Optimal Deterministic Routing and Sorting on Mesh-Connected Arrays of Processors

*Torsten Suel**

Department of Computer Sciences
University of Texas at Austin
torsten@cs.utexas.edu

Abstract

In this paper we introduce a new "derandomization" technique for mesh-connected arrays of processors that allows us to convert several recently proposed randomized algorithms for routing and sorting into deterministic algorithms that achieve the same running time, within a lower order additive term. By applying this technique, we obtain a number of optimal or improved deterministic algorithms for meshes and related networks. Among our main results are the first optimal deterministic algorithms for sorting on the two-dimensional mesh and for routing on the two-dimensional torus and the three-dimensional mesh, as well as an optimal deterministic routing algorithm for the two-dimensional mesh that achieves a queue size of 5. The new technique is very general and seems to apply to most of the randomized algorithms for routing and sorting on meshes and related networks that have been proposed so far.

*Supported by Texas Advanced Research Program (TARP) Award #003658480.

1 Introduction

The mesh-connected array of processors is one of the most thoroughly investigated interconnection schemes for parallel processing. While it has a large diameter in comparison to the various hypercubic networks, it is nonetheless of great importance due to its simple and efficient layout and its good performance in practice. Consequently, a number of parallel machines with mesh topology have been designed and built, and a variety of algorithmic problems have been analyzed as to their complexity on theoretical models of the mesh. Probably the two most extensively studied problems are those of routing and sorting. For an introduction into these problems, and a formal definition of the networks considered in this paper, we refer the reader to [20].

In this paper, we give improved algorithms for a number of routing and sorting problems on meshes and related networks. In our presentation, we will mostly focus on the case of the two-dimensional mesh. We will also state a number of results for higher-dimensional meshes and related networks, but due to space constraints we will omit most of the proofs of these results. We will mainly be 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 following, we assume an $n \times n$ mesh-connected array of synchronous processors. Each of the n^2 processors will be 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 *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 a 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 $\{1, \ldots, n\} \times \{1, \ldots, n\}$ to $\{1, \ldots, n^2\}$. If $\mathcal{I}(i, j) = k$ for some processor $(i, j) \in \{1, \ldots, n\} \times \{1, \ldots, n\}$ and some $k \in \{1, \ldots, n^2\}$, 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 $\mathcal{I}(\mathrm{Rank}\,(y, X))$, where $\mathrm{Rank}\,(y, X) \stackrel{\text{def}}{=} |\{x \in X \mid x \leq y\}|$ 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 ordering* 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* orderings. Sorting algorithms on the mesh are usually designed with a particular indexing scheme in mind, and techniques used for one particular indexing scheme may not work well for others. In this paper, we will assume an indexing scheme which can be described as a snake-like column-major indexing nested inside a snake-like row-major indexing, and which was also used in the algorithms in [6, 7]. This indexing scheme is defined by partitioning the mesh into blocks of size $n^\alpha \times n^\alpha$, and using snake-like column-major indexing inside each block, while the blocks are ordered in the mesh according to snake-like row-major indexing.

1.1 Previous Results

The study of sorting on the two-dimensional mesh was initiated by Orcutt [24] and Thompson and Kung [30], who gave algorithms based on Batcher's Bitonic Sort [1] 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, [12, 19, 23, 27, 28]); 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 will 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; a comparison-exchange operation on adjacent packets may be performed in a single step.

For this model of the mesh, Schnorr and Shamir [29] 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 [13]. The same proof technique has also been used to show lower bounds for arbitrary indexing schemes [13]; the best general lower bound is currently $2.27n$ [5]. Interestingly, 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. It has been argued that such a restriction does not reflect the capabilities of existing parallel machines, and that one should allow any constant number of packets to be stored in a single node.

This has motivated the following *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 [7]. Their algorithm requires a queue size of at least 8 (the queue size is the maximum number of packets located in a single processor at any time). Using very different techniques, Kunde [17] 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 regular 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 indexing scheme mentioned earlier.

However, if one is interested in developing an algorithm that comes closer to the distance bound of $2n - 2$, or in faster algorithms for selection, then it seems very difficult to apply the techniques used in Kunde's deterministic algorithm. In fact, Narayanan [22] 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 [7], on the other hand, was subsequently used by Kaklamanis and Krizanc [6] to design an optimal randomized sorting algorithm, with a running time of $2n + o(n)$ and constant queue size.

For the permutation routing problem, Valiant and Brebner [31] proposed a randomized algorithm with a running time of $(2r - 1)n + o(n)$ and a queue size of $O(\lg n)$ on the r-dimensional mesh, $r \geq 2$. 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 [15]. A randomized routing algorithm with running time $2n + o(n)$ and constant queue size was obtained by Rajasekaran and Tsantilas [26]. Subsequently, Leighton, Makedon, and Tollis [21] 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 [25] gave an improved construction that reduced the queue size to about 112. 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.

1.2 Overview of the Paper

In this paper, we introduce a new "derandomization" technique for meshes that allows us to convert several recently proposed randomized algorithms for routing and sorting into deterministic algorithms that achieve the same running time, within a lower order additive term. We describe the main ideas behind the technique and then apply it to an optimal randomized algorithm for routing on two-dimensional meshes proposed by Kaklamanis, Krizanc, and Rao [8]. 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 [21, 25] had a running time of $2n - 2$ and a queue size of at least 112. Extending our result to other networks, we obtain the first optimal deterministic algorithm for routing on three-dimensional meshes, thus answering an open question posed by Leighton [20, p. 271]. Furthermore, we get the first optimal deterministic algorithm for routing on the two-dimensional torus, and a slightly improved algorithm for four-dimensional meshes.

Next, we apply our new technique to the optimal randomized algorithm for sorting on the two-dimensional mesh proposed by Kaklamanis and Krizanc [6]. 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 [17] achieved a running time of $2.5n + o(n)$ and a queue size of 2. As an extension of this result, we also obtain improved algorithms for sorting on three-dimensional meshes and on two-dimensional and three-dimensional tori. We then describe algorithms for multi-packet routing and sorting that match the bounds of the optimal deterministic algorithm recently proposed by Kunde [18]. Finally, we give improved deterministic algorithms for selection on meshes and related networks.

The paper is organized as follows. Section 2 defines some terminology. Section 3 describes the main idea behind our technique, and applies it to obtain improved algorithms for routing. Section 4 contains our results for sorting. Section 5 describes an application of our technique to multi-packet routing and sorting. Section 6 contains our results for selection. Finally, Section 7 lists some open questions for future research.

2 Terminology

Throughout the paper, we will 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.

3 Optimal Routing with Small Queue Size

In this section we describe a number of deterministic algorithms for permutation routing on meshes and tori. Our algorithms are based on a class of randomized routing schemes recently proposed by Kaklamanis, Krizanc, and Rao [8]. We describe a technique that allows us to convert these randomized algorithms into deterministic algorithms with the same running time, within a lower order additive term.

In the first subsection, we give an informal description of this technique. In Subsection 3.2, we apply the technique to obtain a (fairly) simple optimal routing algorithm for two-dimensional meshes. Subsection 3.3 shows how to extend this result to get optimal algorithms for tori and three-dimensional meshes.

3.1 The Basic Idea

In this subsection, we describe the basic idea underlying our technique for converting randomized into deterministic algorithms. All of our routing and sorting algorithms have the property that they first route each packet to an approximate location, 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 $n^\alpha \times n^\alpha$, with $\frac{2}{3} < \alpha < 1$. 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 [15] and Cheung and Lau [2].

Hence, in the following we are only concerned with the problem of moving the packets into their destination blocks. To solve this problem deterministically, we use two fundamental operations, which we will refer to as the *unshuffle operation* and the *counter scheme*. We will use the unshuffle operation to distribute packets with the same destination block evenly over a sufficiently large region of the network. The counter scheme will be employed to make sure that the incoming packets are evenly distributed over the processors of each destination block.

Formally, for any $m, k > 0$ with $m \bmod k = 0$, the k-way unshuffle operation on m positions $0, \ldots, m-1$ is defined as the permutation π_k that moves the element in position i to position $\pi_k(i) \stackrel{\text{def}}{=} (i \bmod k) \cdot m/k + \lfloor i/k \rfloor$. We say that we perform a k-way unshuffle operation on the columns (rows) of a block of the mesh, if we move all elements located in column (row) i of the block to the corresponding positions in column (row) $\pi_k(i)$ of the block, for all i.

The utility of the unshuffle operation for sorting on meshes was previously observed by Schnorr and Shamir, who used it in the design of their $3n + o(n)$ sorting algorithm in the single-packet model. In the following, we will demonstrate that the unshuffle operation can in many cases be employed as a "substitute" for the use of randomness. Following a general scheme originally proposed by Valiant and Brebner [31], many randomized algorithms for routing on meshes start out by moving all packets to random intermediate locations inside the current row (or column). This has the effect of distributing packets with similar destinations evenly over the columns (rows) of the network, with high probability. We observe that this effect can be achieved deterministically by partitioning the mesh into blocks, locally sorting the packets in each block by their destination blocks, and then applying an unshuffle operation to the sorted blocks. Here, we assume that the set of destination blocks is ordered in some arbitrary fixed way, say according to a row-major ordering of the blocks.

Formally, if we partition a mesh into blocks of size $n^\beta \times n^\beta$, $\frac{2}{3} < \beta < 1$, and sort the packets in each block by their destination blocks into row-major ordering, then the following holds after performing an $(n^{1-\beta})$-way unshuffle operation on the columns of the mesh.

Lemma 3.1 Let B_1 and B_2 be any pair of $n^\beta \times n^\beta$ blocks located in the same row of blocks, and let D be a destination block of size $n^\alpha \times n^\alpha$. 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 proof of the above lemma is straightforward and hence omitted. A similar lemma can be shown in the context of sorting (see Lemma 4.1). Informally speaking, the above lemma says that all elements with a common destination block will be evenly distributed over all blocks that are located in the same row of blocks. By repeating the local sorting of the blocks after the unshuffle operation, we can then make sure that all elements with a common destination block are evenly distributed over the columns of the mesh.

When routing the packets into their 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 4.1). In our deterministic algorithms, we will use the counter scheme mentioned above. 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 will receive 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 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.

3.2 Routing on Two-Dimensional Meshes

In this subsection, we show how the above techniques can be used to obtain an optimal deterministic algorithm for $n \times n$ meshes with a queue size of 5. The algorithm is based on an optimal randomized algorithm proposed by Kaklamanis, Krizanc, and Rao [8]. We will first give a brief description of their algorithm, which has a very simple structure.

Partition the mesh vertically into four quarters Q_0 to Q_3, where Q_i contains the columns $i\frac{n}{4}$ to $(i+1)\frac{n}{4} - 1$. In the algorithm 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 [8] 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.

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 unshuffle operation on the columns of the mesh, rather than being chosen at random. We also need a few additional steps for local sorting and routing, and the counter scheme. All in all, our deterministic algorithm consists of the following steps.

Algorithm ROUTE:

(1) Partition the mesh into destination blocks of size $n^\alpha \times n^\alpha$, and let every packet determine its destination block.

(2) Partition the mesh into blocks of size $n^\beta \times n^\beta$, 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 ordering of the blocks.

(3) In each quarter Q_i, perform an $(\frac{n^{1-\beta}}{4})$-way unshuffle operation 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 will be done during the column routing, using the counter scheme briefly described in the previous subsection. 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, $0 \leq j < \frac{1}{2}n^{1-\alpha}$, is assigned the offset value $(i + j \cdot 2n^{2\alpha-1}) \bmod n^\alpha$.

(7) Route the packets along the rows into their destinations 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. It will be shown below that, 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) will 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, will 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 [8]). 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 will terminate 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 3.1 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 will only be distributed over a small fraction of the incoming rows of D. However, if we look at blocks of n^α consecutive columns, then the elements in these columns will be 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 every processor of D will receive at most 2 packets. The maximum possible queue size 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 during the routing in Step (6).

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

Kaklamanis, Krizanc, and Rao [8] also gave 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 3.2 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 [8], Kaklamanis, Krizanc, and Rao show that their randomized algorithm will route 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.

3.3 Extensions

Kaklamanis, Krizanc, and Rao also give optimal randomized and off-line algorithms for tori and three-dimensional meshes. In this subsection, we will give similar extensions for the deterministic case. Due to space constraints, we can only state the results and make a few informal remarks about the constructions. 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 submesh, described in [8]. Together with Lemma 3.2 this gives the following result.

Theorem 3.2 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 [16]. Our approach can also be used to obtain deterministic algorithms for routing in r-dimensional meshes with $r > 3$. Using the unshuffle operation and the counter scheme, we can convert the randomized algorithm of Valiant and Brebner [31] into a deterministic algorithm with a running time of $(2r-1)n + o(n)$. This can be improved to $(2r-3)n+o(n)$ by using the above algorithm for three-dimensional meshes as a subroutine. For $r = 4$, this gives a slight improvement over the fastest previously known algorithm [16], which achieved a running time of $(5 + \epsilon)n$ and a queue size of $O(1/\epsilon)$. For $r \geq 5$, the best upper bounds continue to be given by an algorithm of Kunde [16], which has a running time of $(r + (r-2)(1/r)^{1/(r-2)} + \epsilon)n$ and a queue size of $O((r^2/\epsilon)^{r-1})$.

In [8], 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 will be given to the packet with the farther distance to travel. Using the techniques described in the previous subsection, it is not difficult to convert this algorithm into a deterministic algorithm with a running time of $2n+o(n)$; the queue size is constant, but slightly higher than that of the first algorithm.

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 our algorithm is between 10 and 15).

Theorem 3.3 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 Optimal Deterministic Sorting

In this section, we will apply the techniques described in Section 3 to a class of randomized sorting algorithms recently proposed by Kaklamanis and Krizanc [6]. As a result, 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 optimal randomized sorting algorithm proposed in [6]. In Subsection 4.2 we describe the modifications required to convert this randomized algorithm into a deterministic one. Subsection 4.3 contains the deterministic algorithm and a proof of the claimed bounds on time and queue size. Finally, Subsection 4.4 gives a few extensions.

4.1 An Optimal Randomized Algorithm

In this subsection we will give a high level description of a randomized algorithm with running time $2n+o(n)$ and constant queue size proposed by Kaklamanis and Krizanc [6]. Their algorithm is based on an earlier $2.5n+o(n)$ time algorithm of Kaklamanis, Krizanc, Narayanan, and Tsantilas [7]. 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 [6].

The following 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 will 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 will be 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 1 and $n/4$ in its row, inside its current subquadrant. Similarly, each column element selects a random location between 1 and $n/4$ 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 locations in the four subquadrants T_0 to T_3 corresponding 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 will arrive at the center points of the four quadrants after $0.5n + o(n)$ steps.

(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 will also be employed in our deterministic algorithm; a description of this scheme is given in the proof of Lemma 4.5. It can be shown that every element will reach 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 will receive 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 [6]. 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 [26] and attributed to Leighton.

- The routing in Step (5) of the algorithm is done according to a rather ingenious schedule described in [6]. 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 will 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.

4.2 Getting a Deterministic Algorithm

In this subsection we will 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, will yield a set of "good", that is, roughly evenly spaced, splitters. In this subsection, we will 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 3.2.

- 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 unshuffle operation described in Subsection 3.1.

- 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 our counter scheme.

As in the routing algorithm of Subsection 3.2, we will 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 will sort the row elements (column elements) in each block by their values, rather than by their destination blocks (which are not yet known during Step (4) of the algorithm), and then perform an $(\frac{n^{1-\beta}}{4})$-way unshuffle operation on the columns (rows) of each subquadrant. The effect of this operation is described in the following lemma, which is a simple generalization of Lemma 3.1.

Lemma 4.1 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 $\{1, \ldots, n^2\}$. 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 will 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 4.1 holds.

Lemma 4.2 Let A be any sequence of consecutive values in $\{1, \ldots, n^2\}$, 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 [3]. 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 properties hold for all i:

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

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

To achieve this, we will select 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 will contain $n^{2-\delta}$ 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 4.3 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 $\text{Rank}\,(s, S) = i$ we have

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

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 $|S| - \text{Rank}(s_1, S)$ elements s_2 with $s_1 < s_2$ in S; hence Rank $(s_1, X) < \text{Rank}(s_1, S) \cdot n^{\delta}$. If $s_2 \leq s_1$, then we have the following two cases:

(a) If s_2 is the largest element in S_j with $s_2 \leq s_1$, 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 $s_2 \leq s_1$, then all elements in $T(s_2)$ must be smaller than s_1.

Note that there are Rank (s_1, S) elements $s_2 \in S$ with $s_2 \leq s_1$, and at most $n^{2-2\delta}$ of these fall under case (a), including s_1 itself. Hence, at least $(\text{Rank}(s_1, S) - n^{2-2\delta}) \cdot n^{\delta}$ 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 4.1 Let D be the splitter set of size n^{δ} consisting of all $s \in S$ with Rank $(s, S) = i \cdot n^{2-2\delta} + 1$, for some nonnegative integer i. Then D is a set of "good" splitters, that is, it satisfies conditions (1) and (2) stated above.

Note that this sampling technique can guarantee good splitters because the sample set is sufficiently large, that is, contains $\omega(n)$ elements. On the other hand, 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.

4.3 Optimal Deterministic Sorting on Two-Dimensional Meshes

In the following description of the deterministic sorting algorithm, we will 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 will have to be substantially changed. The algorithm sorts with respect to the "column-major indexing nested inside a row-major indexing" defined in Section 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 4.4.

(3) Divide the n^2 elements into $n^2/2$ *row elements* and $n^2/2$ *column elements* as described in Subsection 4.2. 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 operation on the columns, as described in Subsection 3.1. 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 operation 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 locations in the four subquadrants T_0 to T_3 corresponding to the locations chosen in Step (4). This step will take time $1.25n$, but every copy will reach 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 4.1 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 will take 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 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 3.2 to distribute the elements evenly over the rows and columns of the destination blocks.

(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 will elaborate on this issue briefly in the proof of Claim 5 in Appendix A. We will establish our result in the following two lemmas.

Lemma 4.4 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 will be 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. □

Lemma 4.5 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 is given in the appendix. Together, Lemma 4.4 and Lemma 4.5 establish the following result.

Theorem 4.2 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 will still work 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 in one model may not be good at all in the other model.

4.4 Extensions

In [6], 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 4.3 There exists a deterministic algorithm for sorting on the three-dimensional mesh with running time $3.5n + o(n)$ and constant queue size.

Theorem 4.4 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 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 r-dimensional meshes and tori, $r \geq 4$, with running times of $(2r - 2.5)n + o(n)$ and $(r - 1)n + o(n)$, respectively. The best deterministic algorithms previously known for these networks required $(2r - 1)n$ steps on the mesh and $rn + o(n)$ on the torus [14].

5 Optimal Multi-Packet Routing and Sorting

The techniques presented in this paper can also be used to obtain optimal deterministic algorithms for k–k routing on r-dimensional meshes. In a k–k routing problem, a processor can initially hold up to k packets, and can receive up to k packets during the routing. For k–k routing, as well as for the related problem of k–k sorting, there exists a lower bound of $\frac{kn}{2}$ due to the bisection width of the network. Kaufmann, Rajasekaran, and Sibeyn [9] recently obtained randomized algorithms for k–k routing and sorting that match this lower bound, within a lower order additive term. Subsequently, Kunde [18] described a deterministic algorithm that achieves a similar bound.

Using the unshuffle operation and the counter scheme, we can design a deterministic algorithm for k–k routing that matches the running time of Kunde's algorithm. The algorithm can be seen as a deterministic variant of one of the randomized algorithms in [9], and shows an interesting relation between randomization and the unshuffle operation.

Consider the following uni-axial algorithm consisting of $2r$ phases. During each phase i, $1 \leq i \leq r$, we perform an unshuffle operation with respect to the ith dimension. This is done by locally sorting blocks of side length n^{β} and subsequently performing an $(n^{1-\beta})$-way unshuffle operation along each linear array in direction of the ith dimension. During each phase i, $r+1 \leq i \leq 2r$, we route the packets along dimension $i-r$ towards their destinations, using the counter scheme to distribute the packets evenly in their destination subcubes. Finally, we use local routing to bring all packets to their final destinations. Using the fact that a k–k relation can be routed in time $\frac{kn}{2} + o(kn)$ on a linear array, it is easy to see that the above algorithm routes any k–k routing problem in time $krn + o(krn)$ on an r-dimensional mesh of side length n. Hence, by running r such uniaxial algorithms simultaneously, we can obtain an algorithm that runs in time $kn + o(krn)$. Before running this algorithm, we have to partition the packets of the k–k relation into r similar subsets; this can be done in the same way as the partitioning of the packets of a 2–2 relation into 2 sets performed in the algorithm of Lemma 3.2.

Note that this is still a factor of 2 away from the lower bound. In [9], Kaufmann, Rajasekaran, and Sibeyn overcome this problem by showing that a k–k relation can be randomized on a linear array in time $\frac{kn}{4} + o(kn)$, with high probability. We can use a very similar idea to prove that the above algorithm runs in time $\frac{kn}{2} + o(krn)$. In the following, we say that an approximate k–k relation κ on n positions $0, \ldots, n-1$ is γ-normal if for any block B of $\gamma(n)$ consecutive positions, the destinations of the elements originating from B and the origins of the elements with destination in B are both evenly distributed over all blocks of size $\gamma(n)$. Using similar arguments as in [9], it can be shown that any γ-normal k–k relation can be routed on a linear array in time $\frac{kn}{4} + o(kn)$, for $\gamma = o(n)$. Not surprisingly, a randomization of a k–k relation is a γ-normal approximate k–k relation, with high probability. Also, both the unshuffle operations in phases 1 to r and the resulting routing problems in phases $r+1$ to $2r$ of our algorithm are γ-normal k–k relations, for some $\gamma = O(n^{1-\epsilon})$. This implies that the above algorithm runs in time $\frac{kn}{2} + o(krn)$.

Using the deterministic sampling technique described in Subsection 4.2, we can convert our k–k routing algorithm into an algorithm for the k–k sorting problem. If we schedule the computation and broadcasting of the splitter set in parallel with the first r phases of unshuffle operations, then we can obtain a deterministic algorithm for k–k sorting that matches the running time of our routing algorithm up to $o(rn)$ steps.

6 Improved Deterministic Algorithms for Selection

Using the sampling technique described in Subsection 4.2, we can also obtain improved deterministic algorithms for selection on meshes, tori, and meshes with diagonal edges. The algorithms are based on a number of randomized algorithms proposed by Kaklamanis, Krizanc, Narayanan, and Tsantilas [7], and by Narayanan [22]. For the two-dimensional mesh, we obtain an algorithm running in time $1.22n$. The best deterministic algorithm previously known required $1.44n$ steps [11]. The new algorithm can easily be adapted to the three-dimensional mesh, the torus, and the mesh with diagonal edges. In each case, the running time will match that of the best known randomized algorithm, given by $1.94n$, $1.13n$, and $0.65n$, respectively [22]. For meshes with diagonals, we can show a lower bound of $0.5n + \frac{n}{48}$.

In the following, we describe our improved deterministic algorithm for selection on the two-dimensional mesh. We will restrict our attention to the problem of selecting the median element at the center processor of the mesh. It was shown by Krizanc and Narayanan [10] that selection can be performed within the distance bound if the rank of the selected element is $o(n^2)$, or if we select at a processor with a distance of at least $n/2$ from the center.

Fortunately, the randomized selection algorithm of Kaklamanis, Krizanc, Narayanan, and Tsantilas is much simpler than the algorithms for sorting [6, 7], and the use of randomization is limited to only a few steps. Furthermore, a deterministic version of this selection algorithm was already described by Krizanc, Narayanan, and Raman [11]; due to a weaker deterministic sampling technique, their algorithm achieves only a running time of $1.44n$. We will be able to reuse most parts of their algorithm, and hence in the following we will focus on the differences between the two algorithms. The improved algorithm works in the following seven steps:

Algorithm SELECT:

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

(2) Route the sample elements into a block of side length $n^{1-\delta/2}$ at the center of the mesh. This can be done in n steps with the routing scheme employed in Step (2) of the sorting algorithm in Subsection 4.3.

(3) Concentrate all n^2 packets into a diamond of radius $\frac{\sqrt{6}-2}{4}n \approx 0.11n$ around the center of the mesh. As shown in [11], this operation can be completed in time $1.11n$.

(4) Sort the sample set in the center block using any standard sorting algorithm for the mesh, for example the algorithm of Schnorr and Shamir [29]. Then select the elements

s_1 and s_2 with $\text{Rank}(s_1, S) = \frac{1}{2}n^{2-\delta}$ and $\text{Rank}(s_2, S) = \frac{1}{2}n^{2-\delta} + n^{2-2\delta}$ as bracketing elements (i.e., upper and lower bounds) for the median. Using Lemma 4.3, it is easy to show that $\text{Rank}(s_2, X) - \text{Rank}(s_1, X) = O(n^{2-\delta})$, and that the median lies between s_1 and s_2. Sorting the center block and selecting s_1 and s_2 will take $o(n)$ time.

(5) Broadcast s_1 and s_2 in the entire middle diamond of radius $0.11n$. This takes time $0.11n$, and is thus completed at the same time as the concentration of the elements into the diamond of radius $0.11n$ described in Step (3).

(6) Every element between s_1 and s_2 routes itself towards the center. At the same time, the exact ranks of s_1 and s_2 in X are computed by a prefix computation that counts those elements that are smaller than s_1, and those that are larger than s_2. This takes another $0.11n$ time steps, after which both the elements between s_1 and s_2 and the global ranks of s_1 and s_2 are located in a block of size $o(n)$ at the center of the mesh.

(7) Choose the median from among the elements that were routed to the center in the previous step. This can be done by sorting these elements, and takes time $o(n)$.

The total running time of the above algorithm is approximately $1.22n$. Ignoring lower order terms, this time consists of n steps to route the sample to the center, $0.11n$ steps to broadcast the bracketing elements in the middle diamond, plus another $0.11n$ steps to collect the results of the broadcast in the center. Hence, an obvious strategy for improving the running time would be to try to concentrate the packets into a middle diamond of radius smaller than $0.11n$. However, Krizanc, Narayanan, and Raman [11] have shown that concentrating n^2 packets into a smaller diamond would actually increase the running time, since, due to the limited number of edges on the perimeter of the diamond, the concentration could not be completed in time for the broadcast in Step (5).

The main difference between this and the previously best deterministic algorithm is in the technique used to select the bracketing elements s_1 and s_2. The $1.44n$ step algorithm selects a sample of size $o(n)$; this means that an additional broadcast of all sample elements into the middle diamond is needed to determine the ranks of the sample elements. Only after this is done, can the bracketing elements be selected from the sample.

For the mesh with diagonals, we can obtain a lower bound of $0.5n + \frac{n}{48}$. This lower bound is based on the observation that in a mesh with diagonals, a large number of elements initially have a distance close to the radius from the center point. If all of these elements have a rank close to the median, then there will not be enough bandwidth available to route all median candidates towards the center. We are not aware of any general lower bounds for selection on other mesh-related networks.

Very recently, Condon and Narayanan [4] have given an improved randomized algorithm for selection that runs in $1.19n$ steps. Using the techniques described in this paper, we can convert their algorithm into a deterministic algorithm with the same running time. The construction is slightly more complicated than in the case of the $1.22n$ time algorithm, and uses both the deterministic sampling technique and the unshuffle operation. Condon and Narayanan also show a number of lower bounds for selection which hold for various restricted models of the mesh. However, even for these fairly restricted models, a large gap remains between the best upper and lower bounds.

7 Summary and Open Problems

In this paper, we have introduced a new technique that allows us to "derandomize" many of the randomized algorithms for routing and sorting on meshes that have been proposed in recent years. By applying this technique, we have obtained optimal or improved deterministic algorithms for a number of routing and sorting problems on meshes and related networks. The new technique is 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 routing and sorting on meshes and related networks whose running time cannot be matched, within a lower order additive term, by a corresponding deterministic algorithm.

This naturally raises the question whether randomization is of any help at all in the design of routing and sorting algorithms for these types of networks. In this context, we point out that 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 within blocks. Also, the results in this paper would not have been possible without the extensive study of randomized schemes for routing and sorting by a number of other authors, which has resulted in a variety of fast randomized algorithms [6, 7, 8, 9, 26].

It is an interesting open question whether our "derandomization" technique can be used to obtain improved deterministic algorithms for other classes of networks, and perhaps even other types of problems. It seems that our techniques are most suitable for networks with large diameter, since we repeatedly sort fairly large subsets of the input. A straightforward application of our technique 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.

In the case of our optimal algorithms for routing and sorting, any further reduction of the queue size would be an interesting improvement. Another possible direction for future research is to try to design algorithms with a simpler control structure than those presented in this paper. In the case of sorting, it is an interesting open question whether there exists an optimal algorithm that does not make any copies of elements, or whether a general lower bound can be shown for this case.

An important open question that remains unsettled is whether there exist optimal algorithms for routing and sorting in r-dimensional meshes, $r > 3$. For large r, the best algorithms currently known are still nearly a factor of 2 away from the diameter lower bound. Finally, it is a challenging open problem to determine the complexity, within a lower order additive term, of the problem of selection on the standard mesh.

Acknowledgements: I would like to thank Christos Kaklamanis, Greg Plaxton, and Rajmohan Rajaraman for helpful discussions.

References

[1] K. E. Batcher. Sorting networks and their applications. In *Proceedings of the AFIPS Spring Joint Computer Conference,* vol. 32, pages 307–314, 1968.

[2] S. Cheung and F. C. M. Lau. Mesh permutation routing with locality. *Information Processing Letters*, 43:101–105, 1992.

[3] R. Cole and C. K. Yap. A parallel median algorithm. *Information Processing Letters*, 20:137–139, 1985.

[4] A. Condon and L. Narayanan. Upper and lower bounds for selection on the mesh. Unpublished manuscript, 1993.

[5] 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.

[6] 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.

[7] 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.

[8] 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.

[9] 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.

[10] D. Krizanc and L. Narayanan. Optimal algorithms for selection on a mesh-connected processor array. In *Fourth Annual IEEE Symposium on Parallel and Distributed Processing*, December 1992.

[11] D. Krizanc, L. Narayanan, and R. Raman. Fast deterministic selection on mesh-connected processor arrays. In *11th Conference on Foundations of Software Technology and Theoretical Computer Science*, pages 336–346, December 1991.

[12] 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.

[13] M. Kunde. Bounds for 1-selection and related problems on grids of processors. In *Fourth International Workshop on Parallel Processing by Cellular Automata and Arrays (PARCELLA)*, pages 298–307. Springer, 1988.

[14] 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.

[15] 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.

[16] 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.

[17] M. Kunde. Concentrated regular data streams on grids: Sorting and routing near to the bisection bound. In *Proceedings of the 32st Annual IEEE Symposium on Foundations of Computer Science*, pages 141–150, October 1991.

[18] M. Kunde. Block gossiping on grids and tori: Deterministic sorting and routing match the bisection bound. In *First Annual European Symposium on Algorithms*, September 1993. To appear.

[19] 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.

[20] F. T. Leighton. *Introduction to Parallel Algorithms and Architectures: Arrays, Trees and Hypercubes*. Morgan-Kaufmann, San Mateo, CA, 1991.

[21] 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.

[22] L. Narayanan. *Selection, Sorting, and Routing on Mesh-Connected Processor Arrays.* PhD thesis, Department of Computer Science, University of Rochester, Rochester, NY, May 1992.

[23] D. Nassimi and S. Sahni. Bitonic sort on a mesh-connected parallel computer. *IEEE Transactions on Computers*, C–28:2–7, 1979.

[24] S. E. Orcutt. *Computer Organization and Algorithms for Very-High Speed Computations*. PhD thesis, Department of Computer Science, Stanford University, September 1974.

[25] S. Rajasekaran and R. Overholt. Constant queue routing on a mesh. *Journal of Parallel and Distributed Computing*, 15:160–166, 1992.

[26] S. Rajasekaran and T. Tsantilas. Optimal routing algorithms for mesh-connected processor arrays. *Algorithmica*, 8:21–38, 1992.

[27] K. Sado and Y. Igarashi. Some parallel sorts on a mesh-connected processor array. *Journal of Parallel and Distributed Computing*, 3:389–410, 1986.

[28] I. D. Scherson and S. Sen. Parallel sorting in two-dimensional VLSI models of computation. *IEEE Transactions on Computers*, 38:238–249, 1989.

[29] 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.

[30] C. D. Thompson and H. T. Kung. Sorting on a mesh-connected parallel computer. *CACM*, 20:263–271, 1977.

[31] 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.

A Proof of Lemma 4.5

Lemma 4.5 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 will restrict our attention to the lower right quadrant of the mesh.

As stated in the algorithm, we will 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 will be given to the element with the farthest distance to travel. In the following, we will 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 will only be concerned 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 will never be delayed until it reaches its destination.

Assuming the above routing scheme, we establish Lemma 4.5 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 [6], 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 4.2, 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 4.1 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 will differ 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 will differ 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 will have $\frac{n}{2} \pm O(n^{3-\beta-2\alpha})$ row elements. $\square$

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 3.2. 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 will guarantee 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 will receive at most one row element. This proves that the routing step achieves a constant queue size. $\square$

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 will be needed for the broadcast of the exact splitter ranks in Step (10) of the algorithm. Thus, the total queue size is bounded by 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 will receive $n^{2\alpha} \pm O(n^{2-\delta})$ elements. By Lemma 4.2, approximately half of these elements will be 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 will arrive 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 will deliver all packets within time $n/2 - 1$, even if processors may initially hold an arbitrary number of packets (see, for example, [20, 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 $n/2 - r$, where r 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 will suffice 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 will 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 will give 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 will reach 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 will 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 will 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 will have 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 will 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 will arrive at this processor $n/4-1$ steps after it was broadcast from the center. Now the elements will 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 will leave the topmost processor and move towards its destination row. Once an element has started moving, it will not be delayed until it reaches its destination row. Between time $n/2$ and $3n/4$, only one row element will leave the topmost processor in any two consecutive steps, and from time $3n/4$ to the end of the routing, three elements will leave the topmost processor in any four consecutive steps. As before, an element will move 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 will be $\sim n/4$ such elements in the column. Since these elements have a higher priority than the rest, all these elements will 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 will 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 will not be 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. $\square$