3 \nu).$$

Hence, we can choose $Z = Y_{j_0}$.

□

Given the above lemmas, we are now ready to proceed with the proof of Lemma 3.4.1.

Proof: Let H' denote the subsequence of H consisting of exactly those increments h_j such that $h_j > \nu'$, let H'' denote the subsequence of H' consisting of exactly those increments h_j such that $h_j \leq \nu$, and let $X = \textit{template}(H'', \nu)$. We know that X is H''-sorted by Lemma 3.3.2. Note that $|H''| \leq \mu$ and $\min(H'') > \nu'$. Hence, by Lemma 3.3.3, we have

$$\sum_{0 \leq i < \nu} X[i] \leq \left\lceil \frac{\nu}{\nu'} \right\rceil^{\mu} \leq \frac{\nu}{\lg^3 \nu}.$$

By Lemmas 3.3.5 and 3.4.4, the existence of X implies the existence of a 0-1 permutation Y of length ν such that Y is H''-sorted and

$$\sum_{0 \leq i < \alpha} Y[i] \geq \sum_{\alpha \leq i < \nu} Y[i].$$

The existence of Y then establishes, via Lemma 3.4.6, the existence of a 0-1 permutation Z of length ν such that:

- Z is H''-sorted,

- $\sum_{0 \leq i < \nu} Z[i] \leq \nu / \lg^3 \nu$, and

- $\sum_{0 \leq i < \beta} Z[i] = \Omega(\nu / \lg^3 \nu)$.

By Lemma 3.3.1 and Lemma 3.3.6 we know that $B = perm(Z)$ is H'-sorted, and Lemma 3.4.2 implies that B contains $\Omega(\nu/\lg^3 \nu)$ elements that are $\Omega(\nu/\lg^2 \nu)$ places out of position. Let $A = perm^*(Z, n)$. By Lemma 3.4.3, A is H'-sorted and contains $\Omega(n/\lg^3 \nu)$ elements that are $\Omega(\nu/\lg^2 \nu)$ places out of position.

□

3.5 Discussion

In this chapter, we have given a fairly simple proof of a lower bound of $\Omega(n \lg^2 n / (\lg \lg n)^2)$ for the size of any Shellsort network, thus ruling out the existence of a network of size $O(n \lg n)$ based on a nonmonotone increment sequence. By extending our argument to the case of non-oblivious algorithms, we have also established a general lower bound for Shellsort that holds for arbitrary increment sequences.

Our lower bound can be further generalized to a fairly large class of "Shellsort-like" algorithms, including the *Shaker Sort* algorithm of Incerpi and Sedgewick [37, 124] as well as other algorithms proposed by Knuth [50] and Dobosiewicz [27]. Poonen [92] has formally defined a class of such algorithms, called *Shellsort-type* algorithms, and has shown how to extend his lower bound to this class. We will not elaborate further on such possible extensions, and instead refer the reader to the presentation in [92] and [89].

The lower bound of Theorem 3.4.1 establishes a trade-off between the running time of a Shellsort algorithm and the length of the underlying increment sequence. We will now compare this lower bound trade-off with the best known upper bound trade-off given by the nonuniform increment sequences of Chazelle (see the nonuniform case of Theorem 3 in [36]). Expressing the running time as a function of the increment sequence length m we obtain the following bounds:

$$\begin{array}{lcl} \text{Lower Bound:} & T & \geq \frac{n}{\lg^5 n} \cdot n^{\frac{1}{(2+\epsilon)\sqrt{m}}} \\ \text{Upper Bound:} & T & \leq mn \cdot n^{\frac{2}{\sqrt{m}}} \end{array}$$

Note that both the factor $1/\lg^5 n$ in the lower bound and the factor m in the upper bound are only significant for increment sequences of length $\Omega(\lg^2 n/(\lg\lg n)^2)$. In every other case, the upper and lower bounds differ only by a factor of $4+\epsilon$ in the exponent. In the lower bound trade-off shown by Poonen, the constant in the exponent is $1/432$ instead of $1/(2+\epsilon)$.

We can also express the length of the increment sequence as a function of the running time. In this case, for $m = o(\lg^2 n/(\lg\lg n)^2)$, the lower and upper bounds are only a constant factor apart. This means that, for a given T, the length of the increment sequence of Chazelle that achieves running time T is only a factor of $16+\epsilon$ larger than the minimum length possible under our lower bound trade-off. (For Poonen's result, this factor would be much larger.) In other words, one cannot hope to match the running time of Chazelle's sequences with significantly shorter increment sequences.

3.6 Open Questions

The primary remaining challenge in the study of Shellsort seems to be the virtual nonexistence of both upper and lower bounds for the average case complexity. A result for a particular increment sequence is given by Knuth [50], who determines an average case running time of $\Omega(n^{3/2})$ for Shell's original sequence. Increment sequences of the form $(h,1)$ and $(h,k,1)$ were investigated by Knuth [50] and Yao [126], respectively. Weiss [123] conducted an extensive empirical study and conjectured that Shellsort will on average not perform significantly better than in the worst case. Any general upper or lower bounds for the average case would certainly be very interesting.

It would be nice to close the remaining gap between the upper and lower bounds. Our lower bound trade-off comes quite close to the known upper bounds, but there is certainly still room for improvement.

Finally, one might try to find interesting "Shellsort-like" algorithms that are not covered by our proof technique, and that lead to improved running times.

Chapter 4

Deterministic Routing and Sorting on Meshes

Thia chapter considers the problems of deterministic routing and sorting on meshes and related networks. We introduce a new technique that can be used to convert many of the randomized algorithms proposed in the literature into deterministic algorithms with (nearly) matching running times. We describe several applications of this technique, including deterministic algorithms for routing and sorting on the two-dimensional $n \times n$ mesh that run in time $2n + o(n)$ with small constant queue size. Apart from these particular applications, we believe that the technique enhances our understanding of routing and sorting on meshes in general, by showing an interesting relation between randomization and local sorting steps.

4.1 Introduction

In this chapter, we describe algorithms and techniques for routing and sorting on meshes and tori. In our presentation, we concentrate on the case of the two-dimensional mesh, but we also state a number of results for other, related networks.

We are mainly concerned with the problems of 1–1 routing and 1–1 sorting, where before and after the operation each processor holds a single element. In the next paragraphs, we recall and expand some of the definitions given in Section 1.2.

In the following, we assume an $n \times n$ mesh-connected array of synchronous processors. Each of the n^2 processors is identified by its row and column coordinates. Every processor is connected to each of its four neighbors through a bidirectional link, and a bounded amount of information can be transmitted in either direction in a single step of a computation. The *packet routing problem* is the problem of rearranging a set of packets in a network, such that every packet ends up at the processor specified in its destination address. A routing problem in which every processor is the source and destination of at most one packet is called a 1–1 routing problem, or *permutation routing problem*. In the 1–1 sorting problem, we assume that every processor initially holds a single packet, where each packet contains a key drawn from some totally ordered set. Our goal is to rearrange the packets in such a way that the packet with the key of rank k is moved to the unique processor with index k, for all k. The index of a processor in the mesh is determined by an *indexing scheme*.

Formally, an *indexing scheme* for an $n \times n$ mesh is a bijection $\mathcal{I}$ from $[n] \times [n]$ to $[n^2]$. If $\mathcal{I}(i, j) = k$ for some processor $(i, j) \in [n] \times [n]$ and some $k \in [n^2]$, then we say that processor (i, j) has index k. The problem of sorting an input with respect to an indexing scheme $\mathcal{I}$ is to move every element y of the input to the processor with index $\mathcal{I}(\text{Rank}\,(y, X))$, where $\text{Rank}\,(y, X) \stackrel{\text{def}}{=} |\{x \in X \mid x < y\}|$ and X denotes the set of all input elements. An example of a simple indexing scheme is the *row-major* indexing scheme, or *row-major* order, which is given by indexing the processors from the left to the right, and from the top row to the bottom row. It can be formally defined by

$$\mathcal{I}(i_1, j_1) < \mathcal{I}(i_2, j_2) \Leftrightarrow (i_1 < i_2) \vee [(i_1 = i_2) \wedge (j_1 < j_2)].$$

A related indexing scheme is the *snake-like row-major* order defined by

$$\mathcal{I}(i_1, j_1) < \mathcal{I}(i_2, j_2) \Leftrightarrow (i_1 < i_2) \vee [(i_1 = i_2) \wedge ([(i_1 \text{ odd }) \wedge (j_1 < j_2)] \vee [(i_1 \text{ even }) \wedge (j_1 > j_2)])] .$$

Similarly, one can define the *column-major* and *snake-like column-major* orders.

Sorting algorithms on the mesh are usually designed with a particular indexing scheme in mind, and techniques developed for one indexing scheme may not work well for others. Throughout this chapter, we assume a *blocked* indexing scheme similar to the one used in [42, 43]. The indexing scheme is defined by partitioning the mesh into blocks of size $n^\alpha \times n^\alpha$, and using an arbitrary indexing inside each block, while the blocks themselves are ordered in the mesh according to snake-like row-major indexing.

Finally, the *queue size* is the maximum number of packets that have to be stored at any processor during the execution of an algorithm. We point out that there are several different possible definitions of the queue size. Under one definition, each communication link of a processor has an associated buffer that can hold a single packet, and the queue size is defined as the number of packets that do not fit into these buffers, and that are stored in an additional queue inside the processor. We will refer to this as the *internal queue size* of an algorithm. (Thus, a *hot-potato* or *deflection* routing algorithm has an internal queue size of zero.) In the remainder of this thesis, we define the queue size as the maximum number of packets that are located at a single processor at any point in time, including those packets that are routed in the next step. Thus, an internal queue size of c on a d-dimensional mesh corresponds to a queue size of at most $c + 2d$ in our model. A good routing or sorting algorithm should have a fast running time, a small queue size, and a simple control structure.

4.1.1 Previous Results

The study of sorting on the two-dimensional mesh was initiated by Orcutt [83] and Thompson and Kung [114], who gave algorithms based on Batcher's Bitonic Sort [7] with running times of $O(n \lg n)$ and $6n + o(n)$, respectively. In the following years, a number of sorting algorithms were proposed for the mesh (see, for example, [52, 62, 82, 100, 101]); these algorithms make a variety of different assumptions about the power of the underlying model of the mesh. More recently, most of the work has focused on variants of the two models described in the following, which we refer to as the *single-packet model* and the *multi-packet model*.

The *single-packet model* (also often referred to as the *Schnorr-Shamir model*) assumes that a processor can hold only a single packet at any point in time, plus some unbounded amount of additional information. This unbounded additional information may be used to decide the next action taken by the processor; however, it may not be used to create a new packet and substitute it for the currently held packet. At any step in the computation, a single packet plus an unbounded amount of header information may be transmitted across each directed edge. It is assumed that a comparison-exchange operation between adjacent packets can be performed in a single step.

For this model of the mesh, Schnorr and Shamir [102] showed an upper bound of $3n + o(n)$ for sorting into row-major order. They also proved a lower bound of $3n - o(n)$, independently discovered by Kunde [54]. The same proof technique has also been used to show lower bounds for arbitrary indexing schemes [54]; the best general lower bound is currently $2.27n$ [32]. Note that the upper bound does not make use of the unbounded local memory and header information permitted in the model, while the lower bounds hold even under these rather unrealistic assumptions. Thus, the power of the model seems to be mainly determined by the restriction to a single packet per processor.

Another model that has recently received some attention is the *multi-packet model* of the mesh (also sometimes referred to as the *MIMD model*). In this model, a processor may hold a constant number of packets at any point in time, and packets may be copied or deleted. In any step of the computation, a single packet plus $O(\lg n)$ header information can be transmitted across each directed edge, and local memory is restricted to $O(\lg n)$ bits. The only general lower bound for sorting and routing on the multi-packet model of the mesh is given by the diameter of the network, and several groups of authors have recently described sorting algorithms for this model that achieve a running time of less than $3n$.

A $2.5n + o(n)$ time randomized algorithm for this model was given by Kaklamanis, Krizanc, Narayanan, and Tsantilas [43]. Their algorithm requires a queue size of at least 8. Using very different techniques, Kunde [58] designed a deterministic algorithm matching the $2.5n + o(n)$ randomized bound. Apart from being deterministic, Kunde's algorithm also has a number of other advantages over that of Kaklamanis, Krizanc, Narayanan, and Tsantilas. The algorithm has a fairly simple structure, and no processor holds more than 2 packets at any point in time. The algorithm does not make any copies of packets, and it generalizes nicely to meshes of arbitrary dimension and to multi-packet sorting problems. Moreover, the elements are sorted into snake-like row-major order, while the randomized algorithm sorts with respect to the somewhat more complicated blocked indexing scheme mentioned earlier.

However, if one is interested in developing an algorithm that comes closer to the distance bound of $2n - 2$, then it seems very difficult to apply the techniques used in Kunde's deterministic algorithm. In fact, Narayanan [81] has shown that any deterministic algorithm for sorting into row-major order that achieves a queue size of 2, and that does not make any copies of elements, must take at least $2.125n$ steps. The approach taken in the randomized algorithm [43], on the other hand, was subsequently used by Kaklamanis and Krizanc [42] to design an optimal randomized sorting algorithm, with a running time of $2n + o(n)$ and constant queue size.

The permutation routing problem has also been extensively studied under the multi-packet model of the mesh. In particular, Valiant and Brebner [117] proposed a randomized algorithm with a running time of $(2d-1)n + o(n)$ and a queue size of $O(\lg n)$ on the d-dimensional mesh, $d \geq 2$. A deterministic algorithm for the two-dimensional mesh with a running time of $(2+\epsilon)n$ and a queue size of $O(1/\epsilon)$ was described by Kunde [56], and a randomized routing algorithm with running time $2n + o(n)$ and constant queue size was described by Rajasekaran and Tsantilas [99]. Subsequently, Leighton, Makedon, and Tollis [69] gave a deterministic algorithm for routing that runs in $2n-2$ steps with constant queue size. However, the exact value for the queue size is rather large. Rajasekaran and Overholt [98] gave an improved construction that reduced the queue size to below 200. Very recently, Kaklamanis, Krizanc, and Rao have obtained several fairly simple optimal randomized and off-line algorithms for the two-dimensional and three-dimensional mesh, and for the two-dimensional torus.

Finally, the routing problem has also been studied under a number of more restricted models, such as *hot-potato* routing, *oblivious* routing, or routing along minimal paths. For an overview of results on such restricted, and hence more realistic, models, we refer the reader to the surveys in [11, 115].

4.1.2 Overview of this Chapter

In this chapter, we describe a new technique that allows us to convert a number of recently proposed randomized algorithms for routing and sorting into deterministic algorithms that achieve the same running time, within a lower order additive term. We explain the main idea behind the technique, and describe several applications. For convenience, we will sometimes use the term "derandomization" (in quotes) to refer to this technique. However, we point out that it is not related to the techniques that are commonly associated with this term in the literature (e.g., see [77]).

As our first application, we consider a randomized algorithm for routing on two-dimensional meshes proposed by Kaklamanis, Krizanc, and Rao [44]. As a result, we obtain a deterministic algorithm for permutation routing on two-dimensional meshes with a running time of $2n + o(n)$ and a queue size of 5. The only optimal deterministic algorithm previously known for this problem [69, 98] had a running time of $2n - 2$ and a queue size of at least 112. Extending this result to other networks, we obtain the first optimal deterministic algorithms for routing on the three-dimensional mesh and the two-dimensional torus.

Next, we apply the technique to an optimal randomized algorithm for sorting on the two-dimensional mesh proposed by Kaklamanis and Krizanc [42]. We obtain a deterministic algorithm that runs in time $2n + o(n)$ with a queue size of about 25. The fastest deterministic algorithm previously known for this problem [58] achieved a running time of $2.5n + o(n)$ and a queue size of 2. We also obtain improved deterministic algorithms for sorting on three-dimensional meshes and on two-dimensional and three-dimensional tori. Finally, we point out some additional applications of the technique to multi-packet sorting and selection, and to routing on meshes with buses.

The chapter is organized as follows. Section 4.2 defines some terminology. Section 4.3 describes the basic ideas underlying our technique. Section 4.4 contains the results for permutation routing, and Section 4.5 contains the results for sorting. Finally, Section 4.6 describes some other applications of our techniques, and Section 4.7 offers some concluding remarks.

4.2 Terminology

Throughout this chapter, we frequently have to reason about quantities that are determined to within a lower order additive term. We use the notation $\sim f(n)$ ("approximately $f(n)$") to refer to a term in the range between $f(n) - o(f(n))$ and

$f(n) + o(f(n))$. Also, we say that a set of k elements is *evenly distributed* among m sets if every set contains $\sim k/m$ elements. For $k_1, k_2 \geq 1$, a k_1–k_2 relation is a routing problem in which each processor is the source of at most k_1 packets and the destination of at most k_2 packets. An approximate k_1–k_2 relation on a linear array is a routing problem in which each block of m consecutive processors is the source of at most $mk_1 + o(n)$ packets and the destination of at most $mk_2 + o(n)$ packets.

Given a partition of the mesh into blocks of equal size, we use the terms *row of blocks* and *column of blocks* to refer to the sets of blocks with common vertical and horizontal coordinates, respectively. Finally, we say that an algorithm is *optimal* if its running time is $\sim l$, where l is the best lower bound.

4.3 Basic Ideas

A large number of randomized algorithms for routing and sorting on fixed-connection networks have been designed in recent years, and in a number of cases these algorithms are superior to the best deterministic solutions in terms of both performance and simplicity. Many of these algorithms follow a very simple two-phase scheme proposed by Valiant [116], in which the elements are first randomly distributed over a sufficiently large region of the network. In the second phase, the elements are then routed towards their destinations.

The main purpose of the randomization phase in these algorithms is to distribute packets with similar ranks or destinations evenly over the network. In the following, we describe a technique that simulates this effect in a deterministic fashion.

4.3.1 The Sort-and-Unshuffle Operation

Our technique is based on a combination of local sorting and off-line routing. Formally, assume a fixed-connection network with N processors, each containing a single element. Partition the network into $N^{1-\gamma}$ groups of N^{γ} processors, for some $\gamma < 1$, and sort the elements within each group with respect to their key values (or destinations, in the case of a routing problem). After this sorting step, the element with rank j in group i is located in processor j of the group, for all i with $0 \leq i < N^{1-\gamma}$ and all j with $0 \leq j < N^{\gamma}$. In the second step, we route the element in processor j of group i to processor $i + \left\lfloor \frac{j}{N^{1-\gamma}} \right\rfloor \cdot N^{1-\gamma}$ in group $j \bmod N^{1-\gamma}$.

We refer to the above operation as the *sort-and-unshuffle* operation. (This name is motivated by the close relationship between the off-line routing problem in the second step of the operation, and the class of *κ-way unshuffle* permutations defined further below.) Note that after the second step of the operation, all sets of elements with similar key values are approximately evenly distributed among the groups of processors. More precisely, the following holds.

Lemma 4.3.1 *Let A be any set of consecutive values in $[n^2]$. Then after execution of the sort-and-unshuffle operation, every group of processors contains between $\frac{|A|}{N^{1-\gamma}} - N^{1-\gamma}$ and $\frac{|A|}{N^{1-\gamma}} + N^{1-\gamma}$ elements with rank in A.*

In the case of the two-dimensional mesh, the groups of processors in the sort-and-unshuffle operation will usually be square submeshes, or *blocks*, of processors. Thus, we will partition the mesh into blocks of size $n^{\beta} \times n^{\beta}$, for some $\beta < 1$. We assume that the blocks are indexed in such a way that blocks with consecutive indices are adjacent (or close to each other). The packets are then sorted, and the element in processor j of block i is routed to $i + \left\lfloor \frac{j}{n^{2-2\beta}} \right\rfloor \cdot n^{2-2\beta}$ in group $j \bmod n^{2-2\beta}$.

We point out that the idea of deterministically "spreading" elements of similar rank and destination over the network is not really new. In particular, similar techniques are used in the *Columnsort* algorithm of Leighton [64] and the $3n + o(n)$ algorithm of Schnorr and Shamir [102], as well as in several of Kunde's algorithms (e.g., see [57, 59]). However, none of these papers elaborates on the close relationship between these techniques and the ideas used in many randomized algorithms.

In the following, we attempt to explain this relationship, and to design a general method that can be used to "derandomize" (in an informal sense) many of the known randomized algorithms on meshes and related networks. As a result, we not only obtain improved deterministic algorithms for a number of problems related to routing and sorting, but we also get some new insights that may lead to a more unified perspective of the multitude of routing and sorting algorithms that have been proposed for the mesh.

4.3.2 Implementation on Meshes

As an example, consider the following simplified variant of the randomized routing algorithm for the mesh described by Valiant and Brebner [117]. The algorithm first sends every packet to a random location in the network along row-column paths. In the second phase, the packets are routed towards their destinations, again along row-column paths. It can be shown that the above algorithm terminates in approximately $4n$ steps, with high probability; the queue size is $O(\lg n)$. To convert this algorithm into a deterministic algorithm, we substitute the sort-and-unshuffle operation for the first phase of the algorithm. In addition, we also have to change the second phase of the algorithm slightly, in order to avoid the queue size from growing too large. The details of this change depend on the particular structure of the algorithm, the indexing scheme, and the size of the blocks in the sort-and-unshuffle operation, and we will not elaborate on these issues at this point.

While the above algorithm is very simple, many other randomized algorithms have a significantly more complex structure. In particular, the elements are often not randomized over the entire network, but only within a fairly small region. Also, consecutive steps of the computation may be overlapped in a sophisticated manner. This raises a number of additional technical issues that need to be resolved in order to "derandomize" such an algorithm.

To simulate such a restricted randomization within a small region of the network, we can define a corresponding restricted variant of the sort-and-unshuffle operation. For example, the randomized algorithms considered in the subsequent sections use several forms of restricted randomization within rows or columns. A corresponding sort-and-unshuffle operation might, for example, operate within rows of blocks of length n and height n^β. Such a sort-and-unshuffle operation can be implemented using the class of κ-way unshuffle permutations defined in the following.

Definition 4.3.1 *For any $n, \kappa > 0$ with $n \bmod \kappa = 0$, we define the κ-*way unshuffle *permutation on n positions $0, \ldots, n-1$ as the permutation π_κ that moves the element in position i to position $\pi_\kappa(i) \stackrel{def}{=} (i \bmod \kappa) \cdot n/\kappa + \lfloor i/\kappa \rfloor$. We say that we perform a κ-way unshuffle permutation* on the columns (rows) *of a mesh, if we move all elements located in column (row) i to the corresponding positions in column (row) $\pi_\kappa(i)$, for all i.*

A sort-and-unshuffle operation within a row of blocks of length n and height n^β can then be implemented by first sorting the elements in each block of size $n^\beta \times n^\beta$ by their destination blocks, into row-major order, and then performing an $(n^{1-\beta})$-way unshuffle permutation on the columns of the mesh. After this step, the following holds.

Lemma 4.3.2 *Let B_1 and B_2 be any pair of $n^\beta \times n^\beta$ blocks located in the row of blocks, and let D be any destination block. Let N_i denote the number of packets in B_i that have a destination in D, for $1 \leq i \leq 2$. Then we have $|N_1 - N_2| \leq n^{1-\beta}$.*

The above lemma is a special case of Lemma 4.3.1. A corresponding lemma for the case of sorting is presented in Subsection 4.5.2 (see Lemma 4.5.1). The above lemma says that all elements with a common destination block are approximately evenly distributed over all blocks in the row of blocks. By repeating the local sorting of the blocks after the sort-and-unshuffle operation, we could then make sure that all elements with a common destination block are evenly distributed over all columns of the row of blocks, rather than just over all blocks.

4.3.3 The Counter Scheme

As mentioned in the context of the simplified variant of the Valiant/Brebner algorithm, to get a correct deterministic algorithm it is often not possible to simply replace each randomized step by a corresponding deterministic step. In many cases, it is also necessary to modify some of the other, deterministic steps of a randomized algorithm, particularly in the routing of the elements to their final destinations. All of the deterministic algorithms obtained in this chapter have the property that they first route each packet to an approximate destination, and then use local routing to bring each packet to its correct final destination.

More precisely, we partition the network into destination blocks of size $n^\alpha \times n^\alpha$. Every packet is then routed to some position inside the destination block containing its destination address. (In the case of sorting, some packets will actually be routed to neighboring destination blocks.) Once this has been completed, we can then use local routing over a distance of $O(n^\alpha)$ to bring the packets to their final destinations. Algorithms for the local routing problem with a running time of $O(n^\alpha)$ have been described by Kunde [56] and Cheung and Lau [15]. This simplifies the task somewhat, since we only have to be concerned with the problem of moving the packets to their destination blocks.

However, when routing the packets into the destination blocks, we have to make sure that not too many packets enter across the same edge, and that no processor of the block receives too many packets. In a randomized setting, this can be achieved by routing each packet to a random location within its destination block (see, for example, the randomized sorting algorithm of Kaklamanis and Krizanc described in Subsection 4.5.1). In our deterministic algorithms, we will use the counter scheme described in the following.

To explain the idea behind this technique, we consider a routing scheme in which all packets are routed along the columns, until they turn into the rows and enter their destination blocks across the row edges. We assume that, after entering its destination block, each packet keeps on moving in its current direction until it encounters a processor with a free slot in memory. Thus, if we can make sure that all packets with a common destination block are evenly distributed among the incoming rows of the block, then no processor of the block receives too many packets. The counter scheme distributes the packets in each column with a common destination block evenly among the entering rows using a system of *counters*.

In every column, we maintain one counter for each destination block of the mesh. All counters are initially set to zero. Whenever a packet headed for a certain destination block arrives at the location of the corresponding counter, this counter is increased. (More precisely, we have two counters for each destination block, one located above the destination block and counting forward, and one located below the destination block and counting backward.) The new value of the counter, together with a fixed offset value assigned to each counter, determines the row that the packet should choose to enter its destination block. It will be shown that, in the algorithms presented in the next sections, this scheme distributes the packets evenly among the incoming rows of any destination block, provided that we assign an appropriate pattern of offset values to the counters.

4.4 Permutation Routing

In this section we apply the ideas explained in the previous section to a randomized routing algorithm for the two-dimensional mesh proposed by Kaklamanis, Krizanc, and Rao [44]. We obtain a deterministic algorithm with a running time of $2n + o(n)$ and a queue size of 5. We also describe some other applications to two-dimensional tori and three-dimensional meshes.

In the first subsection, we give a brief description of the randomized algorithm in [44]. Subsection 4.4.2 contains the new deterministic algorithm. Finally, Subsection 4.4.3 gives some extensions of the result.

4.4.1 A Simple Randomized Algorithm

We now give a brief description of the randomized algorithm in [44]. Partition the mesh vertically into four quarters Q_0 to Q_3, where Q_i contains the columns $i\frac{n}{4}$ to $(i+1)\frac{n}{4}-1$. Every packet is then first routed along the row to an intermediate destination, where it turns into a column. In this column, the packet moves to its destination row, and then in the destination row to its final destination. The intermediate destination is chosen randomly according to the following rules:

(1) Packets in Q_0 and Q_1 with a destination in Q_0 or Q_1 choose an intermediate position in Q_0.

(2) Packets in Q_0 and Q_1 with a destination in Q_2 or Q_3 choose an intermediate position in Q_2.

(3) Packets in Q_2 and Q_3 with a destination in Q_0 or Q_1 choose an intermediate position in Q_1.

(4) Packets in Q_2 and Q_3 with a destination in Q_2 or Q_3 choose an intermediate position in Q_3.

It is shown in [44] that this routing scheme results in a running time of $2n+O(\lg n)$ and a queue size of $O(\lg n)$, with high probability. (The queue size can be improved to $O(1)$ with some modifications in the algorithm.) An off-line version of the algorithm runs in time $2n-1$ with a queue size of 4.

4.4.2 The Deterministic Algorithm

The high-level structure of our deterministic algorithm is very similar. As in the randomized algorithm, all packets are first routed along the rows to intermediate locations, then along the columns to their destination rows, and finally along the rows to their final destinations. The intermediate locations also satisfy the above four rules, but are now determined by an appropriate unshuffle permutation on the columns of the mesh, rather than being chosen at random. We also need a few additional local steps, and the counter scheme. (Actually, instead of the counter scheme, we could also use an appropriate local sorting step at the end of the column routing in Step (6) of the algorithm.) The deterministic algorithm consists of the following steps.

Algorithm ROUTE:

(1) Partition the mesh into destination blocks of size $n^{\alpha} \times n^{\alpha}$, $2/3 \leq \alpha < 1$, and let every packet determine its destination block.

(2) Partition the mesh into blocks of size $n^{\beta} \times n^{\beta}$, $2/3 \leq \beta < 1$, and sort the packets in each block by their destination blocks, into row-major order. Here, it is assumed that the set of destination blocks is ordered in some arbitrary fixed way, say according to a row-major order of the blocks.

(3) In each quarter Q_i, perform an $(\frac{n^{1-\beta}}{4})$-way unshuffle permutation on the columns.

(4) Route all packets in Q_1 whose destination is in Q_0 or Q_1 into Q_0. Route all packets in Q_0 and Q_1 whose destination is in Q_2 or Q_3 into Q_2. Route all packets in Q_2 and Q_3 whose destination is in Q_0 or Q_1 into Q_1. Route all packets in Q_2 whose destination is in Q_2 or Q_3 into Q_3. The routing is done in such a way that only row edges are used, and that every packet travels a

distance that is a multiple of $n/4$.

(5) Again sort the packets in each $n^\beta \times n^\beta$ block by their destination blocks, into row-major order.

(6) In each column of the mesh, route every packet to a row passing through its destination block. Note that up to this point, we have not yet determined the exact row across which a packet will enter its destination block. This is now done during the column routing, using the counter scheme explained in Subsection 4.3.3. This scheme is described in more depth in the following Step (6a). It will be shown that at most 3 packets turn in any single processor.

(6a) In order to get to its destination block, a packet traveling along its column could turn in any of the n^α consecutive rows passing through that block. To make sure that the row elements are distributed evenly among these rows, we maintain in each column $n^{2-2\alpha}$ counters, two for each of the $\frac{1}{2}n^{2-2\alpha}$ destination blocks in the half of the mesh that contains the column. (Note that all packets are already in the correct half of the mesh before Step (6).) The $n^{1-\alpha}$ counters for any particular row of $\frac{1}{2}n^{1-\alpha}$ destination blocks are located in the $\frac{1}{2}n^{1-\alpha}$ processors immediately above and below the n^α rows passing through these destination blocks. Whenever a row element destined for a particular block arrives at one of the two corresponding counters, this counter is either increased by one, modulo $2n^{2\alpha-1}$ (in the case of the counters above the destination rows), or decreased by one, modulo $2n^{2\alpha-1}$ (in the case of the counters below the destination rows). The row across which the packet will enter its destination block is determined by the sum, modulo n^α, of the new counter value and a fixed offset value associated with each counter. A counter in column i of the half, $0 \leq i < n/2$, that corresponds to a destination block in the jth column of destination blocks in this half of the mesh,

$0 \leq j < \frac{1}{2}n^{1-\alpha}$, is assigned the offset value $(i + j \cdot 2n^{2\alpha-1}) \bmod n^{\alpha}$.

(7) Route the packets along the rows into their destination blocks in a greedy fashion, giving priority to the element with the farther distance to travel. After entering its destination block, a packet will stop at the first processor that has a free memory slot for an additional packet. Here, we say that a processor has a free slot if it currently holds less than 3 packets that are not just passing through the processor. Due to the counter scheme in Step (6a), the incoming packets are evenly distributed over the rows of any destination block.

(8) Perform local routing over a distance of $O(n^{\alpha})$ to bring every element to its final destination.

Let us first analyze the running time of the above algorithm. Clearly, each of the Steps (1), (2), (5), and (8) only take time $o(n)$. Step (3) and Step (4) can be overlapped as follows. Rather than first performing the unshuffle operation in Step (3), and then doing the overlapping in Step (4), we can send the packets directly to the locations they will assume after Step (4). This means that all blocks in Q_0 and Q_3, as well as those blocks in Q_1 and Q_2 that are close to the center column, have received all of their elements by time $0.5n+o(n)$, while it takes up to time $0.75n+o(n)$ for the other blocks in Q_1 and Q_2 to receive all of their packets. As soon as a block has received all of its packets, it can perform the local sort in Step (5), and start with the column routing in Step (6). This routing problem is an approximate 2–2 relation on a linear array, and can hence be routed in $n+o(n)$ steps (see [44]). Thus, Step (6) of the algorithm will terminate between time $1.5n+o(n)$ and $1.75n+o(n)$, depending on the location of the column in the mesh. Assuming that Step (6a) has distributed the packets evenly over the incoming rows of each destination block, Step (7) can be interpreted as the problem of routing an approximate 2–1 relation on a linear array of length $n/2$, where packets that have a distance of d to travel are not allowed to move before time $n/2 - d$. This routing process is started at time $1.5n + o(n)$ and terminates at time $2n + o(n)$. Thus, the above algorithm runs in time $2n + o(n)$.

It remains to show that the packets are indeed evenly distributed over the incoming rows of each destination block, and that the total queue size is bounded by 5. Consider a destination block D and two $n^\beta \times n^\beta$ blocks B_1 and B_2 located in the same quarter and the same row of blocks. Lemma 4.3.2 says that the number of packets with destination block D will differ by at most $n^{1-\beta} = o(n^\alpha)$ between B_1 and B_2, after Step (4). This implies that after Step (5), the number of packets with destination block D will differ by at most $2n^{1-\beta}$ between any two columns in the quarter. There are at most $n^{2\alpha}$ packets with destination block D in the quarter. Hence, any of the $\frac{n}{4}$ columns in the quarter can contain at most

$$\frac{n^{2\alpha}}{\frac{n}{4}} + 2n^{1-\beta} =\sim 4n^{2\alpha-1}$$

packets with destination block D, which are evenly distributed among $2n^{2\alpha-1}$ rows by the counter technique (up to a difference of 1). (Due to the assignment of offset values to the counters, packets with different destination blocks always turn in different processors.) This implies that at most 3 packets turn in any single processor. If we limit our attention to a single column, then all packets with destination block D in that column are distributed over only a small fraction of the incoming rows of D. However, if we look at blocks of n^α consecutive columns, then the elements with destination block D in these columns are evenly distributed among all incoming rows of D, due to the n^α different offset values of the $2n^\alpha$ counters corresponding to D. This implies that the routing in Step (7) terminates in such a way that every processor of D receives at most 3 packets.

The maximum possible queue size in Step (6) of the algorithm is given by a scenario in which 3 packets have to turn in a given processor, while 2 other packets are temporarily passing through the processor. The maximum queue size during

Step (7) is also 5; all other steps achieve even smaller queue sizes.

One issue we have ignored so far is that a packet may already be located in a row passing through its destination block before Step (6). Such a packet will not pass any counter on its way along the column. We can assign destination rows to these packets before the start of the column routing, and set the initial values of the counters accordingly. This can be done locally during Step (5) of the algorithm. Altogether, we have shown the following result.

Theorem 4.4.1 *There exists a deterministic routing algorithm for two-dimensional meshes with a running time of* $2n + o(n)$ *and a queue size of* 5.

For $\alpha = \beta = 2/3$, the running time of the algorithm is $2n + O(n^{2/3})$. A modified version of the algorithm runs in time $2n + O(n^{1/2})$.

Kaklamanis, Krizanc, and Rao [44] also give a randomized algorithm that routes any 2–2 relation in time $2n + o(n)$, and a corresponding off-line scheme with a running time of $2n$ and a queue size of 8. For the deterministic case, we can show the following result.

Lemma 4.4.1 *Any 2–2 relation can be routed deterministically in time* $2n + o(n)$ *with a queue size of* 10.

The algorithm proceeds as follows. First, we partition the packets into two sets such that all packets with a common destination block are evenly divided between the two sets. This can be done deterministically by sorting the packets in each block of size $n^\beta \times n^\beta$ by destination blocks, and taking the two sets as the packets with odd and even ranks, respectively. We then route both sets simultaneously, using the deterministic algorithm given above. One of the sets will be routed on row-column-row paths, and the other one on column-row-column paths. Due to the overlap between the three phases of the algorithm, it is possible that packets in different phases of the algorithm contend for the same edge. These contentions will be resolved by giving priority to the packet in the lower numbered phase. In [44], Kaklamanis, Krizanc, and Rao show that their randomized algorithm routes any 2–2 relation in time $2n + o(n)$, with high probability. It can be checked that their proof also extends to our deterministic algorithm.

4.4.3 Extensions

Kaklamanis, Krizanc, and Rao also give optimal randomized and off-line algorithms for tori and three-dimensional meshes. In this subsection, we give similar extensions for the deterministic case. The first extension, an optimal algorithm for three-dimensional meshes, is achieved by a reduction to the problem of routing a 2–2 relation on a two-dimensional subnetwork, described in [44]. Together with Lemma 4.4.1, this gives the following result.

Theorem 4.4.2 *There exists a deterministic algorithm for permutation routing on the three-dimensional mesh with a running time of* $3n + o(n)$ *and a queue size of* 13.

The fastest deterministic algorithm previously known for this problem has a running time of $(3 + \frac{1}{3})n$ and is due to Kunde [57]. Our approach can also be used to obtain deterministic algorithms for routing in d-dimensional meshes with $d > 3$. Using the unshuffle operation and the counter scheme, we can convert the randomized algorithm of Valiant and Brebner [117] into a deterministic algorithm with a running time of $(2d-1)n + o(n)$. This can be improved to $(2d-3)n + o(n)$ by using the above algorithm for three-dimensional meshes as a subroutine. For $d = 4$, this gives a slight improvement over the fastest previously known algorithm [57], which achieves a running time of $(5 + \epsilon)n$ and a queue size of $O(1/\epsilon)$. However, for larger values of d, this approach does not give an improvement over Kunde's results. Algorithms for multi-dimensional meshes with significantly better running times are presented in the next chapter of this thesis.

In [44], Kaklamanis, Krizanc, and Rao give a second optimal randomized algorithm for the two-dimensional mesh that has a slightly simpler structure than the one described in the previous subsection. As before, all packets are routed on row-column-row paths. A packet that originates in column i and whose destination is in column i' chooses its intermediate column uniformly at random from all l with $|l-i|+|l-i'| \leq n-1$. If several packets contend for an edge, priority is given to the packet with the farther distance to travel. Using the techniques of this chapter, it is not difficult to convert this algorithm into a deterministic algorithm with a running time of $2n + o(n)$ and a queue size of 6.

Finally, Kaklamanis, Krizanc, and Rao give an optimal randomized algorithm for the two-dimensional torus that has a very similar structure. In this algorithm, one half of the packets is routed on row-column-row paths, and the other half on column-row-column paths. A packet that is routed on a row-column-row path, and that originates in column i and is destined for column i', chooses its intermediate column uniformly at random from all l with $|l-i|+|l-i'| \leq \frac{n}{2}-1$. The case of the packets that are routed on column-row-column paths is symmetric. If several packets contend for an edge, priority is given to the packet with the farther distance to travel. This algorithm can also be converted into a deterministic one, and we obtain the following theorem. (The exact queue size of the algorithm is between 10 and 20.)

Theorem 4.4.3 *There exists a deterministic algorithm for permutation routing on the $n \times n$ torus with a running time of $n + o(n)$ and constant queue size.*

4.5 Optimal Deterministic Sorting

In this section, we apply the techniques described in the previous sections to a class of randomized sorting algorithms proposed by Kaklamanis and Krizanc [42]. We obtain the first optimal deterministic sorting algorithm for two-dimensional meshes, as well as improved deterministic algorithms for the three-dimensional mesh and the two-dimensional and three-dimensional torus.

In the first subsection, we give a description of the randomized algorithm in [42]. In Subsection 4.5.2 we describe the modifications required to convert this randomized algorithm into a deterministic one. Subsection 4.5.3 contains the deterministic algorithm and a proof of the claimed bounds on time and queue size. Finally, Subsection 4.5.4 gives a few extensions.

4.5.1 An Optimal Randomized Algorithm

In this following we give a high level description of a randomized algorithm with running time $2n + o(n)$ and constant queue size proposed by Kaklamanis and Krizanc [42]. Their algorithm is based on an earlier $2.5n + o(n)$ time algorithm of Kaklamanis, Krizanc, Narayanan, and Tsantilas [43]. The complete structure of the algorithm is quite complicated, and so our description will necessarily ignore a number of important details. For a full description the reader is referred to [42].

Our description of the algorithm uses a slightly different numbering of the steps than the original description. The mesh is divided into four quadrants Q_0, Q_1, Q_2, and Q_3. The four quadrants are again divided into a total of 16 subquadrants, labeled T_0 to T_{15}. We assume that the four subquadrants located around the center are labeled T_0 to T_3. In addition, a block B of side length $o(n)$ around the center of the mesh is used to sort the sample elements and select the splitters.

Algorithm RANDOMSORT:

(1) Select a random sample set S of size $o(n)$ from the n^2 elements using coin flipping.

(2) Each sample element picks a random location in the block B at the center of

the mesh, and routes a copy of itself greedily towards that location. To make sure that the routing is completed in n steps, we give the sample elements priority over all other elements.

(3) Each of the n^2 packets in the mesh flips a coin, and, depending on the outcome, declares itself either a *row element* or a *column element*.

(4) Each row element selects a random location between 0 and $n/4-1$ in its row, inside its current subquadrant. Similarly, each column element selects a random location between 0 and $n/4-1$ in its column. Note that in this step, the elements do not actually go to their selected destination. Thus, Step (4) takes time $o(n)$.

(5) Now copies of each element are routed to the four locations in the middle subquadrants T_0 to T_3 that correspond to the locations randomly selected in Step (4). This means that each of the four subquadrants T_0 to T_3 receives copies of all n^2 elements in the mesh.

(6) The sample set is sorted in the center block B, and n^δ elements of equidistant ranks are chosen as splitters. This takes time $o(n)$.

(7) The n^δ splitters are broadcast in the middle subquadrants T_0 to T_3. During the broadcast, the global ranks of the splitters are computed using a pipelined prefix computation that counts, for each splitter, the number of elements that are smaller. The results of this computation arrives at the center points of the four quadrants $0.5n+o(n)$ steps after they were sent out.

(8) Each element, upon receiving the splitter elements broadcast from the center of the mesh, can determine its rank to within $O(n^{2-\delta})$, the accuracy of the splitters. From this approximate rank, the element can compute the block of side length $O(n^{1-\delta/2})$ most likely to contain its final destination. If this block

is outside its current quadrant, the element kills itself. Otherwise, it selects a random location within this block.

(9) All surviving elements route themselves to the chosen location. The routing is done in a greedy fashion, where row elements first route along their column to the correct row, while column elements first route along their row to the correct column. However, a slightly more complicated priority scheme than the usual "farthest distance to travel first" is required in this routing step. The same priority scheme is also employed in our deterministic algorithm; a description of this scheme is given in the proof of Lemma 4.5.5. It can be shown that every element reaches its approximate destination within time $n + o(n)$ after the splitters were broadcast from the center of the mesh.

(10) The exact ranks of the splitter elements are broadcast in each quadrant, starting at the center of the quadrant after completion of Step (7). Hence, every element receives the exact splitter ranks within $n+o(n)$ steps after the splitters were broadcast from the center.

(11) Now local routing over a distance of $O(n^{1-\delta/2})$ can be used to bring each element to its final location in time $o(n)$.

The above algorithm can be scheduled in time $2n + o(n)$. For a more complete description of the algorithm, and a proof of the stated time bounds, we refer the reader to the paper by Kaklamanis and Krizanc [42]. Here, we only add the following remarks considered important in the present context.

- The algorithm sorts with respect to an indexing scheme with the property that processors whose indices differ by $O(n^{2-\delta})$ are at most $O(n^{1-\delta/2})$ steps apart. If this condition is not satisfied, as, for example, in row-major indexing, then the elements will not be able to compute good approximate destinations from their approximate ranks in Step (8).

- One of the purposes of the randomization in Steps (2),(3), and (4) is to get a good bound on the queue size. However, randomization alone will only guarantee a queue size of $O(\lg n)$ with high probability. To reduce the queue size to a constant, the algorithm uses a packet redistribution technique described in [99] and attributed to Leighton.

- The routing in Step (5) of the algorithm is done according to a rather ingenious schedule described in [42]. In this schedule, the row elements and column elements of a subquadrant may move along different paths. However, all row elements (column elements) of a subquadrant move in lock step until they enter their destination subquadrant. The routing to the random locations selected in Step (4) is done either before the elements start to move according to the schedule, or upon entering the destination subquadrant, or after they have already reached the destination subquadrant. While we will not go into the details of this routing schedule, it is nonetheless important to realize that Step (5) is deterministic, since the random locations of the elements were already chosen in the preceding step. The routing in Step (5) would work equally well if those destinations had been chosen according to some deterministic strategy. Hence, we will be able to use this schedule in our deterministic algorithm without modification.

- Finally, note that the routing in Step (5) has to take at least $1.25n$ steps, and thus will not be completely finished when the set of splitters is broadcast at time $n + o(n)$. However, it can be shown that all elements reach their destination before the arrival of the splitter front.

4.5.2 Getting a Deterministic Algorithm

In this subsection we explain the modifications that have to be made in the randomized algorithm described in the previous subsection in order to get an optimal deterministic algorithm. The randomized algorithm uses randomization in a number of different phases, and for a number of different purposes, which are informally described in the following.

- Randomization is used in Step (1) of the algorithm to select a sample set that, with high probability, yields a set of "good", that is, roughly evenly spaced, splitters. In this subsection, we describe a deterministic sampling technique that guarantees such a set of "good" splitters, and which can be substituted for the randomized sampling in Step (1).

- In Step (3), elements use a coin flip to identify themselves as either row elements or column elements. The effect of this coin flipping technique is that, with high probability, about half of the elements become row elements (resp. column elements), and that the set of row elements (resp. column elements) is spread out evenly over the range of input values. This can be achieved deterministically by sorting locally and taking the elements with even ranks as row elements, and the elements with odd ranks as column elements, as in the algorithm underlying Lemma 4.4.1.

- In Step (4), every row element chooses a random position in its row inside its subquadrant, and every column element chooses a random position inside its column. This has the effect that, with high probability, the row elements (column elements) of similar rank and, hence, similar final destination, are evenly distributed among the columns (rows) of their subquadrant. This is needed in Step (9) of the algorithm to make sure that the routing of the elements to their destination blocks is finished within the required time bounds and with constant queue size. The effect of this randomization step will be "simulated" with the sort-and-unshuffle operation described in Section 4.3.

- Finally, in Step (8) every element selects a random location within its destination block. Here, randomization is used to assure that not too many elements route themselves to the same location in their destination block. As demonstrated in the previous section, this can be achieved deterministically by using the counter scheme.

As in the routing algorithm of Subsection 4.4.2, we divide the mesh into blocks of size $n^\beta \times n^\beta$, with $\frac{2}{3} < \beta < 1$. When applying the unshuffle operation to simulate Step (4) of the randomized algorithm, we sort the row elements (column elements) in each block into row-major (column major) order, and then perform an $(\frac{n^{1-\beta}}{4})$-way unshuffle permutation on the columns (rows) of each subquadrant. The effect of this operation is described in the following lemma, which follows directly from Lemma 4.3.1.

Lemma 4.5.1 *Let B_1 and B_2 be any pair of $n^\beta \times n^\beta$ blocks located in the same row (column) of blocks of some subquadrant T_i, $0 \leq i \leq 15$, and let A be any set of consecutive values in $[n^2]$. Let N_j denote the number of elements in B_j whose global rank among all n^2 elements is in A, for $1 \leq j \leq 2$. Then we have $|N_1 - N_2| \leq \frac{n^{1-\beta}}{4}$.*

To simulate the effect of Step (3) of the randomized algorithm, we sort each block of size $n^\beta \times n^\beta$, and label all elements with even ranks as row elements, and all elements with odd ranks as column elements. We remark that this technique is closely related to the unshuffle operation. More precisely, the following analogue of Lemma 4.5.1 holds.

Lemma 4.5.2 *Let A be any set of consecutive values in $[n^2]$, and let the number of row elements and column elements whose global rank among all n^2 elements is in A be denoted by N_r and N_c, respectively. Then we have $|N_r - N_c| \leq n^{2-2\beta}$.*

The last ingredient needed for our deterministic algorithm is a deterministic sampling technique that results in a set of "good" splitter elements. Our technique is essentially a simplified version of a more sophisticated sampling technique used in the parallel selection algorithm of Cole and Yap [20]. Our goal is to deterministically select a set of approximately evenly spaced splitters from a set of keys X of cardinality n^2. More precisely, we are interested in selecting a set of splitter elements $D = \{d_0, \ldots, d_{t-1}\}$ with $d_{i+1} > d_i$, such that the following property holds for all i:

(1) $\frac{(i-1)n^2}{t} \leq \text{Rank}\,(d_i, X) \leq \frac{in^2}{t}$

To achieve this, we will our sample set using the following two steps:

(i) Partition the mesh into blocks of size $n^\delta \times n^\delta$, $\frac{2}{3} < \delta < 1$, and sort the elements in each block.

(ii) Select n^δ equidistant elements from each sorted block as sample elements, starting with the smallest element and going up to the (n^δ)th largest element. If the elements were sorted into row-major order in the first step, then we can simply select the elements in the first column of each block.

The sample set selected in the above two steps contains $n^{2-\delta}$ elements, which are routed to the center of the mesh and sorted. We claim that the global rank of each sample element can now be computed to within an additive term of $n^{2-\delta}$. More precisely the following lemma holds.

Lemma 4.5.3 *Let S be a sample set of size $n^{2-\delta}$ chosen from a set X of size n^2 in the manner described above. Then for any $s \in S$ with $Rank(s, S) = i$ we have*

$$(i + 1 - n^{2-2\delta}) \cdot n^\delta < Rank(s, X) < (i + 1) \cdot n^\delta.$$

Proof: Let X_i denote the set of elements in block i of the mesh, $0 \leq i < n^{2-2\delta}$. Partition the sample set S into $n^{2-2\delta}$ subsets S_i, $0 \leq i < n^{2-2\delta}$, where each S_i consists of those elements of S that were drawn from subset X_i in the first phase of the sampling algorithm. Now associate with each $s \in S_i$ the set $T(s)$ consisting of all elements $x \in X_i$ with $s \leq x < s'$, where s' is next larger sample element drawn from the same subset X_i. Note that this defines a partition of the input set X, and that each of the $n^{2-\delta}$ sets $T(s)$ contains exactly n^δ elements.

Now let $s_1 \in S_i$ and $s_2 \in S_j$ be two arbitrary sample elements. If $s_1 < s_2$, then every element of $T(s_2)$ must be larger than s_1. There are $|S| - \text{Rank}\,(s_1, S) - 1$ elements s_2 with $s_1 < s_2$ in S; hence $\text{Rank}\,(s_1, X) < (\text{Rank}\,(s_1, S) + 1) \cdot n^\delta$. If $s_2 \le s_1$, then we have the following two cases:

(a) If s_2 is the largest element in S_j with $s_2 \le s_1$, then all elements in $T(s_2)$, except for s_2 itself, can be either smaller or larger than s_1.

(b) If s_2 is not the largest element in S_j with $s_2 \le s_1$, then all elements in $T(s_2)$ must be smaller than s_1.

Note that there are $\text{Rank}\,(s_1, S) + 1$ elements $s_2 \in S$ with $s_2 \le s_1$, and at most $n^{2-2\delta}$ of these fall under case (a), including s_1 itself. Hence, at least $(\text{Rank}\,(s_1, S) + 1 - n^{2-2\delta}) \cdot n^\delta$ elements in X are smaller than s_1.

□

The following theorem establishes a way of selecting a set of "good" splitters from the sample. It can be proved by a simple application of the above lemma.

Theorem 4.5.1 *Let D be the splitter set of size n^δ consisting of all $s \in S$ with Rank $(s, S) = i \cdot n^{2-2\delta}$, for some nonnegative integer i. Then D is a set of "good" splitters, that is, it satisfies Property (1) stated above.*

Note that, while the sample set contains $\omega(n)$ elements, the splitter set selected from the sample is of size $o(n)$. The latter fact will be used in the step of our sorting algorithm where the entire splitter set is broadcast to every packet in the mesh.

4.5.3 The Deterministic Algorithm

In the following description of the deterministic sorting algorithm, we maintain the numbering of the steps used in the randomized algorithm. Some of the steps in the algorithm can be taken directly from the randomized algorithm, but others have to be substantially changed. The algorithm sorts with respect to the blocked indexing defined in Section 4.1, where the size of the blocks in the indexing is n^α, for some constant α. The size of the sample and splitter sets is determined by a constant δ, already used in the description of the sampling technique in the previous subsection. Finally, we have to choose a constant β that determines the size of the blocks used by the unshuffle operation. These constants have to be chosen such that $\frac{2}{3} < \alpha, \beta, \delta < 1$.

Algorithm SORT:

(1) Select a sample set of size $n^{2-\delta}$ by sorting blocks of size $n^\delta \times n^\delta$ and taking the first column in each block. This takes time $O(n^\delta) = o(n)$.

(2) Route a copy of the sample set to a block B of size $n^{1-\delta/2} \times n^{1-\delta/2}$ at the center of the mesh. This can be completed in n steps; the details of this routing step are given in the proof of Lemma 4.5.4.

(3) Divide the n^2 elements into $n^2/2$ *row elements* and $n^2/2$ *column elements* as described in Subsection 4.5.2. This operation takes time $O(n^\beta) = o(n)$.

(4) In each block of size $n^\beta \times n^\beta$, sort the row elements into row-major order. Now select for each row element a new location in its row, within its current subquadrant, corresponding to an $(\frac{n^{1-\beta}}{4})$-way unshuffle permutation on the columns, as described in Section 4.3. Similarly, sort the column elements in each block into column-major order, and select new locations according to an $(\frac{n^{1-\beta}}{4})$-way unshuffle permutation on the rows. Again, as in the randomized algorithm, the elements will not actually move to the chosen locations in this

step. This will be done in Step (5).

(5) This step is the same as in the randomized algorithm. We route copies of each element to the four locations in the middle subquadrants T_0 to T_3 that correspond to the locations chosen in Step (4). This step takes time $1.25n$, but every copy reaches its location before the arrival of the splitter elements.

(6) This step is also the same as in the randomized case. The sample set is sorted in the center block B, and n^δ elements of equidistant ranks are chosen as splitters. This takes time $O(n^{1-\delta/2}) = o(n)$, and Theorem 4.5.1 guarantees that every splitter can determine its global rank to within $O(n^{2-\delta})$.

(7) This step is again the same as in the randomized algorithm. The splitters are broadcast in each of the subquadrants T_0 to T_3, and the exact global ranks of the splitter elements are computed. This takes time $0.5n$.

(8) Each element hit by the splitter front can determine its rank to within a range of $O(n^{2-\delta})$ ranks. This enables the element to determine the block of side length n^α that will contain most of the elements within this range in the final sorted order. If that block is outside its current quadrant, then the element kills itself. Note that an element may actually not end up in this block in the final sorted order, but the properties of our indexing scheme guarantee that the chosen block will be close to its final destination. Now, before routing the elements to their approximate destinations, we perform the following additional step:

(8a) Divide the mesh into blocks of size $n^\beta \times n^\beta$. As soon as such a block has been completely traversed by the splitter front, the row elements in the block are sorted into row-major order by their $n^\alpha \times n^\alpha$ destination blocks, where the ordering of the destination blocks can be arbitrary. Similarly, the column elements in the block are sorted into column-major order by destination blocks. The purpose of this step is to distribute the row (column) elements with a common destination block evenly among the columns (rows) of the $n^\beta \times n^\beta$ block.

Note that this step takes time $O(n^\beta) = o(n)$ per block, from the moment the splitter front enters the block until the sorting of the row and column elements in the block is completed. Thus, we can initiate the routing in the following Step (9) by broadcasting a *Start* signal from the center of the mesh $O(n^\beta)$ steps after the broadcast of the splitter set.

(9) After the arrival of the *Start* signal, every element routes itself greedily towards its destination block. Row elements go first along the columns until they reach their destination row, and column elements travel first along their row until they reach their destination column. We can employ the same priority scheme that is used in the randomized algorithm. Note that up to this moment, the exact destinations of the elements inside their destination blocks have not yet been determined. This will be done during the routing, in the following Step (9a). It will be established in Lemma 4.5.5 that the routing terminates in $n + o(n)$ steps with constant queue size. A more detailed description of the routing is given in the proof of the lemma.

(9a) Use the counter scheme described in the routing algorithm in Subsection 4.4.2 to distribute the elements evenly over the rows and columns of the destination blocks. (Alternatively, this could also be achieved by interleaving the routing in Step (9) with a non-constant number of local sorting steps, such that the total time spent on local sorting is $o(n)$. The same idea could also be applied in the case of the routing algorithm for the torus discussed in Subsection 4.4.3.)

(10) This step is the same as in the randomized algorithm. The exact ranks of the splitter elements are broadcast from the center of each quadrant $0.5n$ steps

after the splitters were sent out from the center. After another $0.5n$ steps, all elements have received the splitter ranks.

(11) We now perform local routing over a distance of $O(n^\alpha)$ to bring each element to its final destination. This takes time $O(n^\alpha)$.

Our claim is that this algorithm runs in time $2n+o(n)$ with constant queue size. The exact bound for the queue size is at most 25; we elaborate on this issue briefly in the proof of Claim 5 in Appendix A. We establish our result in the following two lemmas.

Lemma 4.5.4 *The sample set of size $n^{2-\delta}$ selected in Step (1) can be routed in n steps to a block of size $n^{1-\delta/2} \times n^{1-\delta/2}$ around the center of the mesh, without delaying the routing in Step (5) by more than $o(n)$ steps.*

Proof: Since our sample set is of size $\omega(n)$, we have to be a bit careful in the design of this routing step to make sure that the movement of the splitters towards the center does not delay the movement of the packets in Step (5). We propose the following solution. After Step (1), all elements in the sample set are located in the first column of their respective $n^\delta \times n^\delta$ block. Now move all sample elements located in a block that is in the ith row of blocks into the ith column of that block, for $i = 1, \ldots, n^{1-\delta}$. This can be done in $o(n)$ time by locally routing inside each block. Now use column routing to move all sample elements to the n^δ middle rows of the mesh. This is completed in $0.5n$ steps. Next, we use row routing to move the sample elements into the block in the center, which takes another $0.5n$ steps. Observe that in the routing we have only used edges in $n^{2-2\delta} = o(n)$ columns and $n^\delta = o(n)$ rows of the mesh. Hence, we can guarantee that the routing of Step (5) is delayed by at most $o(n)$ steps by simply reserving these edges for the sample elements, and restricting the other packets to the remaining rows and columns. (It can be shown that this restriction is not really necessary.)

□

Lemma 4.5.5 *The greedy routing to destination blocks in Step (9) runs in time $n + o(n)$ with constant queue size.*

The proof of Lemma 4.5.5 is given in Appendix A. Together, Lemma 4.5.4 and Lemma 4.5.5 establish the following result.

Theorem 4.5.2 *There exists a deterministic algorithm for sorting on the $n \times n$ mesh with running time $2n + o(n)$ and constant queue size.*

It is not difficult to see that the above algorithm still works if we sort with respect to a slightly different indexing scheme, in which the blocks of size $n^\alpha \times n^\alpha$ are ordered along the diagonals rather than along the rows. This is somewhat interesting in that there exists a lower bound of $4n - o(n)$ in the single-packet model for this modified indexing scheme. Thus, an indexing scheme that is "good" for the multi-packet model may not be "good" at all for the single-packet model.

4.5.4 Extensions

In [42], Kaklamanis and Krizanc extend their results to three-dimensional meshes and two-dimensional and three-dimensional tori. These extensions also hold for the deterministic case, and we get the following results.

Theorem 4.5.3 *There exists a deterministic algorithm for sorting on the three-dimensional mesh with running time $3.5n + o(n)$ and constant queue size.*

Theorem 4.5.4 *There exists a deterministic algorithm for sorting on the two-dimensional torus with running time $1.25n + o(n)$ and constant queue size.*

Theorem 4.5.5 *There exists a deterministic algorithm for sorting on the three-dimensional torus with running time $2n + o(n)$ and constant queue size.*

The best deterministic algorithms previously known for these problems required running times of $5n + o(n)$, $2n + o(n)$ and $3n + o(n)$, respectively. Using the above algorithms for three-dimensional meshes and tori as subroutines, we can obtain improved algorithms for sorting on d-dimensional meshes and tori, $d \geq 4$, with running times of $(2d - 2.5)n + o(n)$ and $(d - 1)n + o(n)$, respectively. The best deterministic algorithms previously known for these networks required $(2d - 1)n$ steps on the mesh and $dn + o(n)$ on the torus [55]. In the next chapter, we present algorithms that significantly improve on these bounds, for all $d > 5$.

4.6 Some Other Applications

The techniques described in this chapter can also be applied to a number of other randomized algorithms. In the following, we briefly describe a few of these applications.

Optimal algorithms for k–k sorting on meshes and tori have recently been proposed in [47] and [59]. The two algorithms are very similar, and they can both be seen as an efficient implementation of Leighton's Columnsort [64] on the mesh. The algorithm in [47] is obtained by "derandomizing" a randomized algorithm in [46], and provides a particularly simple and elegant application of the sort-and-unshuffle operation. The result also indicates an interesting relationship between Columnsort and certain classes of simple randomized sorting algorithms.

Kaufmann, Meyer, and Sibeyn [45] have recently reported a routing algorithm with a running time of $2n + O(1)$ and an internal queue size of 2, based on the randomized algorithm of Kaklamanis, Krizanc, and Rao [44] that was considered in Subsection 4.4.3. (This translates into a queue size of at most 6 in our model.) This improves on other recent algorithms [17, 106] that are based on the approach of Leighton, Makedon, and Tollis [69].

Our techniques can also be used to derive faster deterministic algorithms for meshes with additional (non-reconfigurable) row and column buses, based on a class of randomized algorithms described by Sibeyn, Kaufmann, and Raman [107]. See Chapter 6 for a definition of these networks.

Another application leads to faster deterministic algorithms for selection on meshes and tori [111]. The fastest randomized selection algorithm currently known runs in $1.15n$ steps [22]. Using the techniques in this chapter, this bound can be matched deterministically.

In our presentation, we have only considered those applications that lead to an improvement over the best previous deterministic results. However, many other randomized algorithms in the literature can also be converted into deterministic ones, including the "original" randomized algorithm for the mesh by Valiant and Brebner [117] and the algorithm of Rajasekaran and Tsantilas [99].

4.7 Conclusion

In this chapter, we have described a set of techniques that allows us to "derandomize" (in an informal sense) many randomized algorithms for routing and sorting on meshes that have been proposed in recent years. By applying these techniques, we can obtain optimal or improved deterministic algorithms for a number of routing and sorting problems on meshes and related networks. The techniques are very general and seems to apply to most of the randomized algorithms that have been proposed in the literature. In fact, as a result of this work, we are currently not aware of any randomized algorithm for permutation routing or sorting on meshes or related networks whose running time cannot be matched, within a lower order additive term, by a deterministic algorithm. (Of course, this claim does not hold for more restricted models of the mesh, e.g., models that only allow *hot-potato routing*, or that disallow local sorting steps.)

This raises the question whether randomization is of any help at all in the design of routing and sorting algorithms for the type of theoretical mesh model assumed in this chapter. In this context, three important points have to be made.

- Many of the randomized algorithms still have a simpler control structure and smaller lower order terms than their deterministic counterparts, which repeatedly perform local sorting. In fact, the results of this chapter could be interpreted as saying that randomization is an efficient way of avoiding such local sorting steps.
- The results in this chapter would not have been possible without the extensive study of randomized schemes for routing and sorting by a number of authors, which has resulted in a variety of fast randomized algorithms [42, 43, 44, 46, 99, 117]. Thus, randomization can also be seen as a useful tool in the design of deterministic algorithms.
- Finally, most randomized routing algorithms, including those in [44], can also be applied to dynamic routing problems, in which packets are continuously generated during an ongoing computation. In contrast, the deterministic algorithms obtained with our techniques cannot be easily adapted to the dynamic case, due to the local sorting steps.

It is an interesting question whether the ideas described in this chapter may also be useful for other classes of networks, and perhaps even other types of problems. Of course, as presented the techniques are only efficient on networks with large diameter, since we repeatedly sort fairly large subsets of the input. A straightforward application to networks with small diameter, such as the hypercubic networks, would lead to a blow-up in the running time due to the time spent on local sorting.

Subsequent to this work, Sibeyn [105] has reported an optimal algorithm for sorting into row-major order. The algorithm achieves a queue size of 5, and is significantly simpler than the one described in Section 4.5. However, a number of questions still remain open, such as the existence of an optimal sorting algorithm that does not make any copies, or the exact complexity of the selection problem on the mesh.

Another open problem is the complexity of permutation routing and 1–1 sorting on multi-dimensional meshes. The best algorithms currently known are still nearly a factor of 2 away from the diameter lower bound. We will focus on this problem in the next chapter of this thesis

In this subsection, we have only discussed open questions that are directly related to the results of this chapter. Some other suggestions for future work on meshes can be found in Chapter 7.

Chapter 5

Bounds for Multi-Dimensional Meshes

This chapter establishes improved bounds for 1–1 routing and sorting on multi-dimensional meshes and tori. In particular, we give a fairly simple deterministic algorithm for sorting on the d-dimensional mesh of side length n that achieves a running time of $3dn/2 + o(n)$ without making any copies of the elements. We give deterministic algorithms with running times of $5dn/4+o(n)$ and $3dn/4+o(n)$ for the d-dimensional mesh and torus, respectively, that make one copy of each element. We also show lower bounds for sorting with respect to a large class of indexing schemes, under a model of the mesh where each processor can hold an arbitrary number of packets. Finally, we describe algorithms for permutation routing whose running times come very close to the diameter lower bound.

5.1 Introduction

Much of the previous work on mesh routing and sorting has concentrated on the one-dimensional and two-dimensional cases, while the meshes of dimension $d > 2$

(hereinafter referred to as *multi-dimensional meshes*) have received somewhat less attention. Although the problems of routing and sorting on these networks have previously been studied by a number of authors, there are still considerable gaps between the best upper and lower bounds.

In this chapter, we focus on the problems of 1–1 routing and sorting on multi-dimensional meshes with constant dimension d. Recall that a d-dimensional mesh of side length n consists of $N = n^d$ processors, where each processor is identified by a d-tupel $(p_1, \ldots, p_d)$ in $[n]^d$. Two processors $P = (p_0, \ldots, p_{d-1})$ and $Q = (q_0, \ldots, q_{d-1})$ are connected by a bidirectional communication link iff there exists an i in $[d]$ with $|p_i - q_i| = 1$ and $p_j = q_j$ for all j in $[d]$ with $j \neq i$. The d-dimensional torus is obtained from the d-dimensional mesh by adding *wrap-around* edges between all pairs of processors $(p_0, \ldots, p_{d-1})$ and $(q_0, \ldots, q_{d-1})$ such that there exists an i in $[d]$ with $|p_i - q_i| = n - 1$ and $p_j = q_j$ for all j in $[d]$ with $j \neq i$.

In Section 4.1, we defined the *row-major* and *snake-like row-major* indexing schemes for the two-dimensional mesh. Both of these schemes can be naturally extended to multi-dimensional networks; see [58] for a formal definition. In this chapter, as in the preceding one, we assume a *blocked* indexing scheme. The indexing scheme is defined by partitioning the mesh into d-dimensional blocks of side length n^α, for some $\alpha < 1$, and using a snake-like indexing scheme inside each block, while the blocks themselves are ordered according to another snake-like indexing. Blocked indexing schemes have recently been used in a number of fast sorting algorithms (e.g., see [42, 43, 47, 59]).

An obvious lower bound for the running time of any algorithm for 1–1 routing or sorting is given by the diameter D of the network. (That is, $D = d(n-1)$ for the d-dimensional mesh and $D = dn/2$ for the d-dimensional torus.) The performance of an algorithm for routing or sorting on theoretical models of the mesh is commonly measured by its running time, its *queue size* (that is, the maximum number of packets any node has to store during the algorithm), and of course by factors such as the overall simplicity of the algorithm and the demands it places on the local control hardware or software. In the following, we focus on the time complexity of routing and sorting on meshes and tori of arbitrary constant dimension, and we express our results in terms of the diameter D of the network. All presented algorithms have a queue size of $O(d)$.

5.1.1 Previous Results

Early examples of sorting algorithms for multi-dimensional meshes were given by Thompson and Kung [114] and Nassimi and Sahni [82]. A lower bound of $2D - n - o(n)$ for sorting on multi-dimensional meshes was established for the single-packet model [53, 102], in which each processor can only hold a single packet at any time. This bound applies to most of the indexing schemes used in the literature. An algorithm with a running time of $2D - n + o(n)$ was subsequently described by Kunde [55].

No non-trivial general lower bounds are known for sorting on d-dimensional meshes in the multi-packet model, in which a processor can hold any constant number of packets at a time. The best upper bound in this model is currently at $2D - 5n/2 + o(n)$. (This result can be obtained from the $7n/2 + o(n)$ time sorting algorithm for the three-dimensional mesh in [42].) Hence, for large values of d, this upper bound is still nearly a factor of 2 away from the diameter lower bound. This is also true in the case of the d-dimensional torus, where the best upper bound is currently at $2D - n + o(n)$. (This bound is implied by the $2n + o(n)$ time sorting algorithm for the three-dimensional torus in [42].)

Slightly better results have been obtained for permutation routing on d-dimensional networks. For this problem, Kunde [57] has described algorithms that run in time $(d + (d-2)(1/d)^{1/(d-2)} + \epsilon)n$ on the mesh, and in about half that time on the torus. For networks of low dimension, this is a significant improvement over previous results. However, for larger dimensions this result is again nearly a factor of 2 away from the diameter lower bound. In fact, even for off-line routing no better results are currently known.

For the problems of k–k routing and sorting on d-dimensional networks, there are obvious lower bounds of $kn/2$ for the mesh and $kn/4$ for the torus, due to the bisection width of the networks. For $k \geq 4r$, several randomized and deterministic algorithms have recently been proposed that match this lower bound, within a lower order additive term [46, 47, 59].

5.1.2 Overview of this Chapter

In this chapter, we show improved bounds for 1–1 routing and sorting on multi-dimensional meshes and tori. Our first result is a deterministic algorithm for sorting on multi-dimensional meshes of side length n that achieves a running time of $3D/2 + o(n)$. The algorithm has a fairly simple structure, and does not make any copies of the packets. We also show that the running time of the algorithm can be reduced to $5D/4 + o(n)$ by making one copy of each packet. A similar technique is then applied to the multi-dimensional torus, leading to a deterministic algorithm with a running time of $3D/2 + o(n)$. In contrast, the fastest previously known sorting algorithms required $2D - 5n/2 + o(n)$ steps on the mesh and $2D - n + o(n)$ steps on the torus.

Thus, our algorithms improve significantly over previous results for sorting, and in fact even for off-line routing, on multi-dimensional meshes and tori. The ideas underlying our algorithms are quite simple, but the ideas used in the design and analysis are somewhat different from those in previous algorithms for multi-dimensional networks. While we restrict our attention in this chapter to constant values of d, the claimed time bounds also hold for a limited range of networks of non-constant dimension.

In addition, we show lower bounds for sorting with respect to a large class of indexing schemes, under a model of the mesh where each processor can hold an arbitrary number of packets. Our lower bounds are the first non-trivial general lower bounds for sorting in the multi-packet model of the mesh, and they imply that our upper bounds are nearly optimal on networks of sufficiently high constant dimension under a large class of indexing schemes. (Some restricted lower bounds for the two-dimensional mesh have been obtained by Narayanan [81].) In fact, we are not aware of any fast sorting algorithm for multi-dimensional networks that uses an indexing scheme not covered by our lower bound. Using similar ideas, we can also establish a lower bound for selection on multi-dimensional meshes.

Finally, we describe algorithms for permutation routing on multi-dimensional meshes and tori whose running times nearly match the diameter lower bound. In particular, the algorithms achieve a running time of $D + \epsilon n$, for any $\epsilon > 0$ and d sufficiently large (depending on ϵ).

The remainder of this chapter is organized as follows. Section 5.2 contains some useful definitions and lemmas. Section 5.3 describes our algorithms for sorting. Section 5.4 contains the lower bounds, and Section 5.5 gives our results for permutation routing. Finally, Section 5.6 lists some open questions for future research.

5.2 Preliminaries

In this section, we give some useful definitions and lemmas. We begin with a brief discussion of the *sort-and-unshuffle* operation described in Chapter 4. In Subsection 5.2.2, we state some results on greedy routing of certain classes of permutations.

5.2.1 Randomization and Unshuffling

In the following, we review the technique for converting randomized into deterministic algorithms for routing and sorting on meshes described in Chapter 4. The technique is based on an operation called *sort-and-unshuffle*. The purpose of this operation is to evenly distribute packets with similar destinations (in the case of routing) or similar ranks (in the case of sorting) over some large region of the network, using a combination of local sorting and off-line routing.

In the following, we assume an arbitrary blocked indexing scheme on a d-dimensional mesh, where the blocks have side length n^α with $\alpha < 1$. In the first step of the sort-and-unshuffle operation, the packets are sorted inside each block. In the second step, the packets of each block are distributed evenly over all the blocks. This is done by routing the packet of rank j, $0 \leq j < n^{d\alpha}$, in block i, $0 < i \leq n^{d(1-\alpha)}$, to position $i + \left\lfloor j/n^{d(1-\alpha)} \right\rfloor \cdot n^{d(1-\alpha)}$ in block $j \bmod n^{d(1-\alpha)}$.

Note that this second step is an off-line routing problem; the particular permutation that has to be routed will be referred to as *unshuffle* permutation. (If we lay out the processors of the network in a chain according to the indexing function, then this permutation is identical to an $(n^{d(1-\alpha)})$-way unshuffle operation on the chain, as defined in Subsection 4.3.2.)

Informally speaking, the structure of the unshuffle permutation exhibits many of the "nice" properties commonly associated with "average" or "random" permutations. In particular, the unshuffle permutation has the property that the destinations of the packets in any region of the network are approximately evenly distributed over the entire network. As a consequence, an unshuffle permutation can usually be routed as efficiently as a random permutation.

It was shown in Chapter 4 that the sort-and-unshuffle operation can in many cases be employed as a "substitute" for randomization. Following a scheme originally proposed by Valiant [116], many randomized algorithms for routing and sorting on meshes start by sending the packets to random intermediate destinations. This has the effect of distributing packets with destinations close to each other evenly over the network. The sort-and-unshuffle operation simulates this effect in a deterministic manner. Using this relationship between randomization and the sort-and-unshuffle operation, the algorithms of this chapter can be described in both randomized and deterministic terms.

Note that the definition of the unshuffle permutation is with respect to a particular indexing scheme and its associated constant α. If $\alpha = 1/2$, then every unshuffle permutation sends exactly one packet from block i_0 to block i_1, for all i_0, i_1 with $0 \le i_0, i_1 < n^{d(1-\alpha)}$. (That is, every unshuffle permutation performs an "all-to-all" or "total exchange" among blocks of side length n^{α}.) For any two unshuffle permutations π_0 and π_1 with respect to indexings $\mathcal{I}_0$ and $\mathcal{I}_1$, respectively, with $\alpha = 1/2$, we have $\pi_0 = \rho_0 \circ \pi_1 \circ \rho_1$, where ρ_0 and ρ_1 are appropriately chosen local permutations that move packets only within the blocks. Thus, to show that all unshuffle permutations with $\alpha = 1/2$ can be efficiently routed, it suffices to show that some unshuffle permutation with $\alpha = 1/2$ can be efficiently routed. Similarly, we can also reduce any unshuffle permutation with $\alpha > 1/2$ to the case $\alpha = 1/2$ by performing local permutations before and after the unshuffle permutation. Thus, for the remainder of this section, we assume $\alpha = 1/2$.

5.2.2 Some Results on Greedy Routing

Routing is used as an important subroutine in many sorting algorithms for fixed-connection networks. In the algorithms presented in this chapter, we use the sort-and-unshuffle operation of Chapter 4, and therefore we need efficient routing schemes for unshuffle permutations. (Our results can also be obtained using randomization instead of the sort-and-unshuffle operation. We believe that the analysis is somewhat simpler in the deterministic case.)

We consider two different greedy routing schemes, which we refer to as the *standard greedy* and the *extended greedy* routing scheme. In the *standard greedy* routing scheme [65], every packet moves greedily towards its destination along edges of increasing dimension. In the case of edge contentions, priority is given to the packet with the farthest distance to travel.

In the *extended greedy* routing scheme, several permutations are simultaneously routed by running d "copies" of the standard greedy routing scheme. More precisely, we partition the set of packets into d sets $S_0, \ldots, S_{d-1}$ of (approximately) equal size, and route the packets in set S_i along edges of increasing dimension modulo d, starting with dimension i and ending with dimension $(i-1) \bmod d$.

Note that if the input for the extended greedy routing scheme is not given in the form of k separate permutations, but as a k–k routing problem, then we need to make sure that the origins and destinations of the packets in each set S_i are (approximately) evenly distributed over the entire network. This can be done either in a randomized way, by having each packet choose a random set S_i, or in a deterministic way, by locally sorting blocks of side length $o(n)$, and defining S_i as the set of packets with a local rank y such that $y \bmod d = i$.

It is a natural question to ask how many (random or unshuffle) permutations can be routed simultaneously under the above routing schemes. To make this question more precise, we define the notions of *diameter-optimal* and *distance-optimal* routing. We say that a routing algorithm on a d-dimensional mesh is *diameter-optimal* if all packets are delivered to their destination in time $D + o(n)$, where D is the diameter of the network. We say that a routing algorithm is *distance-optimal* if each packet is delivered in time $S + o(n)$, where S is the distance between the source and the destination of the packet.

In the context of an optimal randomized algorithm for k–k sorting, it was shown by Kaufmann, Rajasekaran, and Sibeyn [46] that up to $4d$ random permutations can be routed diameter-optimally on d-dimensional meshes and tori under the extended greedy routing scheme, with high probability. The same bound can also be shown for unshuffle permutations, leading to the optimal deterministic algorithms for k–k sorting in [47, 59]. However, these results cannot be extended to the case of distance-optimal routing.

For the standard greedy routing scheme, it is easy to see that one unshuffle permutation can be routed distance-optimally on d-dimensional meshes and tori. Leighton [65] has shown that this is also the case for a single random permutation, with high probability. (In fact, his result shows that it is unlikely that any packet is delayed by more than $O(\lg n)$ steps.) For the extended greedy routing scheme, we can show the following result.

Lemma 5.2.1 *Up to $2d$ unshuffle permutations can be routed distance-optimally on the d-dimensional torus.*

Proof: We show how to route $2d$ unshuffle permutations with $\alpha = 1/2$ in such a way that no two packets ever contend for an edge, and hence no packet ever gets delayed during the routing. For the sake of simplicity, we assume $n^\alpha = n^{1/2}$ to be odd.

Due to the structure of a torus, we can consider every block of side length $n^{1/2}$ to be at the center of a d-dimensional mesh (without wrap-around edges). There are $n^{d/2}$ blocks containing $n^{d/2}$ processors each. In each block B_i, we can now assign one processor to each of the $n^{d/2}$ blocks in the entire network, as follows. We assign the center processor of B_i to B_i itself, and we assign each other processor P_j to the unique block in the network whose position with respect to B_i corresponds to the position of P_j with respect to the center processor of B_i.

Thus, in each block B_i, we obtain a mapping ϕ_i that maps each block B_j in the network to a processor $\phi_i(B_j)$ in B_i. We now use this mapping to define the start positions of the packets originating in B_i. We assume that the packets are given in the form of $2d$ permutations S_ν, $0 \leq \nu < 2d$, and that permutation S_ν is routed first along dimension $\nu \bmod d$.

We first assign start positions to the packets in S_0 to S_{d-1}. Define $\psi_\nu : [n^{1/2}]^d \mapsto [n^{1/2}]^d$ as the function that shifts the coordinates of an element of $[n^{1/2}]^d$ by ν positions to the right, or formally, $\psi_\nu((x_0, \ldots, x_{d-1})) = (x_{d-\nu}, \ldots, x_{d-1}, x_0, \ldots, x_{d-\nu-1})$.

Then a packet in S_ν with source in B_i and destination in B_j is initially located in processor $\psi_\nu(\phi_i(B_j))$ in B_i. (Here, we assume that each processor in B_i is identified by an element of $[n^{1/2}]^d$, with $(0, \ldots, 0)$ being the processor with the smallest global coordinates in B_i.)

A packet starting in location $(x_0, \ldots, x_{d-1})$ of B_i is routed to the corresponding processor $(x_0, \ldots, x_{d-1})$ of its destination block B_j. This means that the distance a packets travels in a single dimension is always a multiple of $n^{1/2}$, and that packets only turn into the next dimension at times $l \cdot n^{1/2}$ with $l \in \mathbf{N}$. Due to the above assignment of initial locations to the packets, and due to the structure of the torus, whenever a packet p_0 in S_0 has to turn, say from dimension ν into dimension $\nu + 1$, then the (at most) $d - 1$ other packets $p_j \in S_j$, $1 \leq j < d$, currently located in the same processor have the property that p_j has to turn from dimension $(\nu + j) \bmod d$ into dimension $(\nu + j + 1) \bmod d$. In addition, a packet p_j that turns into a new dimension continues to travel in the same direction as the packet $p_{(j+1) \mathrm{mod} d}$ it "replaces" in this dimension. Thus, all packets can turn into their new dimensions without contention.

It remains to show that the permutations S_d to S_{2d-1} can be routed at the same time as S_0 to S_{d-1}. Let $\sigma : [n^{1/2}]^d \mapsto [n^{1/2}]^d$ be the function that maps each processor $(x_0, \ldots, x_{d-1}) \in [n^{1/2}]^d$ in a block B_i to the unique processor $(y_0, \ldots, y_{d-1})$ that has the same distance from the center processor of B_i, but is located in the opposite direction from the center processor. Then a packet in $S_{\nu+d}$, $0 \leq \nu < d$, with source in B_i and destination in B_j is initially located in processor $\sigma((x_0, \ldots, x_{d-1}))$, where $(x_0, \ldots, x_{d-1}) = \psi_\nu(\phi_i(B_j))$ is the initial location the packet would have assumed if it were contained in set S_ν. This means that throughout the routing, no packet in the sets S_d to S_{2d-1} will ever contend for an edge with a packet in the sets S_0 to S_{d-1}, since any packets from these sets that are encountered always move in the opposite direction.

$\square$

However, this simple analysis does not extend to the case of the d-dimensional mesh without wrap-around edges. In fact, it is not difficult to show that even d random permutations cannot be routed distance-optimally on the d-dimensional mesh under the extended greedy routing scheme. Using a more complicated analysis, we can show the following result for meshes without wrap-around edges.

Lemma 5.2.2 *Up to $\lfloor d/2 \rfloor$ unshuffle permutations can be routed distance-optimally on the d-dimensional mesh.*

We only describe the main ideas behind the proof. Let $\alpha = 1/2$. We partition the set of packets into d subsets by splitting each of the $\lfloor d/2 \rfloor$ permutations π_i into two sets S_{2i} and S_{2i+1}, such that S_{2i} contains all the packets that have a destination in a block B_j with $j = 0 \bmod 2$. The initial locations of the packets in S_0 to S_{d-1} are then chosen in exactly the same way as in the case of the sets S_0 to S_{d-1} in the proof of the previous lemma. (That is, in the definition of the mappings ϕ_i, we treat the mesh as if it were a torus with wrap-around connections.) To show that the packets are routed distance-optimally under the extended greedy routing scheme, we need to upperbound the number of packets that have to pass through any directed edge at any point in time; this is done using bounds for certain surfaces in d-dimensional space similar to the bounds for volumes in Lemma 2 of [32].

The maximum number of permutations that can be routed distance-optimally on the mesh is actually slightly larger than the above bound. In particular, we can prove the following result for the case of the three-dimensional mesh.

Lemma 5.2.3 *For $d = 3$, two unshuffle permutations can be routed distance-optimally on the d-dimensional mesh.*

5.3 Upper Bounds for Sorting

In this section, we give improved algorithms for 1–1 sorting. In the first subsection, we describe the basic ideas underlying our algorithms. Subsections 5.3.2 and 5.3.3 contain our algorithms for multi-dimensional meshes and tori, respectively.

5.3.1 Basic Ideas

The basic ideas underlying our algorithms are quite simple. Consider the case of the d-dimensional mesh, and let C denote the set of processors that have a distance of at most $D/4$ from the center. Note that exactly half of the processors of the network are contained in this *center region* C. Also, no processor in C has a distance of more than $3D/4$ from any other processor of the network.

These observations lead to the following idea for a fast sorting algorithm. In the first phase, we concentrate all packets into the center region C, in such a way that packets of similar ranks are evenly distributed over C. Next, we locally sort the packets inside each block of side length n^α (as defined by the blocked indexing scheme) that is contained in C. Since all packets were evenly distributed over the center region in the first phase, we can use the local ranks of the packets to obtain good approximations of the global ranks, and hence the final destinations, of all packets. In the third phase, we route each packet to some location in the block containing its approximate final destination. In the fourth phase, we use local sorting to bring each packet to its final destination.

Note that no packet has to travel a distance of more than $3D/4$ in the first or the third phase. Thus, if we can show that the routing in these two phases can be done distance-optimally, then the above scheme runs in time $3D/2 + o(n)$.

In the next subsection, we give a more detailed description of a deterministic algorithm based on the ideas presented in this section. We show that the routing problems in the first and third phase of the algorithm can be reduced to the simultaneous routing of several unshuffle permutations. We also present an even faster algorithm that makes one copy of each packet. In Subsection 5.3.3, we use similar ideas to obtain algorithms for the d-dimensional torus.

5.3.2 Sorting on Multi-Dimensional Meshes

In the following, we give fast deterministic sorting algorithms based on the ideas described in the previous subsection. We assume a blocked snake-like indexing scheme with blocks of side length n^α. In addition, we also assume an arbitrary fixed numbering of the $n^{d(1-\alpha)}/2$ blocks located in the center region C, independent of the indexing scheme. We begin with the following simple algorithm:

Algorithm SimpleSort:

(1) Sort the packets in each block of side length n^α.

(2) Distribute the packets in each block evenly over all blocks in C. This is done by routing the packet of rank i, $0 \le i < n^{d\alpha}$, in block j, $0 < j \le n^{d(1-\alpha)}$, to position $j + \left\lfloor i/n^{d(1-\alpha)} \right\rfloor \cdot n^{d(1-\alpha)}$ in block $i \bmod (n^{d(1-\alpha)}/2)$ in C. (Here, the numbering of the destination blocks is with respect to the arbitrary fixed numbering of the blocks in C.) Note that each processor in C receives exactly two packets.

(3) Sort the packets in each block of side length n^α in C.

(4) Send the packets in each block in C towards their destinations. This is done by routing the packet of rank i, $0 \le i < 2n^{d\alpha}$, in block j, $0 < j \le n^{d(1-\alpha)}/2$ of C to position $j + (i \bmod 2n^{d(2\alpha-1)}) \cdot n^{d(1-\alpha)}/2$ in block $\left\lfloor i/(2n^{d(2\alpha-1)}) \right\rfloor$. (Here, the numbering of the source blocks is with respect to the arbitrary fixed numbering of the blocks in C.) Note that each processor in the network receives exactly one packet.

(5) Perform two steps of odd-even transposition sort between neighboring blocks.

The correctness of the above algorithm is implied by the following lemma, which can be proved along the lines of Lemma 3.2 in [47].

Lemma 5.3.1 *After Step (4) of Algorithm SimpleSort, each packet is at most one block away from its destination.*

Next, we analyze the running time of the above algorithm. Clearly, Steps (1), (3), and (5) each run in time $O(n^\alpha) = o(n)$. For the routing in Step (2), the following can be shown.

Lemma 5.3.2 *Step (2) of Algorithm SimpleSort can be reduced to the routing of two partial unshuffle permutations.*

Proof: Consider the packets of a single unshuffle permutation π. Let S be the set of processors that contain a packet with destination in C. In each block, exactly half of the processors are in S, and the destinations of the packets in these processors are evenly distributed over all blocks in C. Let π' be the partial permutation consisting only of the packets that are initially located in S. After π' has been routed, we move the remaining half of the packets to the processors in S. By routing another instance of the partial unshuffle permutation π', we can now distribute these remaining packets evenly over the blocks in C. Of course, the two instances of π' can also be started simultaneously.

$\square$

By Lemma 5.2.3, we know that two partial unshuffle permutations can be routed distance-optimally on meshes of dimension $d \geq 3$. Since no packet has to travel a distance of more than $3D/4$, this implies that the routing is completed in time $3D/4 + o(n)$. Also note that the routing problem in Step (4) is exactly the inverse of the problem in Step (2), and therefore runs within the same time bound. This establishes the following result.

Theorem 5.3.1 *For any constant d, there exists a deterministic sorting algorithm for the d-dimensional mesh with a running time of $3D/2 + o(n)$ that does not make any copies of the packets.*

By Lemma 5.2.2, up to $\lfloor d/2 \rfloor$ unshuffle permutations can be routed distance-optimally on d-dimensional meshes. By modifying Algorithm SimpleSort appropriately, we can use this extra bandwidth to establish the following result for k–k sorting.

Corollary 5.3.1.1 *If $k \leq \lfloor d/4 \rfloor$, then there exists a deterministic algorithm for k–k sorting on the d-dimensional mesh with a running time of $3D/2 + o(n)$ that does not make any copies of the packets.*

We can also get a slight improvement in the running time for 1–1 sorting, by concentrating the packets into a smaller center region C. In general, however, the running time of this improved algorithm is still $(3/2 - \epsilon)D$, for all $\epsilon > 0$ and d sufficiently large (depending on ϵ).

Corollary 5.3.1.2 *Let $C(r)$ be the set of processors of distance at most r from the center point. If $|C(r)| \geq 2n^d/d$, then there exists a deterministic sorting algorithm for the d-dimensional mesh with a running time of $D + 2r + o(n)$.*

Next, we show that the time for 1–1 sorting can be reduced to $5D/4 + o(n)$ by making one copy of each packet. To do so, we modify Algorithm SimpleSort appropriately; the resulting algorithm is called CopySort. Steps (1), (3), and (5) remain the same as in Algorithm SimpleSort. The routing in Step (2) of SimpleSort is augmented as follows. As before, we distribute the packets evenly over the blocks in C. In addition, we make one copy of each packet, and route this copy to a processor in the unique block of the center region C that is located exactly on the opposite side of the center point than the destination processor of the original in this step, and that has the same distance from the center point. The routing of the copies can be done simultaneously with the routing of the originals, and the entire Step (2) can be implemented by routing four partial unshuffle permutations. By Lemma 5.2.2, the routing is completed in $3D/4 + o(n)$ steps for $d \geq 8$. The following lemma can be shown using simple geometric arguments.

Lemma 5.3.3 *After Step (3) of Algorithm CopySort, no processor in the network is more than a distance of $D/2 + o(n)$ away from both the original and the copy of any packet.*

In Step (4) of CopySort, we first delete either the original or the copy of each packet, depending on which one is farther away from the destination. Then the remaining packets are routed towards their destination. This routing can again be implemented by four partial unshuffle permutations. By Lemma 5.3.3, no packet has to travel more than a distance of $D/2$. This establishes the following result.

Theorem 5.3.2 *For any constant $d \geq 8$, there exists a deterministic sorting algorithm for the d-dimensional mesh with a running time of $5D/4 + o(n)$.*

For larger values of d, this result can again be slightly improved by concentrating into a smaller center region. Alternatively, we can also adapt the algorithm to k–k sorting with $k \leq \lfloor d/8 \rfloor$.

5.3.3 Sorting on Multi-Dimensional Tori

In this subsection, we adapt the ideas of the previous subsections to the case of the d-dimensional torus. We describe a modification of the Algorithm CopySort from the previous subsection, which we refer to as TorusSort. As before, Steps (1), (3), and (5) perform local sorting operations. In Step (2), we distribute the packets evenly over the entire network. In addition, we also make a copy of each packet, and route this copy to a processor in the unique block in the network that is $D/2$ steps away from the destination processor of the original packet in this step. Step (2) can be implemented by routing two full unshuffle permutations; the routing takes time $D + o(n)$. Then the following lemma holds.

Lemma 5.3.4 *After Step (3) of Algorithm TorusSort, no processor in the network is more than a distance of* $D/2 + o(n)$ *away from both the original and the copy of any packet.*

As before, half of the packets are deleted in Step (4), and the remaining packets are routed towards their destination. This routing can be implemented by two partial unshuffle permutations. By Lemma 5.3.4, no packet has to travel more than a distance of $D/2 + o(n)$. Using Lemma 5.2.1 we obtain the following result.

Theorem 5.3.3 *For any constant* d*, there exists a deterministic sorting algorithm for the* d*-dimensional torus with a running time of* $3D/2 + o(n)$*.*

By modifying Algorithm TorusSort appropriately, and using the extra bandwidth supplied by Lemma 5.2.1, we can establish the following result.

Corollary 5.3.3.1 *For any constant* d*, there exists a deterministic algorithm for* d*–*d *sorting on the* d*-dimensional torus with a running time of* $3D/2 + o(n)$*.*

Alternatively, we can also get a slight improvement in the running time for 1–1 sorting. As an example, we can obtain a fairly simple algorithm for the two-dimensional torus that uses four copies of each packet and runs in time $1.375n$. In general, however, the running time of the improved algorithm is still $(3/2 - \epsilon)D$, for all $\epsilon > 0$ and d sufficiently large (depending on ϵ).

5.4 Lower Bounds for Sorting

In this section, we establish lower bounds for sorting on multi-dimensional meshes and tori under the multi-packet model. The lower bounds hold for a large class of indexing schemes, including most of the indexing schemes used in the literature. Our lower bound technique is an extension of the *Joker Zone* argument of Kunde [53] and Schnorr and Shamir [102] to the multi-packet model. An important difference is that our lower bounds are based on edge capacity arguments, and do not place any limits on the number of packets that can be held inside a single processor. We begin the section with a few definitions.

We say that an indexing scheme $\mathcal{I}$ of the d-dimensional mesh is *compatible* if there exists a $\beta < 1$ such that for every index $i \in [n^d - n^{\beta d}]$, the set of processors with indices in $\{i, \ldots, i + n^{\beta d} - 1\}$ contains a complete $(d-1)$-dimensional subnetwork of side length n. (Informally speaking, this means that a compatible indexing scheme has the property that a joker zone of $n^{\beta d}$ packets suffices to move the final destination of a packet to any processor within a $(d-1)$-dimensional sub-network.) Note that the natural extensions of the row-major, snake-like, blocked row-major, and blocked snake-like indexing schemes to multi-dimensional networks are compatible indexing schemes. In the remainder of this section, we assume an arbitrary compatible indexing scheme with associated constant β.

We use $C_{d,\gamma}$ to denote the processors of a d-dimensional diamond of radius $(1-\gamma) \cdot D/4$ around the center of a d-dimensional mesh. (That is, the set of processors that have a distance of at most $(1-\gamma) \cdot D/4$ from the center.) The number of processors in $C_{d,\gamma}$ is denoted by $V_{d,\gamma}$, and the number of processors on the surface of $C_{d,\gamma}$ is denoted by $S_{d,\gamma}$. Then the following upper bounds can be shown using Chernoff Bounds [14].

Lemma 5.4.1 *For any d and any $\gamma > 0$, we have*

$$V_{d,\gamma} \leq e^{-\gamma^2 d/4} \cdot n^d$$

and

$$S_{d,\gamma} \leq \frac{8}{\gamma} \cdot e^{-\gamma^2 d/16} \cdot n^{d-1}.$$

5.4.1 Sorting without Copying

We first establish a lower bound for sorting under the restriction that no copies of the packets can be made. Our main lemma for this case is as follows.

Lemma 5.4.2 *Let d and γ be chosen such that*

$$d \cdot S_{d,\gamma} \cdot \left(\left(\frac{1}{2} + \frac{1-\gamma}{4} \right) \cdot D - dn^\beta \right) < n^d - V_{d,\gamma}$$

holds for large enough n. If no copying of packets is allowed, then sorting on the d-dimensional mesh with respect to an arbitrary compatible indexing scheme takes at least $D + (1-\gamma) \cdot D/2 - n - dn^\beta$ steps.

Proof: Consider the computation of an arbitrary sorting algorithm up to time $(\frac{1}{2} + \frac{1-\gamma}{4}) \cdot D - dn^\beta$. At most $d \cdot S_{d,\gamma}$ packets can enter the diamond $C_{d,\gamma}$ in each step. Thus, the above inequality implies that not all of the $n^d - V_{d,\gamma}$ packets that are initially outside the diamond can have entered up to this point.

Now consider an arbitrary packet located outside the diamond at time $(\frac{1}{2} + \frac{1-\gamma}{4}) \cdot D - dn^\beta$. This packet has a distance of at least $(\frac{1}{2} + \frac{1-\gamma}{4}) \cdot D$ from at least one of the corners of the network. (Otherwise, the packet would be in the diamond.) Thus, the present position of the packet is independent of the content of a block of side length n^β located in that corner.

As we assume a compatible indexing scheme, the content of this block can force the destination of the packet to be in any processor of a $(d-1)$-dimensional sub-network of side length n. There exists a processor in this sub-network that has a distance of at least $(\frac{1}{2}+\frac{1-\gamma}{4})\cdot D-n$ from the current position of the packet. Hence, at least $(\frac{1}{2}+\frac{1-\gamma}{4})\cdot D-n$ additional steps are needed under some assignment of values to the corner block.

□

Theorem 5.4.1 *If no copying of packets is allowed, then for every $\epsilon \geq 0$ there exists a d_0 such that for all $d \geq d_0$, sorting on the d-dimensional mesh with respect to a compatible indexing scheme takes at least $(3/2-\epsilon)D$ steps.*

To establish this theorem, we use Lemma 5.4.1 to show that the condition in Lemma 5.4.2 holds for $\gamma = 3\epsilon$ and d sufficiently large (depending on ϵ), The claim then follows by a direct application of Lemma 5.4.2. Together with Theorem 5.3.2, this result establishes a separation between the complexities of sorting with and without copying, for large values of d. Unfortunately, Lemma 5.4.1 does not give any good bounds for small values of d. In this case, we can show lower bounds by adapting our argument to the particular network in question. In particular, we can establish the following theorem.

Theorem 5.4.2 *If no copying of packets is allowed, then for $d \geq 5$ the diameter bound cannot be asymptotically matched under any compatible indexing scheme.*

For the torus, it can be shown that the lower bounds for the single-packet model also extend to the multi-packet model, assuming that no copying is allowed. Informally speaking, the reason is that the torus does not have a center point towards which the packets could be routed.

5.4.2 Sorting with Copying

Our lower bound technique can also be extended to a model in which unlimited copying of packets is allowed. For this case, we obtain the following result.

Theorem 5.4.3 *If unlimited copying of packets is allowed, then for every $\epsilon \geq 0$ there exists a d_0 such that for all $d \geq d_0$, sorting on the d-dimensional mesh with respect to a compatible indexing scheme takes at least $(5/4 - \epsilon)D$ steps.*

We only describe the main ideas in the proof of the above theorem. The basic idea for this lower bound is that we choose the center diamond small enough such that only a small fraction of the packets can be routed into this diamond. Next, we argue that the edge bandwidth of the network does not allow every packet to distribute a large number of copies of itself over the network. (Formally, the number of communication steps required to route copies of a packet to a number of locations in the network is lowerbounded by the length of a minimal "broadcast tree" connecting these locations.) This implies that an appropriate loading of the joker zones can force the rank of a packet to be such that no copy is close to its destination.

However, this technique does not give any non-trivial lower bounds for reasonable values of d. We expect that some results for smaller d can be obtained by adapting our argument to the particular low-dimensional network in question. In the case of the torus, we obtain the following result.

Theorem 5.4.4 *If unlimited copying of packets is allowed, then for every $\epsilon \geq 0$ there exists a d_0 such that for all $d \geq d_0$, sorting on the d-dimensional torus with respect to a compatible indexing scheme takes at least $(3/2 - \epsilon)D$ steps.*

The lower bounds can be extended to many non-compatible indexing schemes. In fact, it is not difficult to show that the above bounds hold for the vast majority of all possible indexing schemes. (A similar result for the single-packet model was described by Kunde [54].) Of course, such a result is not a very good measure for the generality of our lower bounds, since most indexing schemes are highly irregular and thus unsuitable for any efficient sorting scheme. More important in this respect is that we are not aware of any fast sorting algorithm that assumes an indexing scheme not covered by our lower bound.

5.4.3 Selection

Using similar ideas, we can also show a lower bound of $(9/16-\epsilon)\cdot D$ for the problem of selecting the median at the center processor of a high-dimensional mesh. A trivial lower bound for this problem is given by the radius of the network. (That is, $D/2$ for the multi-dimensional mesh and D for the multi-dimensional torus.)

By Lemma 5.4.1, we know that for any $\epsilon > 0$ and any sufficiently large d, only a small fraction of the packets can enter $C_{d,\epsilon}$ in the first $D/2$ steps of any algorithm. Let x be any processor outside $C_{d,\epsilon}$. Then the set of processors that have a distance of at most $(5/16-2\epsilon)\cdot D$ from x contains only a small fraction of the n^d processors in the network. This means that up to time $(5/16-2\epsilon)\cdot D$, no packet located outside $C_{d,\epsilon}$ can be "ruled out" as the median element. Hence, up to $(1-\epsilon)\cdot D/4$ additional steps may be necessary to move the median to the center processor, and we get the following result.

Theorem 5.4.5 *For every $\epsilon \geq 0$ there exists a d_0 such that for all $d \geq d_0$, selection on the d-dimensional mesh takes at least $(9/16-\epsilon)D$ steps.*

An upper bound of $D + o(n)$ can be obtained by a modification of the sorting algorithms in Section 5.3. For large values of d, this result can be improved to $3D/4 + \epsilon n$. On the multi-dimensional torus, a running time of $D + \epsilon n$ can be achieved for large d, thus coming very close to the trivial lower bound of D.

5.5 Permutation Routing

The lower bounds established in the previous section are restricted to the case of sorting. In this section, we show the existence of algorithms for permutation routing on multi-dimensional networks that nearly match the diameter lower bound. The algorithms are based on similar ideas as the sorting algorithms in Subsection 5.3. In particular, they use a similar reduction to the distance-optimal routing of a number of unshuffle permutations.

Consider the following idea for a randomized routing algorithm. For a packet with source processor x and destination processor y, we define $S(x,y)$ as the set of processors that have a distance of at most $D/2$ from both x and y. Note that $S(x,y)$ is non-empty for all x and y. Thus, a simple two-phase algorithm could route a packet with source x and destination y by first sending the packet to a random processor in $S(x,y)$, and then to its destination y. If we could solve the resulting two routing problems distance-optimally, then we would obtain a total running time of $D + o(n)$ for the algorithm.

Unfortunately, we do not know how to reduce these two routing problems to a small number of random or unshuffle permutations. To do so, we have to modify the above algorithm slightly. We define $S_\nu(x,y)$ as the set of processors that have a distance of at most $D/2 + \nu$ from both x and y. In the first phase of the algorithm, we now route each packet with source x and destination y to a random processor in $S_\nu(x,y)$. In the corresponding deterministic algorithm, we partition the network into blocks of side length n^α, and distribute all packets with source in block X and destination in block Y evenly over $S_\nu(X,Y)$, the set of blocks that have a distance of at most $D/2 + \nu$ from both block X and block Y.

If we choose ν such that $k \cdot |S_\nu(X,Y)| \geq n^d$ holds for all blocks X and Y, then we can reduce each phase of the algorithm to the simultaneous routing of k unshuffle permutations. For a block X, define $c(X)$ as the corner processor that is closest to X. Then we can lowerbound $|S_\nu(X,Y)|$ by $|S_\nu(c(X),c(Y))|$. An analysis shows that for $d \geq 4$ and $\nu = n/2$, we have $\lfloor d/2 \rfloor \cdot |S_\nu(c(X),c(Y))| \geq n^d$, and hence we can reduce each phase of the algorithm to the routing of $\lfloor d/2 \rfloor$ unshuffle permutations. Using Lemma 5.2.2, we obtain the following result.

Theorem 5.5.1 *For all d, there exists a deterministic algorithm for permutation* *routing on the d-dimensional mesh with a running time of $D + n + o(n)$.*

The routing scheme can be easily adapted to the multi-dimensional torus. For $d \geq 3$ and $\nu = n/16$, we have $2d \cdot |S_\nu(X, Y)| \geq n^d$, and by Lemma 5.2.1 we obtain the following result.

Theorem 5.5.2 *For all d, there exists a deterministic algorithm for permutation routing on the d-dimensional torus with a running time of $D + n/8 + o(n)$.*

An analysis using the bounds in Lemma 5.4.1 shows that in high-dimensional meshes (tori), most processors have a distance of around $D/2$ from any particular corner (any particular processor). This means that as d increases, we can choose smaller and smaller values for ν.

Theorem 5.5.3 *For all $\epsilon > 0$, there exists a d_0 such that for all $d \geq d_0$, permutation routing can be done in time $D + \epsilon n$ on d-dimensional meshes and tori.*

Finally, by making careful use of the bandwidth provided by Lemma 5.2.1, we can show the following result for the torus. Note that this result comes very close to both the diameter and the distance bound.

Theorem 5.5.4 *For all $\epsilon > 0$, there exists a d_0 such that for all $d \geq d_0$, $2d$–$2d$ routing can be done in time $(1 + \epsilon) \cdot D$ on d-dimensional tori.*

5.6 Open Questions

In this chapter, we have shown improved bounds for routing and sorting on multi-dimensional meshes and tori. While our bounds are nearly tight for high-dimensional networks, we do not obtain very good bounds for networks of small, fixed dimension. In particular, it is an interesting question whether there exists an optimal algorithm for sorting on the two-dimensional mesh that does not make any copies, or whether any optimal sorting algorithm exists for some $d \geq 3$.

Another open question is whether the lower bounds for sorting can be extended to arbitrary indexing schemes. One possible approach to this problem is to try to adapt some of the techniques that have been used to show lower bounds for arbitrary indexing schemes in the single-packet model [31].

It would also be nice to obtain algorithms for permutation routing that match the diameter bound more closely. For example, one might try to overlap the two routing phases of the algorithm in Section 5.5, and bound the running time of the resulting algorithm. Finally, it is an open question whether the diameter and bisection bounds can be matched simultaneously for routing on meshes of dimension $d \geq 2$.

Chapter 6

Routing and Sorting on Meshes with Buses

This chapter considers the problems of permutation routing and sorting on several models of meshes with fixed and reconfigurable buses. We describe two fairly simple deterministic algorithms for permutation routing on two-dimensional networks, and an algorithm for d-dimensional networks. We also give deterministic algorithms for 1–1 sorting. The algorithms can be implemented on a variety of different models of meshes with buses.

6.1 Introduction

One of the main drawbacks of the theoretical mesh model is its large diameter in comparison to many other networks, such as the hypercube and its bounded-degree variants [66]. An $n \times n$ mesh has a radius of $n - 1$, and hence even computations that require only a very limited amount of communication, for example prefix computations, still require at least $n - 1$ communication steps.

To remedy this situation, it was proposed by several authors [10, 41, 109] to augment the mesh architecture with high-speed buses that allow fast communication between processors located in different areas of the mesh. This has resulted in a large body of literature on various different models of meshes with bus connections, and a number of important algorithmic problems have been studied under these models. Among the most frequently studied problems on meshes with buses are Maximum [1, 10, 21, 79], Prefix Sums [6, 13, 21, 61, 94, 109], Selection [12, 33, 94, 109], as well as certain algorithmic problems in image processing and graph theory [38, 40, 79, 93, 110]. Additional literature can be found in [75] and the above references.

Due to the low communication requirements of most of the above problems, significant speed-ups over the standard mesh can be achieved. The exact time complexities of the proposed algorithms depend heavily on the properties of the bus system. For example, the maximum of n^2 elements can be computed in time $O(\lg\lg n)$ on an $n \times n$ mesh with a fully reconfigurable bus, while the same problem requires $\Theta(n^{1/3})$ steps on a mesh with fixed row and column buses. On the mesh without buses, at least $n-1$ steps are needed. In the following, we briefly describe some of the main features of the bus system that determine the power of the model.

(1) Architecture of the bus system: A bus is called *global* if it is connected to all processors in the network. A bus that is connected to only a subset of the processors is called *local*. Examples of meshes with one or several global buses are given in [1, 10, 79, 109]. Most of the work on local buses has focused on the mesh with row and column buses [21, 94, 110], although other architectures have been proposed [78, 110].

(2) Reconfigurability of the buses: A bus is called *reconfigurable* if it can be partitioned into subbuses, such that each subbus can be used as a separate, independent bus. A bus that is not reconfigurable is called *fixed*. In a system with reconfigurable buses, the possible partitions of the buses depend on the layout

of the bus system. As an example, consider an $n \times n$ mesh with reconfigurable row and column buses laid out in the obvious way. Then each of the n row buses (column buses) can only be partitioned into subbuses connecting groups of consecutive processors of the respective row (column).

(3) Conflict resolution for bus access: Most papers assume that the buses have broadcast capability, that is, a value written on the bus by one processor can be read by all other processors connected to the bus in the next step. Another common assumption is that the result is undefined if several processors attempt to write on the same bus in a single step of the computation. Using the PRAM terminology, we refer to such a bus as being *Concurrent Read Exclusive Write*, or CREW for short. Similarly, we could define CRCW or EREW buses. There is a close relationship between a shared memory cell in a CREW/CRCW/EREW PRAM and a global bus of the same type [78].

Additional features that have been studied include buses with non-unit delay [75, 79], and buses that allow pipelining of messages under certain conditions [30]. Finally, the concept of a mesh with a reconfigurable bus system can also be generalized to reconfigurable networks of arbitrary topology [8].

The model of computation assumed in this chapter is a mesh with row and column buses. We consider both fixed and reconfigurable buses. Of course, all algorithms designed for such a model also run on more powerful models, such as the *Polymorphic Torus* [75], the RMESH [79], or the PARBUS [120], whose bus system can be reconfigured into row and column buses. On the other hand, it does not seem that these more powerful, but also less realistic, models offer any advantages with respect to permutation routing and 1–1 sorting, which are primarily restricted by the bisection width of the network. Unless explicitly stated otherwise, we assume the buses to be CREW.

An alternative way to overcome the diameter restriction of the standard mesh is to augment the network with a sparse system of bidirectional communication links connecting processors in different areas of the mesh. Examples for this approach are the *Mesh of Trees* [66], or the *Packed Exponential Connections* [49]. It turns out that many of the algorithms and techniques described in this chapter can be adapted to these classes of networks, and we will point this out in a few instances.

6.1.1 Related Results

In this chapter, we fovus on the problems of permutation routing and 1–1 sorting (see Subsection 1.2.3 for a definition of these problems). It is easy to see that both of these problems require at least $\Theta(n)$ steps on all proposed variants of meshes with buses, due to bisection width. However, the exact complexity of these problems has only recently been investigated.

The study of permutation routing on meshes with row and column buses was proposed by Leung and Shende [72]. They assume a model of computation, hereinafter referred to as the *mesh with fixed buses*, that consists of a mesh with non-reconfigurable row and column buses in addition to the standard mesh edges. For the one-dimensional case, they obtain a permutation routing algorithm running in $2n/3$ steps with small constant queue size. They also show a matching lower bound of $2n/3$ for this problem; this lower bound can be extended to multi-dimensional networks. For the two-dimensional case, Leung and Shende show that every permutation can be routed off-line in $n + 1$ steps. They also describe a deterministic on-line algorithm that routes in time $(7/6 + \epsilon)n + o(n)$ and queue size $O(1/\epsilon)$ on the two-dimensional mesh with fixed buses, and in time $(7(d-1)/6 + \epsilon)n + o(n)$ and queue size $O(\epsilon^{1-d})$ on d-dimensional networks. (Recall that the queue size is the maximum number of packets any node has to store during the algorithms.) In a subsequent paper [73], they obtain an improved algorithm for the two-dimensional case, running in time $(1 + \epsilon)n + o(n)$ with a queue size of $O(1/\epsilon)$.

Rajasekaran and McKendall [96, 97] describe randomized algorithms for routing and sorting on a network in which the mesh edges have been replaced by a global reconfigurable bus. This model is essentially the same as the PARBUS, but has the additional property that every subbus of length 1 can be used in the same way as a bidirectional edge in a standard mesh. This means that in this case a message can be transmitted in either direction in a single step. There is an obvious lower bound of $n/2$ steps for permutation routing and sorting in this model, due to the bisection width of the network. Rajasekaran and McKendall describe a $3n/4$ time deterministic algorithm for permutation routing in the one-dimensional case, and a randomized algorithm for the two-dimensional case that achieves a running time of $(1+\epsilon)n$ and a queue size of $O(1/\epsilon)$, with high probability. They also give randomized algorithms for sorting with the same bounds on running time and queue size.

While the assumption of bidirectional communication in subbuses of length 1 made in the model of Rajasekaran and McKendall may be technologically feasible, it can also be perceived as somewhat unsatisfactory from a theoretical point of view, since it adversely affects the simplicity of the model. In this context, we point out that many of their algorithms, including the $(1 + \epsilon)n$ time routing algorithm for the two-dimensional case, do not make use of this assumption. Similarly, their algorithms do not use any bus connections other than those along a single row or column. Thus, in the following we consider a model with reconfigurable row and column buses, and we assume that only one processor can write on a subbus in any single step, regardless of the length of that subbus. Note that in this model, hereinafter referred to as the *mesh with reconfigurable buses*, there is a trivial lower bound of n steps for permutation routing and sorting due to bisection width.

Comparing the two different models of meshes with buses described above, we observe that the mesh with reconfigurable buses can emulate the standard mesh with constant slowdown by partitioning the buses appropriately. In the case of the mesh with fixed buses, on the other hand, we cannot remove the standard mesh edges without losing the capability of efficiently performing local communication among groups of adjacent processors. In fact, several routing algorithms for such a network with fixed buses and no local connections have been proposed by Iwama, Miyano, and Kambayashi [39]. Due to the impossibility of efficient local communication, their algorithms have a queue size of $\Theta(n)$ in the worst case.

Very recently, and independent of our work, Sibeyn, Kaufmann, and Raman [107] have obtained a randomized routing algorithm for the two-dimensional mesh with fixed buses that runs in time $0.78n$, and an algorithm for d-dimensional networks that runs in time $(2 - 1/d)n + o(n)$. (The exact running time is actually slightly better than this bound.) By applying the techniques described in Chapter 4 of this thesis, it is possible to obtain deterministic algorithms that match the running times of these randomized algorithms, within a lower order additive term.

Sibeyn, Kaufmann, and Raman also show improved lower bounds for routing on meshes with fixed buses. In particular, they show lower bounds of $0.69n$ and $0.72n$ for the two-dimensional and three-dimensional cases, respectively, and a lower bound of approximately $\frac{d-1}{d}n$ for d-dimensional networks. (The lower bound for the two-dimensional case was also discovered by Cheung and Lau [16].)

In other independent work, Cogolludo and Rajasekaran [19] have given a $\frac{17n}{18} + o(n)$ time randomized routing algorithm for the two-dimensional mesh with reconfigurable buses, under the assumption that subbuses of length 1 can be used as bidirectional edges. They also give an algorithm with running time $\frac{2n}{3} - \frac{n}{64} + o(n)$ for a model with two unidirectional reconfigurable buses in each row and column.

For the problem of k–k routing on d-dimensional networks, $d \geq 1$, there are obvious lower bounds of $kn/3$ and kn for the mesh with fixed and reconfigurable buses, respectively, due to the bisection width of the network. For the mesh with fixed buses, Rajasekaran [96] and Sibeyn, Kaufmann, and Raman [107] describe randomized algorithms that match this lower bound, within a lower order additive term. An optimal randomized algorithm for k–k sorting on the mesh with reconfigurable buses can be obtained by a straightforward implementation of the algorithm for the standard mesh given in [46]. Very recent work by Kaufmann, Sibeyn, and Suel [47] and Kunde [59] implies that this bound can also be matched deterministically.

6.1.2 Overview of this Chapter

In this chapter, we study the problems of permutation routing and 1–1 sorting on meshes with row and column buses. We consider several variants of this model, with both fixed and reconfigurable buses.

In the first part of the chapter, we describe two fairly simple algorithms for the two-dimensional case that achieve a running time of $n + o(n)$ and very small queue size, and an algorithm for d-dimensional networks, $d \geq 3$, with a running time of $(2 - 1/d)n + o(n)$ and a queue size of 2. An interesting feature of these algorithms is that they can be efficiently implemented on a variety of different classes of networks. The algorithms are obtained with a new technique that allows us to convert certain off-line routing schemes into deterministic on-line algorithms. We believe that this technique may have further applications.

In the second part of the chapter, we present two algorithms for 1–1 sorting. The first algorithm is based on a deterministic sampling technique, and its running time matches that for permutation routing, within a lower order additive term. The second algorithm is based on a variation of Columnsort, and runs in time $n + o(n)$ on meshes with reconfigurable buses of arbitrary constant dimension, thus nearly matching the bisection lower bound of n steps.

The remainder of this chapter is organized as follows. Section 6.2 contains the results for permutation routing, and Section 6.3 describes our algorithms for sorting. Finally, Section 6.4 lists some open questions for future research.

6.2 Permutation Routing

In this section, we describe a technique that allows us to convert certain off-line routing schemes into deterministic routing algorithms. We then use this technique to design new algorithms for permutation routing on meshes with buses. We begin by giving an alternative description of a simple $n+1$ step off-line routing scheme proposed by Leung and Shende [72, 73]. In Subsection 6.2.2 we show how this off-line routing scheme can be used to obtain a fast and fairly simple deterministic routing algorithm for two-dimensional meshes with buses. Subsection 6.2.3 applies the technique to multi-dimensional networks. Finally, Subsection 6.2.4 gives another algorithm for the two-dimensional case.

6.2.1 Off-line Routing

In the off-line routing scheme of Leung and Shende [72, 73], every packet is routed to its destination by first routing it on a column bus to its destination row, and then routing it on a row bus to its destination column in the following step. Thus, the algorithm does not make use of the mesh edges at all. Leung and Shende show that, for any input permutation, a schedule for the above routing scheme can be computed in time $O(n^{7/2})$ by computing a sequence of n maximum matchings. Once the schedule has been computed, it can be executed in $n+1$ steps.

Now consider the following interpretation of the above scheduling problem. The columns of the mesh are interpreted as *processes* $P_0, \ldots, P_{n-1}$. Every process P_i has exclusive ownership of its column bus, and has to transmit the n packets initially located in its column to their destinations. To do so, a process needs to send packets on the row buses, which are interpreted as *resources* $R_0, \ldots, R_{n-1}$. Before a packet can be transmitted across a row bus to its final destination, it has to be routed within its column to the correct row; this can be done in the preceding step using the column bus. If k packets in column i have a destination in row j, then process

P_i needs to access resource R_j for k time steps. These k steps can be scheduled in any arbitrary order, provided that in any given step, each resource is accessed by at most one process, and each process uses at most one resource. The problem of finding a minimum time schedule that satisfies all of these demands is known as the *Open Shop Scheduling Problem* [29].

For $0 \leq i, j < n$, let $D_{i,j}$, the *demand* of process P_i for resource R_j, be the number of packets in column i that have a destination in row j. Note that

$$\sum_{i=0}^{n-1} D_{i,j} = n \tag{6.1}$$

holds for all j, since every row is the destination of exactly n packets. Correspondingly, we also have

$$\sum_{j=0}^{n-1} D_{i,j} = n \tag{6.2}$$

for all i, since every column is the origin of exactly n packets. A simple algorithm for finding a minimum time schedule computes a sequence of maximum matchings in the bipartite graph $G = (U, V, E)$ defined by $U = \{P_0, \ldots, P_{n-1}\}$, $V = \{R_0, \ldots, R_{n-1}\}$, and $E = \{(P_i, R_j) \mid D_{i,j} > 0\}$. More precisely, the algorithm first computes a maximum matching M of G, and schedules each process with its matched resource for $D_{\min}$ time steps, where $D_{\min} = \min\{D_{i,j} \mid (P_i, R_j) \in M\}$. Next, we subtract $D_{\min}$ from all $D_{i,j}$ with $(P_i, R_j) \in M$, construct a new bipartite graph G' corresponding to the new values of the $D_{i,j}$, and compute a new maximum matching M'. This procedure is repeated until all demands $D_{i,j}$ have been reduced to zero. Using Hall's Matching Theorem, it can be shown that Equations (6.1) and (6.2) guarantee that the resulting schedule has a length of at most n. This in turn implies that at most n matchings have to be computed, since for every matching the length of the schedule is increased by at least one step.

A maximum matching on a bipartite graph with $2n$ vertices can be computed in time $O(n^{5/2})$ using the algorithm of Hopcroft and Karp [35]. Thus, the entire schedule can be computed in time $O(n^{7/2})$. Of course, this makes the algorithm inappropriate for use as an on-line algorithm. In the next subsection, we show how this off-line algorithm can be converted into an on-line algorithm that runs in time $n + o(n)$.

6.2.2 Routing on Two-Dimensional Networks

In order to get a running time of $n + o(n)$, we modify the above algorithm in such a way that the routing schedule can be computed on-line in time $o(n)$. Executing the computed schedule then takes another $n + o(n)$ steps. The key idea in our construction is a technique to reduce the size of the scheduling problem that has to be solved, and thus the size and number of the matchings that have to be computed. Informally speaking, this can be done by partitioning the mesh into a smaller number of processes and resources, and by treating sets of packets with similar sources and destinations as if they were a single packet. This is described more formally in the following.

We partition the network into blocks B_i, $0 \leq i < n^{2-2\alpha}$, of size $n^\alpha \times n^\alpha$, where α is some constant that is smaller, but sufficiently close to 1 (for example, $\alpha = 0.9$). We assume that the blocks B_i are indexed in row-major order. (Thus, B_0 and $B_{n^{2-2\alpha}-1}$ are the blocks in the upper left and lower right corner, respectively.) We now interpret each of the $n^{1-\alpha}$ columns of blocks as a process, and each of the $n^{1-\alpha}$ rows of blocks as a resource. Each process P_i, $0 \leq i < n^{1-\alpha}$, has exclusive ownership of its n^α column buses, while each resource R_j, $0 \leq j < n^{1-\alpha}$, consists of n^α row buses. At most one process is allowed to access a single resource at any point in the algorithm. Thus, a process that has exclusive access to a resource can transmit up to n^α packets across the row buses of the resource in a single step.

We now have to arrange the packets inside the processes in such a way that we can make optimal use of this new configuration. To do so we have to slightly relax the goal of the routing schedule that has to be computed. Rather than requiring each packet to be at its final destination after execution of the schedule, we are content with routing each packet to some position in the $n^\alpha \times n^\alpha$ block that contains its destination. After completion of the schedule, we can then bring the packets to their final destinations by routing locally inside each block.

To arrange the packets for the routing schedule, we sort the blocks into row-major order, where the packets are sorted by the index of their destination block. We say that a row of a block B_i is *clean* if all its packets have the same destination block. Otherwise, we say that the row is *dirty*. All n^α packets in a clean row of a block are transmitted across the row buses to their common destination block in a single step, after they have been routed to the correct row of blocks in the preceding step. If a row of a block is dirty, then the packets in the row are transmitted across the row buses to their respective destination blocks in r separate steps, where r is the number of distinct destination blocks that occur among the packets in the row. In other words, such a row is treated in the same way as r separate rows; this increases the number of steps required to route this row by $r - 1$.

Since there are only $n^{2-2\alpha}$ blocks, this increases the number of steps required to route the elements of a single block across the row buses by at most $n^{2-2\alpha} - 1$. Consequently, the number of steps required to route all the elements of a process P_i across the row buses is increased by less than $n^{3-3\alpha}$. Hence, if $D_{i,j}$ denotes the number of steps that process P_i needs resource R_j, then

$$\sum_{j=0}^{n^{1-\alpha}-1} D_{i,j} < n + n^{3-3\alpha} \tag{6.3}$$

holds for all processes P_i. Correspondingly, it can shown that

$$\sum_{i=0}^{n^{1-\alpha}-1} D_{i,j} < n + n^{3-3\alpha} \tag{6.4}$$

holds for all resources R_j, since for any two blocks B_k, B_l, there can be at most two dirty rows in B_k that contain packets destined for B_l. Equations (6.3) and (6.4) guarantee the existence of a schedule of length at most $n + n^{3-3\alpha} = n + o(n)$ that routes every packet to its destination block.

It remains to show that such a schedule can be computed in time $o(n)$. Since we only have $n^{1-\alpha}$ processes and resources, the graph G that is used in the construction of the schedule has only $2n^{1-\alpha}$ vertices. Hence, a maximum matching in this graph can be computed in time $O\left(\left(n^{1-\alpha}\right)^{5/2}\right)$. For each matching that is computed, at least one edge is removed from the graph. This implies that at most $n^{2-2\alpha}$ matchings have to be computed, and the total time to compute the schedule sequentially is bounded by $O\left(\left(n^{1-\alpha}\right)^{9/2}\right) = o(n)$.

In order to implement this computation on a mesh with buses, all the data needed to construct the graph G is routed on the buses to a small area, say in the center of the mesh, where the schedule is computed and then broadcast to all blocks. It suffices if each block contributes the numbers m_i, $0 \leq i < n^{2-2\alpha}$, where m_i is defined as the number of elements in the block that are destined to block B_i. This can clearly be done in time $o(n)$, since only a small amount of information has to be transmitted. We do not elaborate any further on the implementation of the maximum matching algorithm on the mesh. Since we do not need an algorithm that is faster than the sequential one, this is an easy task. (In fact, we could even afford a straightforward simulation of a turing machine algorithm on the mesh; this could be done with a queue size of one.) All in all, we obtain the following algorithm:

Algorithm ROUTE:

(1) Partition the mesh into blocks of size $n^\alpha \times n^\alpha$. Sort the packets in each block into row-major order by destination blocks. This takes $O(n^\alpha)$ steps.

(2) In each block, compute the m_i, $0 \leq i < n^{2-2\alpha}$ (m_i was defined as the number of packets with destination block B_i). Send the m_i to a block of side length $n^{2-2\alpha}$ in the center of the mesh. This takes $O(n^{2-2\alpha})$ steps.

(3) Compute the schedule and broadcast it to all blocks of the mesh. This takes $O\left((n^{1-\alpha})^{9/2}\right)$ steps.

(4) Execute the computed schedule of length $n + n^{3-3\alpha}$.

(5) Perform local routing inside each block to bring the packets to their final destinations. This takes time $O(n^{\alpha})$.

It remains to show that the above algorithm can be implemented with a small, constant queue size. Consider any destination block B_i inside the mesh, and recall that up to n^{α} packets enter B_i across the row buses in a single step. Due to the sorting in Step (1) of the algorithm, every block in the mesh can have at most two dirty rows that contain elements with destination block B_i. This implies that B_i only receives packets in at most $n^{\alpha} + 2n^{2-2\alpha}$ steps of the schedule. If we require that the packets arriving in the ith such step are stored by the processors in the $(i \bmod n^{\alpha})$th column of B_i, then most processors in B_i only get a single packet, while up to $2n^{2-\alpha}$ processors receive two packets. In addition, every processor in B_i can also contain one packet with source in B_i that has not been sent out yet. Finally, some of the processors in B_i, say those on the diagonal of the block, also have to store the n^{α} packets that can enter the block across the column buses in each step, and that are then routed across the row buses in the following step. This gives a total queue size of 4.

We can decrease the queue size to 3 by assuming that the elements in the diagonal of B_i do not receive any of the packets entering the block across the row buses. To get a queue size of 2, we require that every destination block stops accepting new packets from the row buses after it has received $n^{\alpha} - 1$ batches of packets. It can be shown that every block is still able to deliver the vast majority of its packets to their destination blocks. We can now rearrange the packets in each block, and then deliver the remaining packets; the details of this construction are omitted. This establishes the following result.

Theorem 6.2.1 *There exists a deterministic algorithm for permutation routing on the $n \times n$ mesh with buses that runs in time $n + o(n)$ with a queue size of 2.*

Note that the above algorithm does not assume any particular model of the mesh with row and column buses. In fact, the algorithm can be efficiently implemented on a variety of different classes of networks. For the mesh with fixed buses, this improves upon the best previously known deterministic algorithm [73] in both running time and queue size. As an example, the algorithm in [73] requires a queue size of more than 200 to obtain a running time of $1.2n$. (However, the algorithm is not as fast as the independently discovered algorithms of Sibeyn, Kaufmann, and Raman [107].) On the mesh with reconfigurable buses, our algorithm improves upon the best previously known randomized algorithm of Rajasekaran and McKendall [97], and matches the bisection lower bound, within a lower order additive term.

The algorithm can also be easily adapted to the Polymorphic Torus network described in [75]. (This network is essentially a mesh with reconfigurable row and column buses and additional wrap-around connections.) The resulting algorithm routes any permutation in time $n/2 + o(n)$, and thus nearly matches the bisection lower bound of $n/2$.

For another example, consider a model of the mesh with fixed buses in which the buses have a non-unit propagation delay $\rho(n)$. It was observed by Cheung and Lau [16] that, for any non-constant delay function ρ, routing takes time $2n - o(n)$ in this model, assuming that no pipelining is allowed on the buses. However, if we lift this restriction and allow a processor that sends a packet on the bus to send another packet in the next step, then we can route in time $n + o(n)$, for any $\rho = o(n)$, using a variant of the above algorithm.

The above result shows that for the problem of permutation routing, even a fairly simple algorithm on the mesh with buses can achieve a speed-up by a factor of

2 over meshes without buses. Moreover, our algorithm has a queue size of 2. In this context, we point out that the $3n-3$ step off-line scheme for routing on the standard mesh described by Annexstein and Baumslag [4], as well as the $3n + o(n)$ sorting algorithm of Schnorr and Shamir [102], achieve a queue size of 1 only because in the standard mesh model two packets can be exchanged across an edge in a single step. Since we do not allow two arbitrary processors that are connected to a common bus to exchange two packets in a single step, it seems difficult to design any algorithm with a queue size of 1 that uses the buses to transmit packets.

An even greater speed-up over the standard mesh can be achieved for certain restricted classes of permutations. Consider a partial permutation with only a small number of packets (say, at most ϵn^2). In the case of the standard mesh, this problem still requires a running time of $2n-2$ in the worst case. On the mesh with buses, our only restriction is the bisection bound, and hence we could hope for a speed-up of up to $1/\epsilon$ over full permutation routing. The above algorithm can be adapted in such a way that it achieves this bound for any constant ϵ, provided that the sources and destinations of the packets are approximately evenly distributed over the mesh.

In the following, a partial permutation with no more than ϵn^2 packets is called an ϵ-permutation. We say that an ϵ-permutation is δ-approximate if every $m \times m$ block of the mesh is the source and destination of at most $\epsilon m^2 + \delta$ packets, for all m with $1 \leq m \leq n$. Then the following holds for all $\epsilon > 0$.

Corollary 6.2.1.1 *For any $\delta = o(n)$, there exists a deterministic algorithm that routes every δ-approximate ϵ-permutation in time $\epsilon n + o(n)$ with a queue size of* 2.

6.2.3 Routing on Multi-Dimensional Networks

In the following, we apply the techniques from the previous subsection to obtain an improved deterministic algorithm for routing on multi-dimensional meshes with buses. On a d-dimensional network with side length n, our algorithm achieves a running time of $(2 - 1/d)n + o(n)$ and a queue size of 2. This bound also holds for a limited range of non-constant dimensions, provided that the side length n is sufficiently larger than the dimension d.

Our algorithm is based on a well-known scheme for off-line routing on d-dimensional meshes described by Annexstein and Baumslag [4]. The routing scheme consists of $2d - 1$ phases. In phase i, $1 \leq i \leq d - 1$, each packet is routed along dimension i to an appropriately chosen intermediate location. In phase i, $d \leq i \leq 2d - 1$, each packet is greedily routed along dimension $2d - i$. Each phase of the routing scheme involves a collection of routing problems on linear arrays of length n, and thus takes at most n steps on the standard mesh. Hence, the entire routing is completed after $(2d-1)n$ steps. This bound can be matched on meshes with buses, even if only buses are used to route the packets. (On the mesh with fixed buses, this running time can easily be reduced by using the $2n/3$ time algorithm of Leung and Shende [72] to perform the linear array routing.)

In order to route a given permutation with the above routing scheme, it is necessary to determine appropriate choices for the intermediate locations assumed by the packets in the first $d-1$ phases. The existence of such intermediate locations is implied by Hall's Matching Theorem, and they can be computed by constructing a sequence of perfect matchings in a graph; the details of this construction can be derived from the description in Section 1.7.5 of [66]. While the routing is similar in this respect to the scheme studied in the previous subsection, it is also important to realize the differences between the two schemes. In particular, we are not aware of any interpretation of the d-dimensional scheme as an instance of the Open Shop Scheduling Problem. On the other hand, it does not appear to be possible to generalize the two-dimensional scheme to higher dimensions.

Fast algorithms for computing appropriate intermediate locations are given in [74]. For our purposes, it suffices that the running time of these computations is polynomial in n^d, the number of packets in the network. In order to convert the above off-line routing scheme into an on-line algorithm, we introduce the notion of a *super-packet.* Informally speaking, a *super-packet* consists of a collection of packets that have similar sources and destinations, and that move in lock step. By combining a large number of packets into a single super-packet, we are able to decrease the number of packets in the network (and thus the number and size of the matchings that have to be computed) in such a way that the intermediate locations can be computed in time $o(n)$.

Formally, partition the mesh into d-dimensional blocks of side length n^{α}, for some α close to 1 (say $\alpha = 0.99$). Then sort the packets in each block according to their destination blocks, and combine up to $n^{d\beta}$ packets with a common destination block into a single super-packet, say for $\beta = 0.9$. Thus, the packets in a super-packet can be arranged in a d-dimensional submesh of side length n^{β}. In each block, we obtain at most $n^{d(\alpha-\beta)} + n^{d(1-\alpha)}$ super-packets. Hence, the number of super-packets in the entire mesh is $n^{d(1-\beta)} + o(n^{d(1-\beta)})$. We can assign to each super-packet a unique block of side length n^{β} inside the correct destination block, by running an appropriate prefix computation. We can now interpret the remaining problem as a $o(n^{d(1-\beta)})$-approximate permutation routing problem on a d-dimensional mesh with side length $n^{1-\beta}$, where each communication step takes time n^{β} (since it takes n^{β} steps to move all packets of a super-packet using $n^{(d-1)\beta}$ buses). Due to the small number of super-packets in this new routing problem, we can now compute the intermediate locations for the above routing scheme in time $o(n)$. This directly implies an on-line algorithm for routing on d-dimensional buses with a running time of $(2d-1)n + o(n)$.

An issue we have ignored in the above description is that by combining the packets into super-packets, we only get an $o(n)$-approximate permutation and not a permutation in the strict sense. This problem can be easily overcome by, for example, first routing a (partial) permutation containing the vast majority of the packets with the above algorithm; the few remaining packets can then be routed in

$o(n)$.

In the remainder of this subsection, we show how this algorithm can be modified to run in time $(2 - 1/d)n$. Note that the above algorithm only uses a small part of the available bandwidth, since at any point in time all communication is performed across a single dimension. In order to obtain an algorithm whose running time does not grow linearly with the dimension d, we have to make simultaneous use of all the buses in the network. The basic idea to achieve this is to partition the packets of the routing problem into d sets of packets. Each set of packets can then be routed in time $(2d-1)n/d + o(n)$ using the above algorithm. Since that algorithm uses only a single dimension in each time step, we can route all d sets of packets simultaneously without increasing the running time.

Formally, we partition the mesh into d subnetworks by assigning the label j to each processor with coordinates $(i_0, \ldots, i_{d-1})$ and $i_0 + \cdots + i_{d-1} = j \bmod d$. Next, we partition the packets of the routing problem into d sets by first sorting the packets in each block of side length n^α by destination blocks, as before, and then assigning each packet with rank j in the block to set $j \bmod d$. Note that in this way, for any destination block B, we have an approximately equal number of packets with destination in B in each of the d sets of packets. There are n^d/d packets in each set, and hence in each set there are approximately $n^{d\alpha}/d$ packets with destination block B. Next, we move the packets in set i, $0 \leq j < d$, to the subnetwork consisting of the processors with label i, such that each processor receives exactly one packet. Note that all packets with common source and destination blocks are approximately evenly distributed among the d sets.

We now route every set of packets by first routing it within its subnetwork to the correct destination block, and then within the destination block to its final position. We can simultaneously perform the routing in each of the subnetworks by running a "copy" of the above uni-axial algorithm in each subnetwork. in such a way that no two subnetworks communicate across the same dimension at any point in time.

Due to their special structure, each of the d disjoint subnetworks is connected to all dn^{d-1} buses, and can use all n^{d-1} buses associated with a particular dimension in a single step. Since every subnetwork only contains n^d/d packets, each of the $2d-1$ phases of the uni-axial algorithm only requires n/d steps. The queue size of this algorithm is 4. Using ideas similar to those in the previous subsection, the queue size can be reduced to 2. This gives us the following result.

Theorem 6.2.2 *There exists a deterministic algorithm for routing on d-dimensional meshes with buses that runs in time* $(2-1/d)n + o(n)$ *with a queue size of* 2.

The exact lower order term depends on the algorithm used in the computation of the intermediate locations of the packets. The algorithm can be efficiently implemented on a variety of multi-dimensional networks. An even faster algorithm for reconfigurable networks is presented in the next section in the context of sorting.

6.2.4 Fast Routing without Matching

While the routing algorithms described in the previous subsections are fast from a theoretical point of view, they are certainly not efficient in practice. One source of this inefficiency are the fairly large additive lower order terms in the running times of the algorithms. As an example, choosing $\alpha = 9/11$ results in a lower order term of $O(n^{9/11})$ in the case of the two-dimensional algorithm. As the constant hidden by the big-Oh notation is sufficiently large, this lower order term would dominate the running time of the algorithm on networks of realistic size. Another source of inefficiency is the complicated control structure of the algorithm, especially in the computation of the matchings. In particular, this makes the algorithm unsuitable for any implementation in hardware.

In the following, we describe an $n+O(n^{2/3})$ time algorithm for two-dimensional networks that does not require any computation of matchings, and that uses only prefix computations and local sorting as subroutines. Like the algorithm in Subsection 6.2.2, it is based on the off-line algorithm of Leung and Shende, and assumes that the network is partitioned into blocks of side length n^α, for some α. However, instead of computing an optimal schedule for the usage of the buses, the algorithm computes an assignment of the row buses to the columns of blocks (and of the column buses to the rows of blocks) that stays fixed throughout most of the algorithm. In this assignment, each column of blocks receives in each row of blocks a number of row buses that is proportional to the number of its packets that have a destination in this row of blocks. (Alternatively, the algorithm can also be described as computing an approximate solution for a special case of the *Open Shop Scheduling Problem*.)

Let $\alpha = 2/3$, let $s_{i,j}$ denote the number of packets in the ith column of blocks whose destination is in the jth row of blocks, and let $b_{i,j} = \lfloor \frac{s_{i,j}}{n} \rfloor$. We now assign $b_{i,j}$ row buses in the jth row of blocks to the ith column of blocks, and $b_{i,j}$ column buses in the ith column of blocks to the jth row of blocks. Note that

$$\sum_{i=0}^{n^{1/3}-1} b_{i,j} \leq n^{2/3}$$

holds for all j, $0 \leq j < n^{1/3}$, and

$$\sum_{j=0}^{n^{1/3}-1} b_{i,j} \leq n^{2/3}$$

holds for all i, $0 \leq i < n^{1/3}$. This assures that the total number of buses assigned in each row of blocks and each column of blocks does not exceed $n^{2/3}$. Such an assignment of the row buses to the columns of blocks, and of the column buses to the rows of blocks, can be easily computed from the $b_{i,j}$ using prefix computations.

After the assignment of the buses has been computed, we run the following protocol for $n+1$ steps. In each step, $b_{i,j}$ column buses in the ith column of blocks are used to transmit $b_{i,j}$ packets (with destination in the jth row of blocks) to the jth row of blocks. Also, in each step, $b_{i,j}$ row buses in the jth row of blocks are used to transmit $b_{i,j}$ packets to their destination blocks. Thus, all packets routed along the columns in step k are routed along the rows to their destination blocks in step $k+1$. (We assume that the row buses are idle during the first step of the protocol, and the column buses are idle during the last step.)

After $n+1$ steps of the above protocol, there are at most $s_{i,j} - n \cdot b_{i,j} < n$ untransmitted packets in the ith column of blocks that have a destination in the jth row of blocks. We can now transmit these remaining packets by setting $b_{i,j} = n^{1/3}$ for all i, j, and running the above protocol for another $n^{2/3} + 1$ steps. Finally, local routing inside each block can be used to bring every element to its final destination. Altogether, we obtain the following algorithm.

Algorithm ROUTE2:

(1) Partition the mesh into blocks of side length $n^{2/3}$. Use local sorting and prefix computations to compute the assignment of the buses. This takes time $O(n^{2/3})$.

(2) Run the protocol described above for $n+1$ steps.

(3) Set $b_{i,j} = n^{1/3}$ for all i, j, and run the protocol for another $n^{2/3} + 1$ steps.

(4) Perform local routing inside each block to bring the packets to their final destinations. This takes time $O(n^{2/3})$.

One important detail has been omitted from the description so far. Before running the protocol in Steps (2) and (3), we have to arrange the packets inside the blocks such that, for all i, j, all $b_{i,j}$ row and column buses can be used in any step, and such that no write conflicts occur. This can be done in time $O(n^{2/3})$ using local sorting and prefix computations. This establishes the following result.

Theorem 6.2.3 *There exists a deterministic algorithm for permutation routing on the $n \times n$ mesh with buses that runs in time $n + O(n^{2/3})$ with constant queue size and that uses only prefix computations and local sorting as subroutines.*

The above algorithm achieves a queue size of 4; this can be reduced to 2 by a more careful (but also more involved) implementation. The algorithm can again be implemented on several different models of meshes with buses. Of course, it needs to be pointed out that the algorithm is still too complicated to be of immediate practical interest. However, we believe that the result is interesting in that it indicates that even very simple global operations such as prefix computations might be useful in the design of efficient routing algorithms on meshes with buses. In contrast, all previously described algorithms for these networks use the buses only for the transmission of the packets, and not for the computation of the routing schedule. While such a restriction to local control is appropriate for networks that do not provide any fast global communication, it may be that some amount of global control is useful on networks that support fast (but low bandwidth) global primitives such as prefix computations.

6.3 Sorting

In this section we describe two algorithms for sorting on meshes with buses. The first algorithm makes use of the routing algorithms given in the previous section, and its running time (nearly) matches that for permutation routing on all models of meshes with buses. The second algorithm assumes a mesh with reconfigurable buses, and its running time matches the bisection lower bound for networks of any dimension $d = o(n^{1/3})$. Thus, this algorithm also implies an improved bound for permutation routing on reconfigurable networks.

Recall that in the sorting problem we have to move the element of rank i to the processor with index i, for all i. Sorting algorithms on meshes and related networks are usually designed with a particular indexing of the processors in mind. In the following we assume a *blocked* indexing scheme, in which the network is partitioned into blocks of side length n^{δ}, $2/3 \leq \delta < 1$, and the processors in each block have consecutive indices, while the blocks are indexed in snake-like row-major order.

6.3.1 Sorting by Deterministic Sampling

The following algorithm uses the deterministic sampling technique described in Subsection 4.5.2, which computes a set of splitter elements whose ranks are determined to within an additive lower order term. This essentially reduces the problem of sorting to that of routing an appropriate permutation, plus some local operations. The structure of the algorithm is as follows:

Algorithm SORT:

(1) Sort each block of side length n^δ into row-major order. This takes time $O(n^\delta)$.

(2) Route copies of the elements in the first column of each block to a block B of side length $n^{1-\delta/2}$ in the center of the mesh. This takes time $O(n^{1-\delta/2})$.

(3) Sort the elements in B and select n^δ elements of equidistant ranks as splitter elements. This takes time $O(n^{1-\delta/2})$.

(4) Use prefix computations to compute the exact ranks of the splitters, and broadcast them to all blocks. This takes time $O(n^\delta)$.

(5) It was shown in Theorem 4.5.1 that the ith splitter element has a rank between $(i-1)\cdot n^{2-\delta}$ and $i \cdot n^{2-\delta}$. Hence, every element can now determine its rank to within $n^{2-\delta} = O(n^{2\delta})$. Using prefix computations, we can assign each element a preliminary destination that is at most one block away from its final destination.

(6) Route every element to its preliminary destination using the routing algorithm in Subsection 6.2.2 (or any other routing algorithm).

(7) Perform local sorting between consecutive blocks. This takes time $O(n^\delta)$.

Apart from Step (6), all steps of the above algorithm take time $o(n)$. Thus, the running time of the algorithm is determined by the running time of the routing algorithm used in Step (6), up to a lower order term. For $\delta = 2/3$, we get the following result.

Theorem 6.3.1 *For all models of meshes with buses, there exists a sorting algorithm whose running time matches that for permutation routing, within $O(n^{2/3})$ steps.*

6.3.2 Sorting on Meshes with Reconfigurable Buses

Our second algorithm is based on a variation of Leighton's Columnsort algorithm [64] similar to that described in [47, 59]. The algorithm can be efficiently implemented on several classes of meshes with reconfigurable buses, and also on the Mesh of Trees [66] and the Packed Exponential Connections [49], but it does not give improved bounds on meshes with fixed buses.

We start out by describing how the class of κ-way unshuffle permutations can be efficiently solved on a linear array with a reconfigurable bus. We then give the sorting algorithm for networks of arbitrary dimension, and explain how it can be implemented through a sequence of κ-way unshuffle permutations on linear arrays.

Recall that, for any $n, \kappa > 0$ with $n \bmod \kappa = 0$, the κ-way unshuffle permutation on n elements is defined as the permutation π_κ that moves the element in position i to position $\pi(i) = (i \bmod \kappa) \cdot n/\kappa + \lfloor i/\kappa \rfloor$, for all i in $[n]$. We observe that if $\kappa = n^{1-\delta}$ for some $\delta \geq 1/2$, then a κ-way unshuffle permutation on a linear array of length n has the effect of distributing the elements of each block of length n^δ evenly over all $n^{1-\delta}$ blocks of length n^δ.

Due to bisection arguments, at least $n/2$ steps are required to route an $n^{1-\delta}$-way unshuffle permutation on a linear array with a reconfigurable bus. The following routing scheme matches this bound, within a lower order term. The routing scheme consists of two parts. In the first part, we route all packets that have to move to the right; in the second part, we route all packets that have to move to the left. Since the two parts are symmetric, we only describe how to route the rightgoing packets.

The schedule for the rightgoing packets is divided into $n^{1-\delta}/2$ phases P_i, $0 \leq i < n^{1-\delta}/2$. Phase P_i of the schedule consists of $n^{1-\delta} - 2i$ subphases $S_{i,j}$, and each subphase takes $n^{2\delta-1}$ steps. Thus, the entire schedule has a length of

$$\sum_{i=0}^{(n^{1-\delta}/2)-1} \left(n^{1-\delta} - 2i\right) \cdot n^{2\delta-1} \;=\; \frac{n}{2} - 2n^{2\delta-1} \sum_{i=0}^{(n^{1-\delta}/2)-1} i \;=\; \frac{n}{4} + \frac{n^{\delta}}{2}.$$

Given a partition of the array into $n^{1-\delta}$ blocks of length n^{δ}, we say that block i sends to block j if all packets in block i that have a destination in block j are transmitted to this destination. Note that for all i and j, exactly $n^{2\delta-1}$ packets in block i have a destination in block j. Under our schedule, the rightgoing packets are transmitted according to the following rules:

(a) In any subphase $S_{i,0}$, block i sends to block $n^{1-\delta} - i - 1$.

(b) In any subphase $S_{i,j}$ with $1 \leq j < n^{1-\delta} - 2i$, block i sends to block $i + j - 1$, while block $i + j$ sends to block $n^{1-\delta} - i - 1$.

The following sorting algorithm for d-dimensional networks assumes a blocked indexing scheme with blocks of side length n^{δ}, $\delta = 2/3$. The algorithm alternates local sorting and communication steps. Each communication step performs a *total exchange* operation among the blocks. The *total exchange* operation, also often called *all-to-all personalized communication*, is a well-known communication problem that arises in a number of parallel applications (e.g., see Section 1.3 of [9]).

(1) Sort the elements inside each block. This takes time $O(d \cdot n^{\delta})$ using, say, the k-k sorting algorithm for the standard mesh described in [47, 59].

(2) Perform a total exchange among the blocks, where block i sends the $n^{d(2\delta-1)}$ elements with a local rank of $j \bmod n^{d(1-\delta)}$ to block j, for all i, j.

(3) Sort the elements inside each block.

(4) Perform a total exchange among the blocks, where block i sends the $n^{d(2\delta-1)}$ elements with a local rank between $j \cdot n^{d(2\delta-1)}$ and $(j+1) \cdot n^{d(2\delta-1)} - 1$ to block j, for all i, j.

(5) Perform local sorting between consecutive blocks. This takes time $O(d \cdot n^{\delta})$.

After Step (4) of the algorithm, every element is at most one block away from its final destination (see [47] for a proof of this claim). Thus, the local sorting in Step (5) moves each element to its final destination. Steps (2) and (4) can be implemented by performing an appropriate local permutation in each block, followed by d consecutive $n^{1-\delta}$-way unshuffle permutations, where the ith unshuffle permutation is applied to all linear arrays in direction of the ith dimension. However, using this simple approach we only get a running time of $dn/2 + o(n)$ for each of Steps (2) and (4), since at any point in time only buses along a single dimension are being used.

To overcome this problem, we partition the mesh into d subnetworks, where the νth subnetwork consists of all processors with coordinates $(x_0, \ldots, x_{d-1})$ such that $\sum_{i=0}^{d-1} x_i = \nu \bmod d$. We also partition the set of elements into d subsets, such that each subset contains exactly $n^{d(2\delta-1)}/d$ elements that have to be sent from any block i to any block j.

Each linear array inside a subnetwork has a length of n/d, and can hence perform an unshuffle permutation in time $\frac{n}{2d} + o(n)$. We can now implement Steps (2) and (4) in $n/2 + o(n)$ steps each, by routing each subset within its corresponding subnetwork, where the ith unshuffle permutation is applied to the elements of the jth subset in direction of dimension $(i+j) \bmod d$. This gives the following result.

Theorem 6.3.2 *For any $d = o(n^{1/3})$, there exists a deterministic sorting algorithm for d-dimensional meshes with reconfigurable buses that runs in time $n + o(n)$ with queue size two.*

For the Polymorphic Torus, and the mesh with two unidirectional reconfigurable buses of [19], we can obtain a running time of $n/2 + o(n)$, by simultaneously routing the leftgoing and rightgoing elements in the unshuffle permutation. The same bound can be achieved on the Mesh of Trees. We can also adapt the algorithm to run in time $\frac{n}{2\lg n} + o(\frac{n}{\lg n})$ on the Packed Exponential Connections [49] of arbitrary constant dimension. In all of these cases, the algorithm nearly matches the bisection bound.

Finally, we point out that it is straightforward to adapt the algorithm to the k–k sorting problem, in which each processor is the source and destination of k packets. (For $k < 1$, the k–k sorting problem can be defined in a similar way as the δ-approximate ϵ-permutations in Subsection 6.2.2.) The resulting algorithm matches the bisection lower bound within an additive lower order term for all k with $k = \Omega(1/n^{1-c})$ for some $c > 0$.

Theorem 6.3.3 *For any constant $c > 0$, and for any k, d with $k = \Omega(1/n^{1-c})$ and $d = o\left((n \cdot k^{1/d})^{1/3}\right)$, there exists a k–k sorting algorithm for d-dimensional meshes with reconfigurable buses that runs in time $kn + o(kn)$ with queue size two.*

6.4 Summary and Open Problems

In this chapter, we have described deterministic algorithms for permutation routing and sorting on several models of meshes with fixed and reconfigurable buses.

While the routing algorithms in Section 6.2 are based on fairly simple ideas, they are impractical due to their complicated control structure and the large lower order terms in the running times. It is an open question whether the ideas of this chapter can be used in the design of more practical algorithms.

Another possible research direction is to find efficient algorithms for routing with locality, or for the routing of sparse or irregular communication patterns. In this context, the buses might be helpful in the design of algorithms that adapt to the degree of locality, sparseness, and irregularity of a problem. One possible approach to this problem is to first show the existence of a good off-line solution, and then try to convert this off-line solution into an on-line algorithm using the ideas presented in this chapter.

Chapter 7

Concluding Remarks

In this thesis we have established lower bounds for several classes of sorting networks and algorithms, and have described techniques and algorithms for packet routing and sorting on meshes and related networks. In the following, we discuss a few open questions in the context of our work.

The results of Chapter 2 are related to two central open questions in the theory of parallel sorting. One question asks for the existence of an $O(\lg n)$-time deterministic sorting algorithm for the hypercube. The other question concerns the existence of $O(\lg n)$-depth sorting networks that are "more practical" than the AKS network. (By this we mean sorting networks that have a simpler structure or a smaller associated constant, or that can be mapped efficiently to common classes of fixed-connection networks.) While both of these questions remain open, the results in Chapters 2 and 3 provide negative answers for some classes of algorithms and networks that have been considered in this context.

Another important open question concerns the average-case complexity of Shellsort algorithms. No general upper and lower bounds for the average case are currently known, and it seems that any progress on this question would require some fundamental new ideas. Even for many of the most common increment sequences, no formal analysis of the average case has been done.

The study of routing and sorting on meshes has received a lot of attention in recent years, and significant progress has been achieved. Optimal or nearly optimal deterministic solutions are now known for a number of problems, and while some open questions remain, overall there seems to be a fairly good understanding of the worst-case complexity of these problems on theoretical mesh models that place no restrictions on the complexity and structure of the algorithms.

However, many important questions remain open for more realistic, and thus more restricted, models of the mesh, and for dynamic and irregular routing problems that do not fit into the framework of permutation routing. Examples of such restricted models are *hot-potato* routing, *oblivious* routing, or models that restrict the adaptivity of the algorithm or require packets to move along paths of minimum length; see [11, 115] for an overview. We believe that future work on meshes will likely focus more on these types of problems.

Appendix A

Proof of Lemma 4.5.5

Lemma 4.5.5 The greedy routing to destination blocks in Step (9) runs in time $n + o(n)$ with constant queue size.

Proof: The routing in Step (9) is initiated by a *Start* signal that is broadcast from the center of the mesh at time $n + o(n)$. All time bounds stated in the following are with respect to the moment at which this signal was sent out. In the following analysis of the routing, we restrict our attention to the lower right quadrant of the mesh.

As stated in the algorithm, we assume the same routing scheme as in the optimal randomized algorithm. In this scheme, every element moves to its destination block in two phases. In the first phase, row elements move inside their current column to their destination row, while column elements move inside their current row to their destination column. In the second phase, the elements move to their destination blocks. If several packets that are in the same phase contend for an edge, priority is given to the element with the farthest distance to travel. In the following, we only consider the routing of the row elements during their first phase, and the routing of the column elements during their second phase. Thus, we are only oncerned with the problem of routing inside the columns; a symmetric argument holds for the routing inside the rows.

Until time $0.5n$, we reserve the entire edge capacity of the columns for row elements that are in their first phase. At time $0.5n$, we start reserving half of the bandwidth of each column for column elements in their second phase. More precisely, starting at time $0.5n$, we reserve half of the capacity of the topmost column edge for column elements in their second phase. Starting in the next step, we reserve half of the capacity of the next column edge for the column elements, until at time $0.75n$ all column edges in the center subquadrant T_3 have half of their capacity reserved for the column elements. At time $0.75n$, we start reserving only a quarter of the capacity for column elements. As before, this change is initially applied only in the topmost column, and then propagated downwards. It will be seen that this guarantees that, once an element has started moving, it is never delayed until it reaches its destination.

Assuming the above routing scheme, we establish Lemma 4.5.5 through a series of five claims. The proof of Claim (5) is based on an informal explanation of the corresponding proof for the optimal randomized sorting algorithm in [42], given to the author by Christos Kaklamanis.

Claim (1): During the first phase of the routing, there are $\frac{n}{2} \pm o(n)$ row elements in each of the leftmost $n/4$ columns of the quadrant, and the destinations of the row elements in each column are evenly distributed over all destination blocks.

Proof: Consider any fixed subquadrant of the mesh after Step (3) of the algorithm. By Lemma 4.5.2, the number of row elements in the subquadrant that are destined to a particular $n^{\alpha} \times n^{\alpha}$ destination block differs by at most $\frac{1}{16}n^{2-2\beta}$ from the number of column elements destined to that block. Lemma 4.5.1 then guarantees that, after the $\left(\frac{n^{1-\beta}}{4}\right)$-way unshuffle of the row and column elements in Step (4), the number of elements destined to any particular destination block D differs by at most $\frac{3}{16}n^{2-2\beta}$ between the row elements in any column of blocks and the column elements in any row of blocks of the subquadrant. After all 16 subquadrants have been overlapped into a single subquadrant, this becomes $3n^{2-2\beta} = o(n^\beta)$. Hence, in each block of size $n^\beta \times n^\beta$, the sorting in Step (8a) has the effect of distributing the row elements with destination block D evenly over the n^β columns, and the column elements evenly over the n^β rows, up to a difference of one. Since there are $\frac{1}{2}n^{1-\beta}$ such blocks in each column of blocks in the quadrant, the number of elements destined to any particular destination block differs by at most $\frac{1}{2}n^{1-\beta}$ between the row elements in any column and the column elements in any row. Since there are only $\frac{1}{4}n^{2-2\alpha}$ destination blocks in each quadrant, every column has $\frac{n}{2} \pm O(n^{3-\beta-2\alpha})$ row elements.

□

Claim (2): The queue size remains constant during the routing in Step (9).

Proof: The proof of this claim is similar to the argument of Subsection 4.4.2. Assume the same assignment of offset values to the counters as in the routing algorithm. It follows from Claim (1) that every column contains $\sim 2n^{2\alpha-1}$ elements destined for any particular destination block. Hence, the counter technique guarantees that at most 2 row elements turn into a row in any processor. More precisely, if every column were to contain exactly $2n^{2\alpha-1}$ elements for each destination block, then exactly one row element would turn in any processor, since no two counters corresponding to the same column and the same row of destination blocks would ever have the same value. Due to the low-order variations in the number of elements, we get a bit of overlap between the counters.

Next, we have to show that the initial assignment of values to the counters ensures that not too many row elements enter their destination block across the same row. Consider a fixed destination block D, and any set of n^α consecutive columns. We will show that the values assumed by those $2n^\alpha$ counters in our set of columns that correspond to destination block D are evenly distributed from 0 to $n^\alpha - 1$. Note that the initial values of these counters are evenly distributed from

0 to $n^\alpha - 1$. Claim (1) can then be used to show that $\sim 2n^{2\alpha-1}$ elements with destination block D turn into any particular row. Hence, $\sim \frac{1}{2}n^\alpha$ elements enter destination block D through any particular row. If, after entering D, each element stops in the first processor that has not yet received a row element, then every processor in D receives at most one row element. This proves that the routing step achieves a constant queue size.

□

Note that in the rest of the sorting algorithm the maximum queue size is clearly bounded by some constant ≥ 16. At the beginning of Step (9), some processors can hold up to 16 elements. During the first phase of the routing, some processors may temporarily have to hold up to 18 packets. In addition, up to 2 row elements and up to 2 column elements might have to turn in the processor. Also, a processor could become the destination of at most one row element and one column element in the second phase of the routing. Another memory slot is needed for the broadcast of the exact splitter ranks in Step (10) of the algorithm. Thus, the total queue size around 25. This bound could probably be slightly improved by a more careful analysis and implementation.

Claim (3): Every column receives $\sim n/4$ column elements in the second phase, and the destinations of these elements are evenly distributed among all destination blocks in that column.

Proof: Since the accuracy of the splitters is $O(n^{2-\delta})$, every destination block receives $n^{2\alpha} \pm O(n^{2-\delta})$ elements. By Lemma 4.5.2, approximately half of these elements are column elements. It was shown in the proof of Claim (2) that in any block of n^α consecutive rows, $\sim 2n^{2\alpha-1}$ column elements of any particular destination block turn into any of the n^α columns passing through that block. Multiplying this by the number of blocks of n^α consecutive rows in the subquadrant (which is

$\frac{1}{4}n^{1-\alpha}$), we conclude that every column receives $\sim \frac{1}{2}n^{\alpha}$ elements with any particular destination block. Multiplying this term by the number of destination blocks in the same column (which is $\frac{1}{2}n^{1-\alpha}$), we can infer that every column receives $\sim \frac{n}{4}$ elements.

□

Claim (4): If a row element reaches its destination row by time $n - r + o(n)$, where r is the distance it has to travel inside the destination row, then the element arrives at its destination block by time $n + o(n)$.

Proof: (Sketch) Consider a routing problem on a linear array with $n/2$ processors and $n/2$ packets, where each processor is the destination of exactly one packet. It is well known that a greedy routing strategy that gives priority to the packets with farther distance to travel delivers all packets within time $n/2 - 1$, even if processors may initially hold an arbitrary number of packets (see, for example, [66, Section 1.7.1]). It can be shown by a simple induction on the number of routing steps that this remains true even if we impose the additional constraint that a packet may not move before time $n/2 - d$, where d is the distance the packet has to travel. We can interpret the routing of the column elements inside the column as such a routing problem on a linear array that is started at time $n/2 + o(n)$. In this case, we have $n/2$ processors, but only $n/4$ packets. Hence, half of the capacity suffices to route all packets. Since the routing problem has the additional properties that all packets start in the first $n/4$ processors, and that the destinations of the packets in every large block of processors are evenly distributed over the entire array, it can be shown that the capacity required for this routing problem can be reduced to a quarter after the first $n/4$ steps.

□

We have now established that the elements reach their destination blocks by time $n + o(n)$, provided that they are not delayed too much in the first phase of the routing. The remainder of the proof gives an analysis of this first phase, in which the row elements are routed inside their column. The lemma then follows immediately from Claim (4) and the following result.

Claim (5): Every row element reaches its destination row by time $n - r + o(n)$, where r is the distance the element has to travel in the destination row.

Proof: (Sketch) Note that the routing of the row elements inside any particular column is independent of the routing in any other column. Thus, we can interpret this routing phase as a routing problem on a linear array, where the destinations of the elements in the array are given by the destination rows, while the priorities of the elements are determined by the total Manhattan distances to the destination blocks. We identify every processor in the lower right quadrant by a pair of coordinates (x, y), where $(0, 0)$ denotes the center of the mesh and $(n/2 - 1, 0)$ denotes the upper right corner of the quadrant. Only the $n/4$ columns passing through the upper left subquadrant are used in this phase. Note that the routing in column i, $0 \leq i < n/4$, is started i steps after the routing in column 0. It can be shown that the time for routing the row elements in column $n/4 - 1$ to their destination blocks gives an upper bound for the time it would take to route the same set of elements in any other column, within a lower order additive term. Hence, in the following we limit our attention to the routing in column $n/4 - 1$.

By Claim (1), we know that there are $\sim n/2$ row elements in the topmost $n/4$ processors of the column, and that the destinations of these elements are evenly distributed over all destination blocks in the quadrant. However, we do not know anything about the distribution of these elements inside the column at the beginning of the routing. Some processors could hold up to 8 row elements, while others could have none. In the following, we limit our attention to the following two distributions of the elements inside the column. In the first distribution Δ_1, all

$\sim n/2$ elements are initially located in the topmost processor of the column, with coordinates $(n/4-1, 0)$. In the second distribution Δ_2, all $\sim n/2$ elements are initially located in processor $(n/4-1, n/4-1)$. Note that neither Δ_1 nor Δ_2 can actually occur in the algorithm, since a single processor has at most 8 row elements at the beginning of the routing. We consider these two distributions here because they provide an upper bound for the routing time of all other distributions. More precisely, the following can be shown. Let Δ be an arbitrary distribution of the elements in the column, and let $T(e, \Delta)$ denote the time to route an element e to its destination row under distribution Δ. Then it can be shown that the inequality $T(e, \Delta) \leq \max\{T(e, \Delta_1), T(e, \Delta_2)\}$ holds for all elements e. Thus, if all packets arrive at their destination rows in time under both Δ_1 and Δ_2, then they also arrive in time under any other distribution.

Now consider distribution Δ_1, where initially all $\sim n/2$ elements are located in the topmost processor of the column. The *Start* signal arrives at this processor $n/4-1$ steps after it was broadcast from the center. Now the elements start moving towards their destination row, where priority is given to those elements that have the farthest distance to travel. In any step up to time $n/2$, one row element leaves the topmost processor and move towards its destination row. Once an element has started moving, it is not delayed until it reaches its destination row. Between time $n/2$ and $3n/4$, only one row element leaves the topmost processor in any two consecutive steps, and from time $3n/4$ to the end of the routing, three elements leave the topmost processor in any four consecutive steps. As before, an element moves to its destination row without being delayed once it has left the topmost processor.

Now consider the set of elements that have to travel a total distance of at least $3n/8$. Due to Claim (1), there are $\sim n/4$ such elements in the column. Since these elements have a higher priority than the rest, all these elements leave the topmost processor between time $n/4$ and $n/2$. By Claim (1), the destination blocks of these elements are evenly distributed over the area of the quadrant that is at least $3n/8$

away from the topmost processor. Using simple geometric arguments, it can be shown that all of these elements reach their destination row in time.

Next, consider the set of elements that have to travel a distance between $n/4$ and $3n/8$. There are $\sim n/8$ of these elements, and they leave the topmost processor between time $n/2$ and time $3n/4$. It can be shown that these elements also reach their destination row in time. Similarly, it can be shown that the set of elements that have to travel a distance of less than $3n/16$ can be routed to their destination rows between time $3n/4$ and time n. The remaining problem is now to find a way to route those elements that have to travel a distance between $3n/16$ and $n/4$. We can solve this problem by observing that the capacity reserved for the column elements between time $n/2$ and $3n/4$ is not completely used up by these elements. The reason is that the rows from which the column elements turn into the column are evenly distributed over the topmost $n/4$ rows of the quadrant. Hence, many of the slots reserved for these elements are not immediately claimed by the column elements, and we can use these empty slots to route row elements that only have to travel a short distance. It can be shown that all remaining row elements can be routed in this way, and that they reach their destination row in time.

This proves that all packets reach their destination row in time under distribution Δ_1. A similar argument can be given for distribution Δ_2.

□

Bibliography

[1] A. Aggarwal. Optimal bounds for finding maximum on arrays of processors with k global buses. *IEEE Transactions on Computers*, 35:62–64, 1986.

[2] M. Ajtai, J. Komlós, and E. Szemerédi. Sorting in $c \log n$ parallel steps. *Combinatorica*, 3:1–19, 1983.

[3] George S. Almasi and Allan J. Gottlieb. *Highly Parallel Computing.* Benjamin/Cummings, Menlo Park, CA, 1994. Second Edition.

[4] F. Annexstein and M. Baumslag. A unified approach to off-line permutation routing on parallel networks. In *Proceedings of the 2nd Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 398–406, July 1990.

[5] S. Assaf and E. Upfal. Fault tolerant sorting networks. *SIAM J. Discrete Math.*, 4:472–480, 1991.

[6] A. Bar-Noy and D. Peleg. Square meshes are not always optimal. In *Proceedings of the 1st Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 138–147, July 1989.

[7] K. E. Batcher. Sorting networks and their applications. In *Proceedings of the AFIPS Spring Joint Computer Conference,* vol. 32, pages 307–314, 1968.

[8] Y. Ben-Asher, D. Peleg, R. Ramaswami, and A. Schuster. The power of reconfiguration. *Journal of Parallel and Distributed Computing*, 13:139–153, 1991.

[9] D. P. Bertsekas and J. N. Tsitsiklis. *Parallel and Distributed Computation: Numerical Methods*. Prentice-Hall, Englewood Cliffs, NJ, 1989.

[10] S. H. Bokhari. Finding maximum on an array processor with a global bus. *IEEE Transactions on Computers*, 33:133–139, 1984.

[11] A. Borodin. Towards a better understanding of pure packet routing. In *Proceedings of the 3rd Workshop on Algorithms and Data Structures*, pages 14–25, 1993.

[12] Y. C. Chen, W. T. Chen, and G. H. Chen. Efficient median finding and its application to two-variable linear programming on mesh-connected computers with multiple broadcasting. *Journal of Parallel and Distributed Computing*, 15:79–84, 1992.

[13] Y. C. Chen, W. T. Chen, G. H. Chen, and J. P. Sheu. Designing efficient parallel algorithms on mesh-connected computers with multiple broadcasting. *IEEE Transactions on Parallel and Distributed Systems*, 1:241–245, 1990.

[14] H. Chernoff. A measure of the asymptotic efficiency for tests of a hypothesis based on the sum of observations. *Annals of Mathematical Statistics*, 23:493–509, 1952.

[15] S. Cheung and F. C. M. Lau. Mesh permutation routing with locality. *Information Processing Letters*, 43:101–105, 1992.

[16] S. Cheung and F. C. M. Lau. A lower bound for permutation routing on two-dimensional bused meshes. *Information Processing Letters*, 45:225–228, 1993.

[17] B. S. Chlebus, M. Kaufmann, and J. F. Sibeyn. Deterministic permutation routing on meshes. In *Proceedings of the 5th Annual IEEE Symposium on Parallel and Distributed Processing*, pages 614–621, December 1993.

[18] V. Chvátal. Lecture notes on the new AKS sorting network. Technical Report DCS–TR–294, Department of Computer Science, Rutgers University, 1992.

[19] J. C. Cogolludo and S. Rajasekaran. Permutation routing on reconfigurable meshes. In *Proceedings of the 4th International Symposium on Algorithms and Computation,* Lecture Notes in Computer Science, volume 762, pages 157–166. Springer, 1993.

[20] R. Cole and C. K. Yap. A parallel median algorithm. *IPL*, 20:137–139, 1985.

[21] A. Condon, R. E. Ladner, J. Lampe, and R. Sinha. Complexity of sub-bus mesh computations. Technical Report # 93–10–02, Department of Computer Science and Engineering, University of Washington, 1993.

[22] A. Condon and L. Narayanan. Upper and lower bounds for selection on the mesh. In *Proceedings of the 6th Annual IEEE Symposium on Parallel and Distributed Processing*, October 1994. to appear.

[23] P. F. Corbett and I. D. Scherson. Sorting in mesh connected multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 3:626–632, 1992.

[24] R. E. Cypher. A lower bound on the size of Shellsort sorting networks. *SIAM J. Comput.*, 22:62–71, 1993.

[25] R. E. Cypher. Theoretical aspects of VLSI pin limitations. *SIAM J. Comput.*, 22:58–63, 1993.

[26] R. E. Cypher and C. G. Plaxton. Deterministic sorting in nearly logarithmic time on the hypercube and related computers. *JCSS*, 47:501–548, 1993.

[27] W. Dobosiewicz. An efficient variation of Bubble Sort. *Information Processing Letters*, 11:5–6, 1980.

[28] M. Dowd, Y. Perl, L. Rudolph, and M. Saks. The periodic balanced sorting network. *JACM*, 36:738–757, 1989.

[29] T. Gonzalez and S. Sahni. Open shop scheduling to minimize finish time. *Journal of the ACM*, 23:665–679, 1976.

[30] Z. Guo, R. G. Melhem, R. W. Hall, D. M. Chiarulli, and S. P. Levitan. Array processors with pipelined optical buses. In *Proceedings of the 3rd IEEE Symposium on the Frontiers of Massively Parallel Computations*, pages 333–342, 1990.

[31] Y. Han and Y. Igarashi. Time lower bounds for parallel sorting on multidimensional mesh-connected processor arrays. *Information Processing Letters*, 33:233–238, 1990.

[32] Y. Han, Y. Igarashi, and M. Truszczynski. Indexing functions and time lower bounds for sorting on a mesh-connected computer. *Discrete Applied Mathematics*, 36:141–152, 1992.

[33] E. Hao, P. D. MacKenzie, and Q. F. Stout. Selection on the reconfigurable mesh. In *Proceedings of the 4th IEEE Symposium on the Frontiers of Massively Parallel Computations*, pages 38–45, 1992.

[34] T. N. Hibbard. An empirical study of minimal storage sorting. *Communications of the ACM*, 6:206–213, 1963.

[35] J. E. Hopcroft and R. M. Karp. An $n^{5/2}$ algorithm for maximum matchings in bipartite graphs. *SIAM Journal on Computing*, 2:225–231, 1973.

[36] J. Incerpi and R. Sedgewick. Improved upper bounds on Shellsort. *Journal of Computer and System Sciences*, 31:210–224, 1985.

[37] J. Incerpi and R. Sedgewick. Practical variations of Shellsort. *Information Processing Letters*, 26:37–43, 1987.

[38] K. Iwama and Y. Kambayashi. An $O(\lg n)$ parallel connectivity algorithm on the mesh. In *Information Processing 89*, pages 305–310, 1989.

[39] K. Iwama, E. Miyano, and Y. Kambayashi. Routing problems on the mesh of buses. In *Proceedings of the 3rd International Symposium on Algorithms and Computation,* Lecture Notes in Computer Science, volume 650, pages 155–164. Springer, 1992.

[40] J. Jang, H. Park, and V. K. Prasanna-Kumar. A fast algorithm for computing histogram on reconfigurable mesh. Technical Report IRIS 290, Institute for Robotics and Intelligent Systems, University of Southern California, 1992.

[41] H. F. Jordan. A special purpose architecture for finite element analysis. In *International Conference on Parallel Processing*, pages 263–266, 1978.

[42] C. Kaklamanis and D. Krizanc. Optimal sorting on mesh-connected processor arrays. In *Proceedings of the 4th Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 50–59, July 1992.

[43] C. Kaklamanis, D. Krizanc, L. Narayanan, and T. Tsantilas. Randomized sorting and selection on mesh-connected processor arrays. In *Proceedings of the 3rd Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 17–28, July 1991.

[44] C. Kaklamanis, D. Krizanc, and S. Rao. Simple path selection for optimal routing on processor arrays. In *Proceedings of the 4th Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 23–30, July 1992.

[45] M. Kaufmann, U. Meyer, and J. F. Sibeyn. Towards practical permutation routing on meshes. In *Proceedings of the 6th Annual IEEE Symposium on Parallel and Distributed Processing*, October 1994. to appear.

[46] M. Kaufmann, S. Rajasekaran, and J. F. Sibeyn. Matching the bisection bound for routing and sorting on the mesh. In *Proceedings of the 4th Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 31–40, July 1992.

[47] M. Kaufmann, J. Sibeyn, and T. Suel. Derandomizing algorithms for routing and sorting on meshes. In *Proceedings of the Fifth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 669–679, January 1994.

[48] M. Kik, M. Kutyłowski, and G. Stachowiak. Periodic constant depth sorting networks. In *Proceedings of the 11th Symposium on Theoretical Aspects of Computer Science*, pages 201–212, February 1994.

[49] W. W. Kirkman and D. Quammen. Packed exponential connections - a hierarchy of 2-D meshes. In *Proceedings of the 5th International Parallel Processing Symposium*, pages 464–470, 1991.

[50] D. E. Knuth. *The Art of Computer Programming*, volume 3. Addison-Wesley, Reading, MA, 1973.

[51] C. P. Kruskal and M. Snir. A unified theory of interconnection network structure. *Theoretical Computer Science*, 48:75–94, 1986.

[52] M. Kumar and D. S. Hirschberg. An efficient implementation of Batcher's odd-even merge algorithm and its application in parallel sorting schemes. *IEEE Transactions on Computers*, 32:254–264, 1983.

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

[54] M. Kunde. Bounds for 1-selection and related problems on grids of processors. In *Proceedings of the 4th International Workshop on Parallel Processing by Cellular Automata and Arrays (PARCELLA)*, pages 298–307. Springer, 1988.

[55] M. Kunde. Routing and sorting on mesh-connected arrays. In J. H. Reif, editor, *VLSI Algorithms and Architectures: Proceedings of the 3rd Aegean Workshop on Computing,* Lecture Notes in Computer Science, volume 319, pages 423–433. Springer, 1988.

[56] M. Kunde. Packet routing on grids of processors. In H. Djidjev, editor, *Workshop on Optimal Algorithms,* Lecture Notes in Computer Science, volume 401, pages 254–265. Springer, 1989.

[57] M. Kunde. Balanced routing: Towards the distance bound on grids. In *Proceedings of the 3rd Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 260–271, July 1991.

[58] M. Kunde. Concentrated regular data streams on grids: Sorting and routing near to the bisection bound. In *Proceedings of the 32nd Annual IEEE Symposium on Foundations of Computer Science*, pages 141–150, October 1991.

[59] M. Kunde. Block gossiping on grids and tori: Deterministic sorting and routing match the bisection bound. In *Proceedings of the 1st Annual European Symposium on Algorithms*, pages 272–283, September 1993.

[60] M. Kutyłowski, K. Loryś, B. Oesterdiekhoff, and R. Wanka. Fast and feasible periodic sorting networks of constant depth. In *Proceedings of the 35th Annual IEEE Symposium on Foundations of Computer Science*, November 1994. To appear.

[61] R. E. Ladner, J. Lampe, and R. Rogers. Vector prefix addition on sub-bus mesh computers. In *Proceedings of the 5th Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 387–396, June 1993.

[62] H. W. Lang, M. Schimmler, H. Schmeck, and H. Schröder. Systolic sorting on a mesh-connected network. *IEEE Transactions on Computers*, 34:652–658, 1984.

[63] R. Lazarus and R. Frank. A high-speed sorting procedure. *Communications of the ACM*, 3:20–22, 1960.

[64] F. T. Leighton. Tight bounds on the complexity of parallel sorting. *IEEE Transactions on Computers*, C–34:344–354, 1985.

[65] F. T. Leighton. Average case analysis of greedy routing algorithms on arrays. In *Proceedings of the 2nd Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 2–10, July 1990.

[66] F. T. Leighton. *Introduction to Parallel Algorithms and Architectures: Arrays, Trees, and Hypercubes.* Morgan-Kaufmann, San Mateo, CA, 1991.

[67] F. T. Leighton. Methods for message routing in parallel machines. In *Proceedings of the 24th Annual ACM Symposium on the Theory of Computing*, pages 77–96, May 1992.

[68] F. T. Leighton and Y. Ma. Breaking the $\Theta(n \lg^2 n)$ barrier for sorting with faults. In *Proceedings of the 34th Annual IEEE Symposium on Foundations of Computer Science*, pages 734–743, November 1993.

[69] F. T. Leighton, F. Makedon, and I. G. Tollis. A $2n-2$ step algorithm for routing in an $n \times n$ array with constant queue sizes. In *Proceedings of the 1st Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 328–335, July 1989.

[70] F. T. Leighton and C. G. Plaxton. A (fairly) simple circuit that (usually) sorts. In *Proceedings of the 31st Annual IEEE Symposium on Foundations of Computer Science*, pages 264–274, October 1990.

[71] F. T. Leighton and C. G. Plaxton. Hypercubic sorting networks. Technical Report TR–94–18, University of Texas at Austin, Department of Computer Science, May 1994. Available via anonymous ftp from `ftp.cs.utexas.edu`.

[72] J. Y. Leung and S. Shende. Packet routing on square meshes with row and column buses. In *Proceedings of the 3rd Annual IEEE Symposium on Parallel and Distributed Processing*, pages 834–837, December 1991.

[73] J. Y. Leung and S. M. Shende. On multidimensional packet routing for meshes with buses. *Journal of Parallel and Distributed Computing*, 20:187–197, 1994.

[74] G. F. Lev, N. Pippenger, and L. G. Valiant. A fast parallel algorithm for routing in permutation networks. *IEEE Transactions on Computers*, 30:93–100, 1981.

[75] H. Li and Q. F. Stout. *Reconfigurable Massively Parallel Computers.* Prentice Hall, Englewood Cliffs, New Jersey, 1991.

[76] N. Linial and M. Tarsi. Interpolation between bases and the shuffle exchange network. *European Journal of Combinatorics*, 10:29–39, 1989.

[77] M. Luby. A simple parallel algorithm for the maximal independent set problem. *SIAM J. Comput.*, 15:1036–1053, 1986.

[78] F. Meyer auf der Heide and H. T. Pham. On the performance of networks with multiple busses. In *Proceedings of the 9th Symposium on Theoretical Aspects of Computer Science*, pages 98–108, 1992.

[79] R. Miller, V. K. Prasanna Kumar, D. I. Reisis, and Q. F. Stout. Parallel computations on reconfigurable meshes. *IEEE Transactions on Computers*, 42:678–692, June 1993.

[80] D. E. Muller and F. P. Preparata. Bounds to complexities of networks for sorting and for switching. *Journal of the ACM*, 22:195–201, 1975.

[81] L. Narayanan. *Selection, Sorting, and Routing on Mesh-Connected Processor Arrays.* PhD thesis, Department of Computer Science, University of Rochester, Rochester, NY, May 1992.

[82] D. Nassimi and S. Sahni. Bitonic sort on a mesh-connected parallel computer. *IEEE Transactions on Computers*, C–28:2–7, 1979.

[83] S. E. Orcutt. *Computer Organization and Algorithms for Very-High Speed Computations*. PhD thesis, Department of Computer Science, Stanford University, September 1974.

[84] A. Papernov and G. Stasevich. A method for information sorting in computer memories. *Problems of Information Transmission*, 1:63–75, 1965.

[85] D. Parker. Notes on shuffle/exchange-type switching networks. *IEEE Transactions on Computers*, 29:213–222, 1980.

[86] M. S. Paterson. Improved sorting networks with $O(\log N)$ depth. *Algorithmica*, 5:75–92, 1990.

[87] N. Pippenger. Communication networks. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science, Volume A: Algorithms and Complexity*, pages 805–833. Elsevier/MIT Press, 1990.

[88] C. G. Plaxton. A hypercubic sorting network with nearly logarithmic depth. In *Proceedings of the 24th Annual ACM Symposium on Theory of Computing*, pages 405–416, May 1992.

[89] C. G. Plaxton, B. Poonen, and T. Suel. Improved lower bounds for Shellsort. In *Proceedings of the 33rd Annual IEEE Symposium on Foundations of Computer Science*, pages 226–235, October 1992.

[90] C. G. Plaxton and T. Suel. A lower bound for sorting networks based on the shuffle permutation. *Mathematical Systems Theory*, 27:491–508, 1994.

[91] C. G. Plaxton and T. Suel. A super-logarithmic lower bound for hypercubic sorting networks. In *Proceedings of the 21st International Colloquium on Automata, Languages, and Programming*, pages 618–629, July 1994.

[92] B. Poonen. The worst case in Shellsort and related algorithms. *Journal of Algorithms*, 15:101–124, 1993.

[93] V. K. Prasanna Kumar and C. S. Raghavendra. Image processing on enhanced mesh connected computers. In *Computer Architecture for Pattern Analysis and Image Database Management*, pages 243–247, 1985.

[94] V. K. Prasanna Kumar and C. S. Raghavendra. Array processors with multiple broadcasting. *Journal of Parallel and Distributed Computing*, 4:173–190, 1987.

[95] V. R. Pratt. *Shellsort and Sorting Networks*. PhD thesis, Stanford University, Department of Computer Science, December 1971. Also published by Garland, New York, 1979.

[96] S. Rajasekaran. Mesh-connected computers with fixed and reconfigurable buses: Packet routing, sorting, and selection. In *Proceedings of the 1st Annual European Symposium on Algorithms*, pages 309–320, September 1993.

[97] S. Rajasekaran and T. McKendall. Permutation routing and sorting on the reconfigurable mesh. Technical Report MS-CIS-92-36, Department of Computer and Information Science, University of Pennsylvania, May 1992.

[98] S. Rajasekaran and R. Overholt. Constant queue routing on a mesh. *Journal of Parallel and Distributed Computing*, 15:160–166, 1992.

[99] S. Rajasekaran and T. Tsantilas. Optimal routing algorithms for mesh-connected processor arrays. *Algorithmica*, 8:21–38, 1992.

[100] K. Sado and Y. Igarashi. Some parallel sorts on a mesh-connected processor array. *Journal of Parallel and Distributed Computing*, 3:389–410, 1986.

[101] I. D. Scherson and S. Sen. Parallel sorting in two-dimensional VLSI models of computation. *IEEE Transactions on Computers*, 38:238–249, 1989.

[102] C. P. Schnorr and A. Shamir. An optimal sorting algorithm for mesh-connected computers. In *Proceedings of the 18th ACM Symposium on Theory of Computing*, pages 255–263, May 1986.

[103] R. Sedgewick. A new upper bound for Shellsort. *Journal of Algorithms*, 7:159–173, 1986.

[104] D. L. Shell. A high-speed sorting procedure. *Communications of the ACM*, 2:30–32, 1959.

[105] J. F. Sibeyn. Desnakification of mesh sorting algorithms. In *Proceedings of the 2nd Annual European Symposium on Algorithms*, pages 377–390, September 1994.

[106] J. F. Sibeyn, B. S. Chlebus, and M. Kaufmann. Shorter queues for permutation routing on meshes. In *Proceedings of the 19th Symposium on the Mathematical Foundations of Computer Science*, pages 597–607, August 1994.

[107] J. F. Sibeyn, M. Kaufmann, and R. Raman. Randomized routing on meshes with buses. In *Proceedings of the 1st Annual European Symposium on Algorithms*, pages 333–344, September 1993.

[108] H. J. Siegel. *Interconnection Networks for Large-Scale Parallel Processing: Theory and Case Studies.* Lexington Books, Lexington, MA, 1984.

[109] Q. F. Stout. Mesh-connected computers with broadcasting. *IEEE Transactions on Computers*, 32:826–830, 1983.

[110] Q. F. Stout. Meshes with multiple buses. In *Proceedings of the 27th Annual IEEE Symposium on Foundations of Computer Science*, pages 264–273, 1986.

[111] T. Suel. Optimal deterministic routing and sorting on mesh-connected arrays of processors. Technical Report TR–93–18, University of Texas at Austin,

Department of Computer Science, October 1993. Available via anonymous ftp from `ftp.cs.utexas.edu`.

[112] T. Suel. Improved bounds for routing and sorting on multi-dimensional meshes. In *Proceedings of the 6th Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 26–35, June 1994.

[113] T. Suel. Routing and sorting on meshes with row and column buses. In *Proceedings of the 8th International Parallel Processing Symposium*, pages 411–417, April 1994.

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

[115] M. Tompa. Lecture notes on message-routing in parallel machines. Technical Report # 94–06–05, Department of Computer Science and Engineering, University of Washington, 1994.

[116] L. G. Valiant. A scheme for fast parallel communication. *SIAM Journal on Computing*, 11:350–361, 1982.

[117] L. G. Valiant and G. J. Brebner. Universal schemes for parallel communication. In *Proceedings of the 13th Annual ACM Symposium on Theory of Computing*, pages 263–277, May 1981.

[118] A. Varma and C. S. Raghavendra. Rearrangeability of multistage shuffle/exchange networks. *IEEE Transactions on Communications*, 36:1138–1147, 1988.

[119] J. S. Vitter and P. Flajolet. Average-case analysis of algorithms and data structures. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science, Volume A: Algorithms and Complexity*, pages 431–524. Elsevier/MIT Press, 1990.

[120] B. Wang and G. Chen. Constant time algorithms for the transitive closure and some related graph problems on processor arrays with reconfigurable bus systems. *IEEE Transactions on Parallel and Distributed Systems*, 1:500–507, 1990.

[121] R. Wanka. Fast general sorting on meshes of arbitrary dimension without routing. Technical Report TR–RI–91–087, Department of Computer Science, University of Paderborn, August 1991.

[122] M. A. Weiss. *Lower Bounds for Shellsort.* PhD thesis, Princeton University, Department of Computer Science, June 1987.

[123] M. A. Weiss. Empirical study of the expected running time of Shellsort. *The Computer Journal*, 34:88–91, 1991.

[124] M. A. Weiss and R. Sedgewick. Bad cases for Shaker-sort. *Information Processing Letters*, 28:133–136, 1988.

[125] M. A. Weiss and R. Sedgewick. Tight lower bounds for Shellsort. *Journal of Algorithms*, 11:242–251, 1990.

[126] A. C. C. Yao. An analysis of $(h, k, 1)$-Shellsort. *Journal of Algorithms*, 1:14–50, 1980.

Vita

Torsten Suel was born in Göttingen, Germany, on April 7, 1966, the son of Sigrid and Anton Suel. After graduating from Gymnasium Am Fredenberg (High School) in Salzgitter, Germany, in 1985, he entered the Technical University of Braunschweig, Germany. He received the degree of Diplom-Informatiker from the Technical University of Braunschweig in August 1990. In August 1990, he entered the Graduate School of the University of Texas at Austin, and received the degree of Master of Science in Computer Science in May 1992.

Permanent Address: Otto-Hahn-Ring 70
38228 Salzgitter
Germany

This dissertation was typeset with LaTeX 2_ε[1] by the author.

[1] LaTeX 2_ε is an extension of LaTeX. LaTeX is a collection of macros for TeX. TeX is a trademark of the American Mathematical Society. The macros used in formatting this dissertation were written by Dinesh Das, Department of Computer Sciences, The University of Texas at Austin.