Randomized Routing and Sorting on Fixed-Connection Networks

F. T. Leighton[1,2]
Bruce M. Maggs[1]
Abhiram G. Ranade[3]
Satish B. Rao[1,4]

Abstract

This paper presents a general paradigm for the design of packet routing algorithms for fixed-connection networks. Its basis is a randomized on-line algorithm for scheduling any set of N packets whose paths have congestion c on any bounded-degree leveled network with depth L in $O(c+L+\log N)$ steps, using constant-size queues. In this paradigm, the design of a routing algorithm is broken into three parts: (1) showing that the underlying network can emulate a leveled network, (2) designing a path selection strategy for the leveled network, and (3) applying the scheduling algorithm. This strategy yields randomized algorithms for routing and sorting in time proportional to the diameter for meshes, butterflies, shuffle-exchange graphs, multidimensional arrays, and hypercubes. It also leads to the construction of an *area-universal network*: an N-node network with area $\Theta(N)$ that can simulate any other network of area $O(N)$ with slowdown $O(\log N)$.

This research was supported by the Defense Advanced Research Projects Agency under Contracts N00014–87–K–825 and N00014–89–J–1988, the Office of Naval Research under Contracts N00014–86–K–0593 and N00014–86–K–0564, the Air Force under Contract OSR–89–0271, and the Army under Contract DAAL–03–86–K-0171. Tom Leighton is supported by an NSF Presidential Young Investigator Award with a matching funds provided by IBM.

[1]Laboratory for Computer Science, MIT, Cambridge, MA.

[2]Department of Mathematics, MIT, Cambridge, MA.

[3]Department of Electrical Engineering and Computer Science, University of California, Berkeley, CA.

[4]Aiken Computation Laboratory, Harvard University, Cambridge, MA.

Second and third authors' current address: NEC Research Institute, Princeton NJ.

1 Introduction

The task of designing an efficient packet routing algorithm is central to the design of most large-scale general-purpose parallel computers. In fact, even the basic unit of time in some parallel machines is measured in terms of how fast the packet router operates. For example, the speed of an algorithm in the Connection Machine CM-2 is often measured in terms of *routing cycles* (roughly the time to route a random permutation) or *petit cycles* (the time to perform an atomic step of the routing algorithm). Similarly, the performance of a machine like the BBN Butterfly [5] is substantially influenced by the speed and rate of successful delivery of its router.

Packet routing also provides an important bridge between theoretical computer science and applied computer science; it is through packet routing that a real machine such as the Connection Machine is able to simulate an idealized machine such as the CRCW PRAM. More generally, getting the right data to the right place at the right time is an important, interesting, and challenging problem. Not surprisingly, it has also been the subject of a great deal of research.

1.1 Past work on routing

In 1965 Beneš [6] showed that the inputs and outputs of an N-node Beneš network (two back-to-back butterfly networks) can be connected in any permutation by a set of disjoint paths. Shortly thereafter Waksman [40] devised a simple sequential algorithm for finding the paths in $O(N)$ time. Given the paths, it is straightforward to route a set of packets from the inputs to the outputs an N-node Beneš network in any one-to-one fashion in $O(\log N)$ steps using queues of size 1. A one-to-one routing problem like this is also called a *permutation routing problem.* Although the inputs comprise only $O(N/\log N)$ nodes in an N-node Beneš network, it is possible to route any permutation of N packets in $O(\log N)$ steps by pipelining $\Theta(\log N)$ such permutations. Unfortunately, no efficient parallel algorithm for finding the paths is known.

In 1968 Batcher [4] devised an elegant and practical parallel algorithm for sorting N packets on an N-node shuffle-exchange network in $\log^2 N$ steps[1] using queues of size 1. The algorithm can be used to route any permutation of packets by sorting based on destination address. The result extends to routing many-one problems provided that (as is typically assumed) two

[1]Throughout this paper $\log N$ denotes $\log_2 N$ and $\log^2 N$ denotes $(\log N)^2$.

packets with the same destination can be *combined* to form a single packet should they meet en route to their destination.

No better deterministic algorithm was found until 1983, when Ajtai, Komlós, and Szemerédi [1] solved a classic open problem by constructing an $O(\log N)$-depth sorting network. Leighton [16] then used this $O(N \log N)$-node network to construct a degree 3 N-node network capable of solving any N-packet routing problem in $O(\log N)$ steps using queues of size 1. Although this result is optimal up to constant factors, the constant factors are quite large and the algorithm is of no practical use. Hence, the effort to find fast deterministic algorithms has continued. Recently Upfal discovered an $O(\log N)$-step algorithm for routing on an expander-based network called the multibutterfly [37]. The algorithm solves the routing problem directly without reducing it to sorting, and the constant factors are much smaller than those of the AKS-based algorithms. In [18], Leighton and Maggs show that the multibutterfly is fault tolerant and improve the constant factors in Upfal's algorithm.

There has also been great success in the development of efficient randomized packet routing algorithms. The study of randomized algorithms was pioneered by Valiant [38] who showed how to route any permutation of N packets in $O(\log N)$ steps on an N-node hypercube with queues of size $O(\log N)$ at each node. Valiant's idea was to route each packet to a randomly-chosen intermediate destination before routing it to its true destination. Although the algorithm is not guaranteed to deliver all of the packets within $O(\log N)$ steps, for any permutation it does so with high probability. In particular, the probability that the algorithm fails to deliver the packets within $O(\log N)$ steps is at most $1/N^k$, for any fixed constant k. (The value of k can be made arbitrarily large by increasing the constant in the $O(\log N)$ bound.) Throughout this paper, we shall use the phrase *with high probability* to mean with probability at least $1 - 1/N^k$ for any fixed constant k, where N is the number of packets.

Valiant's result was improved in a succession of papers by Aleliunas [2], Upfal [36], Pippenger [26], and Ranade [29]. Aleliunas and Upfal developed the notion of a *delay path* and showed how to route on the shuffle-exchange and butterfly networks (respectively) in $O(\log N)$ steps with queues of size $O(\log N)$. Pippenger was the first to eliminate the need for large queues, and showed how to route on a variant of the butterfly in $O(\log N)$ steps with queues of size $O(1)$. Ranade showed how combining could be used to extend the Pippenger result to include many-one routing problems, and tremendously simplified the analysis required to prove such a result. As a consequence, it has finally become possible to simulate a step of an N-processor CRCW PRAM on an N-node butterfly or hypercube in $O(\log N)$ steps using constant-size queues on each edge.

Concurrent with the development of these hypercube-related packet routing algorithms has been the development of algorithms for routing in meshes. The randomized algorithm of Valiant and Brebner can be used to route any permutation of N packets on a $\sqrt{N} \times \sqrt{N}$ mesh in $O(\sqrt{N})$ steps using queues of size $O(\log N)$. Kunde [14] showed how to route deterministically in $(2+\varepsilon)\sqrt{N}$ steps using queues of size $O(1/\varepsilon)$. Also, Krizanc, Rajasekaran, and Tsantilis [13] showed how to randomly route any permutation in $2\sqrt{N} + O(\log N)$ steps using constant-size queues. Most recently, Leighton, Makedon, and Tollis discovered a deterministic algorithm for routing any permutation in $2\sqrt{N} - 2$ steps using constant-size queues [19], thus achieving the optimal time bound in the worst case.

1.2 Our approach to routing

One deficiency with the state-of-the-art in packet routing is that aside from Valiant's paradigm of first routing to random destinations, all of the algorithms and their analyses are very specifically tied to the network on which the routing is to take place. For example, the way that the queue size is kept constant in the butterfly routing algorithms is quite different from the way that it is kept constant in the mesh routing algorithms. Moreover, the butterfly and hypercube algorithms are so specific to those networks that no $O(\log N)$-step constant-queue-size algorithm was previously known for the closely related shuffle-exchange network. The lack of a good routing algorithm for the shuffle-exchange network is one of the reasons that the butterfly is preferred to the shuffle-exchange network in practice.

Our approach to the problem differs from previous approaches in that we separate the process of selecting paths for the packets from the process of timing the movements of the packets along their paths. More precisely, we break a routing problem into two stages. In Stage 1, we select a path for each packet from its origin to its destination. In Stage 2, we schedule the movements of the packets along their paths. The focus of this paper is on Stage 2. The goal of Stage 2 is to find a schedule that minimizes both the time for the packets to reach their destinations and the number of packets that are queued at any node. The schedule must satsify the constraint that at each time step each edge in the network can transmit at most one packet.

Of course, there must be some correlation between the performance of the scheduling algorithm and the selection of the paths. In particular, the maximum distance d traveled by any packet is always a lower bound on the time required to route all packets. We call this distance the *dilation* of the set of paths. Similarly, the largest number of packets that must traverse a single edge during the entire course of the routing is a lower bound. We call this number the *congestion* c of the paths. In terms of these parameters, the goal of Stage 1 is to select paths for the packets that minimize c and d.

For many networks, Stage 1 is easy. We simply use Valiant's paradigm of first routing each message to a random destination. It is easily shown for meshes, butterflies, shuffle-exchange networks, etc., that this approach yields values of c and d that are within a small constant factor of the diameter of the network. Moreover, this technique also usually works for many-one problems provided that the address space is randomly hashed.

Stage 2 has traditionally been the hard part of routing. Curiously, however, we have found that by ignoring the underlying network and the method of path selection, Stage 2 actually becomes easier to solve! In [20] for example, Leighton, Maggs, and Rao show that for any set of packets whose paths have congestion c and dilation d, in any network, there is a schedule of length $O(c + d)$ in which at most one packet traverses each edge at each time step, and in which the maximum queue size required is $O(1)$. In this paper, we show that there is an efficient randomized parallel scheduling algorithm for the entire class of bounded-degree *leveled* networks. In a leveled network, each edge is directed and connects a level i node to a level $i + 1$ node, where the level numbers range from 0 to L. We call L the *depth* of the network. The algorithm produces a schedule of length $O(c + L + \log N)$ with high probability, and uses constant-size queues.

By applying the approach just described, we can design fast routing algorithms for the most common fixed-connection networks. The first step is to convert the network at hand into a leveled network. In particular, we create a virtual leveled network that can be efficiently simulated by the existing network, and we figure out how to move packets between the two networks (i.e., we reduce the problem of routing on the given network to the problem of routing on a very similar leveled network). Next, we select paths for the packets so as to minimize the congestion c in the leveled network. Because the network is leveled, the dilation is automatically at most L, which in all of our algorithms is at most a constant factor larger than the diameter of the underlying network. The path selection strategy typically uses some combination of greedy paths and random intermediate destinations. We then conclude by applying the $O(c + L + \log N)$-step scheduling algorithm.

1.3 The application of routing to sorting

Packet routing and sorting have long been known to be closely linked problems on fixed-connection networks. In his fundamental paper, Batcher [4] showed that an algorithm for sorting on a network can usually be converted into an algorithm for packet routing. Reif and Valiant [32], on the other hand, described a method for converting a routing algorithm into a randomized sorting algorithm. As a consequence, they derived randomized sorting algorithms for hypercubes and butterflies that run in $O(\log N)$ steps and use $O(\log N)$-size queues.

In this paper we combine the Reif-Valiant approach with our routing strategy to devise improved algorithms for sorting on fixed-connection networks. For each network considered, the algorithm runs in time proportional to the diameter of the network, and uses constant-size queues. Such algorithms were previously known only for bounded-dimensional arrays [16, 35].

1.4 Outline of the results

The basis of most of the results in this paper is a proof that a variant of Ranade's algorithm can be used to schedule any set of N packets whose paths have congestion c on a bounded-degree leveled network with depth L in $O(c + L + \log N)$ steps using constant-size queues. The algorithm is randomized, but requires only $\Theta(\log^2 N)$ bits of randomness to succeed with high probability. The proof of this result is included in Section 2. Curiously, the proof is simpler than the previous proof of the same result applied specifically to routing random paths in butterflies [29], and allows for improved constant factors.

In Sections 3 through 10 we examine the many applications of the $O(c+L+\log N)$-step scheduling algorithm for leveled networks. These applications include routing algorithms for meshes, butterflies, shuffle-exchange networks, multidimensional arrays and hypercubes, and fat-trees. Section 3 presents the simplest application: routing N packets in $O(\sqrt{N})$ steps on a $\sqrt{N} \times \sqrt{N}$ mesh. Another simple application, described in Section 4, is an algorithm for routing N packets in $O(\log N)$ steps on an N-node butterfly. It is not obvious that the scheduling algorithm can be applied to the shuffle-exchange network because it is not leveled. Nevertheless, in Section 5 we show how to route N-packets in $O(\log N)$ steps on an N-node shuffle-exchange network by identifying a leveled structure in a large portion of the network. In Section 6 we present an algorithm for routing kN packets on an

N-node k-dimensional array with maximum side length M in $O(kM)$ steps. In Section 7, we show how to adapt the scheduling algorithm to route a set of messages with load factor λ in $O(\lambda + \log M)$ steps on a fat-tree [21] with root capacity M.

The fat-tree routing algorithm leads to the construction of an *area-universal network*: an N-node network with area $\Theta(N)$ that can simulate any other network of area $O(N)$ with slowdown $O(\log N)$. An analogous result is shown for a class of *volume-universal* networks.

Our sorting results are included in Sections 8 through 10. In particular, we describe an $O(\log N)$-step algorithm for sorting on an N-node butterfly or hypercube in Section 8, an $O(\log N)$-step algorithm for sorting on a shuffle-exchange network in Section 9, and an $O(kM)$-step algorithm for sorting kM^k items on a k-dimensional array with side length M in Section 10.

1.5 Some comments on the model

All of our algorithms are presented in the *packet model* of computation. In this model time is partitioned into synchronous steps. At each time step, one packet can be transmitted across each edge of the network. The packet model is the natural abstraction for store and forward routing algorithms used on machines such as the NCube, NASA MPP, Intel Hypercube, and Transputer-based machines. It is also robust in the sense that it allows combining, it corresponds nicely to the various PRAM models, and it does not make assumptions about packet lengths. Consequently, it is the most studied model in the literature.

Other models of interest are the circuit switching model [12] and the cut-through or wormhole model [9]. These models arise in practice and are also of theoretical interest, although less is known about them. Although our results have some limited applications in these models, we will primarily concern ourselves with the packet model in this paper.

2 An $O(c+L+\log N)$ scheduling algorithm for leveled networks

In this section we present a randomized algorithm for scheduling the movements of a set of N packets in a leveled network with depth L. By assumption, the paths taken by the packets are given and have congestion c. With high probability, the algorithm delivers all of the packets to their destinations in $O(c + L + \log N)$ steps. The algorithm is *on-line* in the sense that the schedule is produced as the packets are routed through the network. (Note: The number of nodes in the network does not appear in the time to deliver the packets or in the probability of success. There may be more than N or fewer.)

2.1 Leveled networks

In a *leveled* network with *depth* L, the nodes can be arranged in $L+1$ levels numbered 0 through L, such that every edge in the network leads from some node on level i to a node on level $i+1$, for $0 \leq i < L$. The nodes in the network represent processors and the edges represent unidirectional communication links. The processors are assumed to contain some switching hardware for sending and receiving packets. We will assume that each node has in-degree and out-degree at most Δ, where Δ is a fixed constant.

There are 3 kinds of queues in the network. Each node has an *initial* queue in which packets reside before execution begins, and a *final* queue into which the packets destined for the node must be delivered. At the head of each edge is an *edge* queue for buffering packets in transit. We place no restriction on the size of the initial and final queues. The edge queues, however, can each hold at most q packets, where q is a fixed constant. In this paper we shall assume that $q \geq 2$. With minor modifications, however, the algorithm and the analysis can be adapted for the case $q = 1$. At the start of the execution, all of the N packets reside in initial queues. A packet can originate on any level and can have its destination on any higher-numbered level.

2.2 The algorithm

The scheduling algorithm is similar to the one in [29] except that instead of ordering the packets based on destination address, we order them according to randomly-chosen ranks. In particular, each packet is assigned an integer rank chosen randomly, independently, and uniformly from the range $[1, R]$, where R will be specified later. The ranks are used by the algorithm to determine the order in which packets move through each node. The algorithm maintains two important invariants. First, throughout the execution of the algorithm, the packets in each edge queue are arranged from head to tail in order of increasing rank. Second, a packet is routed through a node only after all the other packets with lower ranks that must pass through the node have done so. Special *ghost* packets are used to help the algorithm maintain these invariants.

The algorithm begins with an initialization phase in which the packets in each initial queue are sorted according to their ranks. Ties in rank are broken according to destination address. At the tail of each initial queue a special end-of-stream (EOS) packet is inserted, and is assigned rank $R + 1$.

After initialization, the algorithm operates as follows. At each step, a node examines the head of its initial queue and the heads of any edge queues into the node. If any of these queues are empty, then the node does nothing. Otherwise, it selects the packet with the smallest rank as a candidate to be transmitted. Ties are again broken using the destination address. The selected packet is sent forward only if the queue at the head of the next edge on its path contained fewer than q packets at the beginning of the step. (We assume that the nodes at both the head and tail of an edge can determine how many packets are stored in the edge's queue in constant time.) Thus, an edge queue is guaranteed never to hold more than q packets.

To prevent queues from becoming empty, whenever a node selects a packet for transmission, it sends a *ghost packet* with the same rank on each of the other edges out of the node, provided that their edge queues contained fewer than q packets at the beginning of the step. Because the node sends packets out in order of strictly increasing rank, the rank of the ghost packet provides the receiving node with a lower bound on the ranks of the packets that it will receive on the same edge in the future.

Like other packets, a ghost packet can be selected for transmission if it is at the head of its queue and has a smaller rank than the ranks of packets in all of the other queues. A ghost never remains at a node for more than one step, however. At the end of each step a node destroys any ghosts that were present in its edge queues at the beginning of the step.

End-of-stream (EOS) packets are also given special treatment. Since an EOS packet has rank $R + 1$, it cannot be selected by a node unless there is an EOS packet at the head of the node's initial queue and at the head of each of the queues on all of the node's incoming edges. Once an EOS packet has been selected, the node will create a new EOS packet for each of its outgoing edges and for each edge will attempt to send the corresponding packet at each step until it succeeds. After sending an EOS packet along an edge, a node will not send any more packets along that edge.

Figure 1 shows an example in which a ghost packet expedites the delivery of another packet. For simplicity, initial and final queues are not shown. The next edge on the path for the packet with rank 35 is the upper edge out of

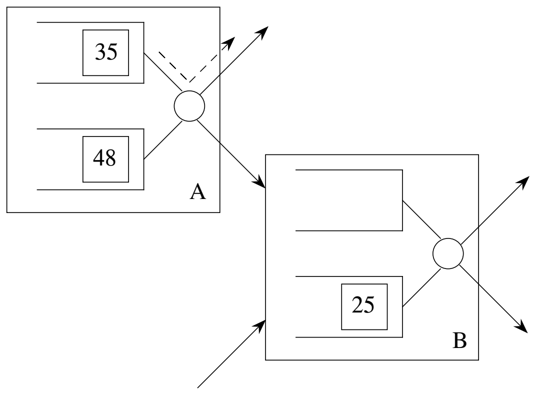

Figure 1: A ghost with rank 35 expedites the delivery of a packet with rank 25.

node A. By sending a ghost with rank 35 on the lower edge, node A informs node B that subsequent packets will not have rank smaller than 35. Node B can then transmit the packet with rank 25 on the next step. Without the ghost packet, the transmission of the packet with rank 25 would be delayed until a packet actually arrived at the top queue of node B.

In the manner of [29], we summarize the properties of the routing algorithm in the following lemmas.

Lemma 2.1 *After the initialization phase, each queue in the network holds packets sorted from head to tail in order of increasing rank. Each node sends out packets in order of increasing rank.*

Proof: The proof is by induction on the number of steps executed by the algorithm. □

Lemma 2.2 *For each node on level i, there is a $t \leq i$ such that at the beginning of time step t, the initial queue and each of the queues on the edges into the node holds a packet of some type. After step t the node sends out a packet on each outgoing edge at every step (unless the corresponding edge queue does not have space) until it transmits an EOS packet on that edge.*

Proof: The proof is by induction on the number of steps executed by the algorithm. $\square$

In the following lemma we denote the rank of a packet p by $\mathrm{rank}(p)$, and the level of a node s by $\mathrm{level}(s)$.

Lemma 2.3 *Suppose a packet p waits at a node s at time t. Then one of the following is true.*

1. *At time t another packet p' with $\mathrm{rank}(p') \leq \mathrm{rank}(p)$ is selected for transmission by s.*
2. *At time t, p is selected for transmission from s to an adjacent node s' at the next level, but the queue on the edge into s' is filled with packets $p'_1, \ldots, p'_q$ with $\mathrm{rank}(p'_q) \leq \ldots \leq \mathrm{rank}(p'_1) \leq \mathrm{rank}(p)$.*
3. *At time t some queue on an edge into s is empty and $t - \mathrm{level}(s) \leq 0$.*

In the first case, we say that p' m-delay*s p in switch s at time t, and in the second, p'_1 through p'_q f-delay p in switch s' at time t.*

Proof: Straightforward.

It is useful to define the *lag* of a packet p at node s at time t as $\mathrm{lag}(p, t) = t - \mathrm{level}(s)$. The lag gives a lower bound on the amount of time the packet has waited in queues before step t.

2.3 Analysis

Our analysis of the algorithm uses a delay sequence argument similar to the ones in [2], [29], and [36]. Each delay sequence corresponds to an event in the probability space. We first show that if some packet is delayed, then a delay sequence occurs. Then by counting all possible delay sequences, we show that it is unlikely that any delay sequence occurs with delay greater than $O(c + L + \log N)$.

Definition 2.4 *An $(\mathcal{L}, \mathcal{W}, \mathcal{R})$-delay sequence consists of 4 components:*

1. *a path through the network of length $\mathcal{L}$ beginning at a node on some packet's path. The path, called the delay path, can traverse the edges of the network in both directions.*
2. *a sequence $s_1, \ldots, s_{\mathcal{W}}$ of not necessarily distinct nodes in the network such $s_1, \ldots, s_{\mathcal{W}}$ appear in order along the delay path.*

3. *a sequence $p_1, \ldots, p_{\mathcal{W}}$ of distinct packets, such that for $1 \leq i \leq \mathcal{W}$, the path for packet p_i passes through switch s_i.*

4. *a non-increasing sequence $r_1, \ldots, r_{\mathcal{W}}$ of ranks such that $r_1 - r_{\mathcal{W}} \leq \mathcal{R}$*

The only use of randomness in the algorithm is in the choice of ranks for the packets. Thus, the probability space consists of R^N equally likely elementary outcomes, one for each possible setting of the ranks. Each delay sequence corresponds to the event in the probability space in which the rank chosen for packet p_i is r_i, for $1 \leq i \leq \mathcal{W}$. Each such event consists of $\mathcal{R}^{N-\mathcal{W}}$ elementary outcomes and occurs with probability $1/\mathcal{R}^{\mathcal{W}}$. We call these events *bad* events. We say that a delay sequence occurs whenever the corresponding bad event occurs. The following lemma shows that whenever the routing takes too long, some delay sequence occurs.

Lemma 2.5 *For any w, if some packet is not delivered by step $L + w$ then a bad event corresponding to a $(L + \frac{2w}{q}, w, R)$-delay sequence has occurred.*

The informal idea behind the proof is that whenever routing takes too long, we can identify a sequence of packets $p_1, \ldots, p_v$, $v \geq w$, that are in some sense responsible. We will show that the first w elements of this sequence, i.e., $p_1, \ldots, p_w$ are the packets on an $(L + 2w/q, w, R)$ delay sequence that has occurred.

We first present an incremental construction to identify the packets $p_1, p_2, \ldots, p_v$. We will use auxiliary sequences $p'_1, \ldots, p'_v$ and $t_0, t_1, \ldots, t_{v-1}$ to facilitate the discussion. The sequence starts with $p_1 = p'_1$ being the last packet delivered and $t_0 > L + w$ being the step at which p'_1 reached its destination.

In general, given p'_i and t_{i-1}, we show how the sequence can be extended. If p'_i is not a ghost, then we set $p_i = p'_i$. If p'_i is a ghost, then we follow p'_i back in time until reaching the node in which p'_i was created from some p_i. In either case we next follow p_i back until time t_i when it was forced to wait in some node s_i. The next packet in the sequence is identified by using Lemma 2.3. If p_i was m-delayed by p' in s_i, we set $p'_{i+1} = p'$. Suppose p_i was f-delayed by $\bar{p}_1, \bar{p}_2, \ldots, \bar{p}_q$ in s', where $\bar{p}_1$ has the largest rank and $\bar{p}_q$ has the smallest. Then we set $p_{i+j} = p'_{i+j} = \bar{p}_j$, $s_{i+j} = s'$ and $t_{i+j} = t_i$ for $1 \leq j \leq q - 1$, and $p'_{i+q} = \bar{p}_q$. If some queue in s_i was empty at t_i, or if $t_i = 0$ we terminate the construction.

The incremental construction extends each sequence by 1 element, or by q elements, depending upon whether there was a m-delay or a f-delay.

We apply the construction until a total of w/q f-delays are encountered, or the construction terminates. Let j denote the number of incremental steps used, of which $f \leq w/q$ involve f-delays, and the remaining $j - f$ involve m delays.

The key observation is that each time a new packet is added to the delay sequence, the lag of the packet being followed back in time is reduced by either one or two.

Lemma 2.6 *Consider an incremental step that starts with p'_i at time t_{i-1}.*

1. *Suppose p_i was m-delayed. Then*

$$\mathrm{lag}(p'_i, t_{i-1}) = \mathrm{lag}(p_i, t_i) + 1 = \mathrm{lag}(p'_{i+1}, t_i) + 1$$

2. *Suppose p_i was f-delayed. Then*

$$\mathrm{lag}(p'_i, t_{i-1}) = \mathrm{lag}(p_i, t_i) + 1 = \mathrm{lag}(p'_{i+q}, t_i) + 2$$

Proof: Since there is no waiting between t_{i-1} and t_i+1, we get $\mathrm{lag}(p'_i, t_{i-1}) = \mathrm{lag}(p_i, t_i+1)$. But since p_i waits at t_i, we have $\mathrm{lag}(p_i, t_i)+1 = \mathrm{lag}(p_i, t_i+1) = \mathrm{lag}(p'_i, t_{i-1})$. For m-delays, we know that p_i and p'_{i+1} are in the same node at t_i, and hence must have identical lags. For f-delays, we get $\mathrm{lag}(p_i, t_i) = \mathrm{lag}(p'_{i+q}, t_i) + 1$, since p'_{i+q} is on the next level. $\square$

Lemma 2.7 *The length v of the sequence $p_1, \ldots, p_v$ is at least w.*

Proof: Suppose $f = w/q$. We know that each f-delay adds q elements to the sequence, and thus $v \geq q(w/q) = w$. Otherwise, we have $f < w/q$, and we know that the construction was terminated because at the last step there was neither an f-delay nor an m-delay, but some queue was found empty, or $t_v = 0$. We know that $\mathrm{lag}(p'_1, t_0) \geq w + 1$, and by Lemma 2.3, $\mathrm{lag}(p'_v, t_{v-1}) = \mathrm{lag}(p_v, t_v) + 1 \leq 1$. Thus, $\mathrm{lag}(p', t_0) - \mathrm{lag}(p', t_{v-1}) \geq w$. By applying Lemma 2.6 j times we get $\mathrm{lag}(p'_1, t_0) - \mathrm{lag}(p'_v, t_{v-1}) = j - f + 2f = j + f$. Thus $j + f \geq w$. But $v \geq j + f(q-1) \geq j + f \geq w$. $\square$

Lemma 2.8 *Consider the path starting from s_1 passing through $s_2, \ldots, s_v$ in that order such that the segment between s_{i-1} and s_i consists of the path of p'_i. The total length of the path is at most $L + 2w/q$.*

Proof: The path has f forward edges. Since it goes back at most L levels, its total length is at most $L + 2f \le L + 2w/q$. $\square$

We now prove Lemma 2.5.

Proof of Lemma 2.5: The nodes and the packets belonging to the delay sequence are obtained by taking the first w elements of the sequences $p_1, \dots, p_v$ and $s_1, \dots, s_v$. The sequence of ranks is $\mathrm{rank}(p_1), \dots, \mathrm{rank}(p_v)$. This is in decreasing order by construction. The delay path is obtained from Lemma 2.8. This has length at most $L + 2w/q$ as required. To complete the proof we observe that all p_i must be real packets, i.e., not EOS or ghost packets, since they delay other packets as well as wait in queues. $\square$

Theorem 2.9 *For any constant k_1, there is a constant k_2 such that the probability that any packet is not delivered by step $L + w$, where $w = k_2 c + o(L + \log N)$ and $R \ge w$, is at most $1/N^{k_1}$.*

Proof: By Lemma 2.5, to bound the probability that some packet is delayed w steps, it suffices to bound the probability that some $(L+2w/q, w, R)$-delay sequence occurs. We begin by counting the number of $(L+2w/q, w, R)$-delay sequences. The delay path can start on any packet's path. Since there are N packets and each follows a path of length at most L, there are at most $N(L+1)$ possible starting points. At each node on the path, there are at most 2Δ choices for the next node on the path. Thus, the number of paths is at most $N(L+1)(2\Delta)^{L+2w/q}$. The number of ways of locating the nodes $s_0, s_1, \dots, s_w$ on the path is at most $\binom{L+2w/q+w}{w}$. The number of ways of choosing the packets $p_1, p_2, \dots, p_w$ such that for $1 \le i \le w$, packet p_i passes through node s_i is at most $(\Delta c)^w$. The number of ways of choosing ranks $r_1, r_2, \dots, r_w$ such that $r_i \ge r_{i+1}$ for $1 \le i < w$ and $1 \le r_i \le R$ for $1 \le i \le w$ is at most $\binom{R+w}{w}$. Each of these delay sequences occurs with probability at most $1/R^w$. Hence, the probability that any delay sequence occurs is at most

$$\frac{(L+1)N(2\Delta)^{L+2w/q}\binom{L+2w/q+w}{w}(\Delta c)^w\binom{R+w}{w}}{R^w}.$$

Using the inequality $\binom{a}{b} \le 2^a$ to bound $\binom{L+2w/q+w}{w}$ and the inequalities $\binom{a}{b} \le (ae/b)^b$ and $R + w \le 2R$ to bound $\binom{R+w}{w}$, the probability is at most

$$2^{\log(L+1)+\log N+(\log\Delta+2)(L+2w/q)} \cdot \left(\frac{4e\Delta c}{w}\right)^w.$$

Observing that $\log(L+1) \leq L + 2w/q$ and factoring $(2^{2(\log\Delta+3)/q})^w$ out of the first factor, our upper bound becomes

$$2^{\log N + (\log\Delta+3)L} \cdot \left(\frac{2^{2(\log\Delta+3)/q+2}e\Delta c}{w}\right)^w .$$

If $c = \Omega(L + \log N)$, then for any k_1, there is a k_2 such that for $w = k_2 c$, the probability is at most $1/N^{k_1}$. If $c = o(L + \log N)$, then for any k_1, there is a κ such that $\kappa = \omega(c)$ and $\kappa = o(L + \log N)$, and for any $w \geq \kappa$, the probability is at most $1/N^{k_1}$. $\square$

2.3.1 Packet combining

For simplicity, we have heretofore ignored the possibility of *combining* multiple packets with the same destination. In many routing applications, there is a simple rule that allows two packets with the same destination to be combined to form a single packet, should they meet at a node. For example, one of the packets may be discarded, or the data carried by the two packets may be added together. Combining is used in the emulation of concurrent-read concurrent-write parallel random-access machines [29] and distributed random-access machines [23].

If the congestion is to remain a lower bound when combining is allowed, then its definition must be modified slightly. The new congestion of an edge is the number of different destinations for which at least one packet's path uses the edge. Thus, several packets with the same destination contribute at most one to the congestion of an edge.

In order to efficiently combine packets, we will use a random hash function to give all of the packets with the same destination the same rank. Since ties in rank are broken according to destination, a node will not send a packet in one of its queues unless it is sure that no other packet for the same destination will arrive later in another queue. Thus, at most one packet for each destination traverses an edge.

We assign ranks using the universal hash function [7]

$$\text{rank}(x) = \left(\left(\sum_{i=0}^{m-1} a_i x^i\right) \bmod P\right) \bmod R$$

which maps a destination $x \in [0, P-1]$ to a rank in $[0, R-1]$ with k-wise independence. Here P is a prime number greater than the total number of destinations, and the coefficients $a_i \in Z_P$ are chosen at random. We show below that it suffices to choose $R = \Omega(c + L + \log N)$. The random coefficients use $O(m \log P)$ random bits. In most applications, only $\log N$-wise independence is needed and the number of possible different destinations is at most polynomial in N, so the hash function requires only $O(\log^2 N)$ bits of randomness.

In the proof of Theorem 2.9, the ranks of the w packets in a delay sequence were chosen independently, i.e., with w-wise independence. In order to use a hash function with m-wise independence, where m may be much smaller than w, we need the following lemma, which shows that in any delay sequence there are smaller subsequences of many different sizes.

Lemma 2.10 *If an (l, w, R)-delay sequence occurs, then a $(2l/\alpha, w/2\alpha, 2R/\alpha)$-delay sequence occurs, for every $\alpha \geq 1$.*

Proof: Suppose that an (l, w, R)-delay sequence occurs. Divide the packet sequence $p_1, \ldots, p_w$ into α contiguous subsequences such that each subsequence has at least $\lfloor w/\alpha \rfloor \geq w/2\alpha$ packets. This also partitions the delay path into subpaths. Let l_i denote the length of the ith subpath and let R_i denote the range of ranks for the ith subsequence, i.e., R_i is the difference between the largest rank in subsequence i and the largest rank in subsequence $i - 1$. We know that there must be fewer than $\alpha/2$ segments with $R_i > 2R/\alpha$, since $\sum R_i = R$. Furthermore there must be fewer than $\alpha/2$ segments satisfying $l_i > 2l/\alpha$, since $\sum l_i = l$. Thus there must exist some segment for which $l_i \leq 2l/\alpha$ and $R_i \leq 2R/\alpha$. □

Theorem 2.11 *For any constant k_1, there are constants k_2 and k_3 such that if the rank of each packet is assigned in the range 0 through R using a hash function with $k_3(\log N + \log L)$-wise independence, the probability that any packet is not delivered by step $L + w$, where $w = k_2 c + o(L + \log N)$ and $R \geq w$, is at most $1/N^{k_1}$.*

Proof: The proof is similar to that of Theorem 2.9. If some packet is not delivered by step $L + w$ then by Lemma 2.5 an $(L + 2w/q, w, R)$-delay sequence occurs. By Lemma 2.10, for any $\alpha > 1$ a $(2(L+2w/q)/\alpha, w/2\alpha, 2R/\alpha)$-delay sequence also occurs. The hash function will be (w/α)-wise independent. We will show that for the right choices of w and α, it is unlikely that any such sequence occurs.

The number of different $(2(L + 2w/q)/\alpha, w/2\alpha, 2R/\alpha)$-delay sequences is bounded as follows. A delay path starts at node on some packet's path.

Thus, there at most $N(L+1)$ starting points for the path. At each node on the path, there are at most 2Δ choices for the next node on the path. Thus, the total number of ways to choose the path is at most $N(L+1)(2\Delta)^{2(L+2w/q)/\alpha}$. The number of ways of choosing $w/2\alpha$ switches on the path is at most $\binom{2(L+2w/q)/\alpha+w/2\alpha}{w/2\alpha}$. The number of ways of choosing $w/2\alpha$ packets that pass through those switches is at most $(\Delta c)^{w/2\alpha}$. The number of ways of choosing the ranks for the packets is at most $R \cdot \binom{2R/\alpha+w/2\alpha-1}{w/2\alpha-1}$ since there are R choices for the rank of the first packet, and the ranks of the other $w/2\alpha - 1$ differ from the first by at most $2R/\alpha$.

If the ranks of the packets are chosen using a $w/2\alpha$-wise independent hash function, then the probability that any particular delay sequence occurs is at most $1/R^{w/2\alpha}$. Thus, the probability that any delay sequence occurs is at most

$$\frac{N(L+1)(2\Delta)^{2(L+2w/q)/\alpha}\binom{2(L+2w/q)/\alpha+w/2\alpha}{w/2\alpha}(\Delta c)^{w/2\alpha} R \cdot \binom{2R/\alpha+w/2\alpha-1}{w/2\alpha-1}}{R^{w/2\alpha}}.$$

Using the inequality $\binom{a}{b} \leq 2^a$ to bound $\binom{2(L+2w/q)/\alpha+w/2\alpha}{w/2\alpha}$, and $c \leq N$ to bound $(\Delta c)^{w/2\alpha}$ by $2^{\log\Delta+\log N}(\Delta c)^{w/2\alpha-1}$, and $\binom{a}{b} \leq (ae/b)^b$, $w \leq R$, and $w/2\alpha - 1 \geq w/4\alpha$ to bound $\binom{2R/\alpha+w/2\alpha-1}{w/2\alpha-1}$ by $(10eR/w)^{w/2\alpha-1}$, our upper bound becomes

$$2^{2\log N+\log(L+1)+2(\log\Delta+2)L/\alpha+8(\log\Delta+2)/q+\log\Delta+1} \cdot \left(\frac{2^{8(\log\Delta+2)/q}20e\Delta c}{w}\right)^{w/2\alpha-1}.$$

Removing constants so that we can better understand the expression, we have

$$2^{\Theta(\log N+\log L+L/\alpha)} \cdot \left(\frac{c}{w}\right)^{\Theta(w/\alpha)}.$$

If $c = \Omega(L+\log N)$, then for any constant k_1 there are constants k_2 and k_3 such that for $w \geq k_2 c$ and $w/2\alpha \geq k_3(\log N + \log L)$, the probability is at most $1/N^{k_1}$. If $c = o(L+\log N)$ then for any k_1 there is a κ such that $\kappa = \omega(c)$ and $\kappa = o(L+\log N)$ and for $w \geq \kappa$, and $w/2\alpha = o(\log N + \log L)$, the probability is at most $1/N^{k_1}$. $\Box$

2.3.2 Variable-length messages

In the preceding discussion we assumed that packets were atomic. However, the algorithm as well as the analysis extends naturally to the case in which we have *messages* each consisting of several packets.

Theorem 2.12 *Consider a leveled network with depth L. Suppose that initially the nodes in the network hold a total of N messages, where each message is at most m packets long. Let C denote the message congestion, i.e., the maximum number of messages that pass through any edge. For any constant k_1, there is a constant k_2 such that the probability that any message is not delivered by step $L + w$, where $w = m(k_2C + o(L + \log N))$, is at most $1/N^{k_1}$, provided each edge queue is long enough to hold at least m packets.*

We can trivially prove the theorem by organizing the operation of the network into *message cycles* each consisting of m steps. During a message cycle, each node in the network can send and receive a single message on each edge. This is equivalent to a packet routing problem in which packets take m steps to cross each edge, and hence must complete in $k_2C + o(L + \log N)$ message cycles, or $m(k_2C + o(L + \log N))$ steps.

We note however that synchronizing the operation of the nodes into message cycles as described above is not necessary. In particular, it is possible to allow two changes:

1. Each node can operate upon the next message as soon as it is done with the previous, rather than having to wait until the end of the current message cycle. This will be useful if most of the messages are small.

2. It is possible to *pipeline* message transmission. Thus a node can start forwarding the first packet of the message with the smallest rank as soon as every incoming queue receives the first packet of its message. To achieve this, messages must be transmitted in a special format. Specifically, the rank must be placed in the leading packet in the message, followed by the destination address, followed by a *type field* that indicates whether the message is real, or a ghost or an EOS, with the data trailing at the end. If the rank cannot be accomodated in one packet, then the more significant bits of the rank must be transmitted before the less significant ones. With the message format as above, it is possible for each node to send outgoing message packets as soon as the corresponding packets arrive on all incoming edges. In fact message combining can also be made to work with pipelining [30, 31].

It is possible to show that Theorem 2.12 still applies. The analysis involves constructing a delay sequence and a counting argument similar that for Theorem 2.9.

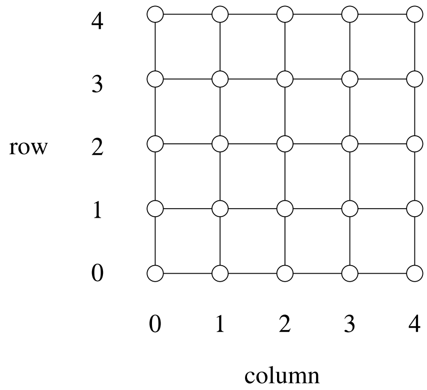

Figure 2: A 5×5 mesh.

3 Routing on meshes

In this section we apply the $O(c + L + \log N)$ scheduling algorithm to route N packets on a $\sqrt{N} \times \sqrt{N}$ mesh in $O(\sqrt{N})$ steps using constant-size queues. Although $O(\sqrt{N})$-step routing algorithms for the mesh were known before [13, 14, 39], they all have more complicated path selection strategies.

In an $n \times n$ *mesh*, each node has a distinct label (x, y), where x is its column and y is its row, and $0 \leq x, y \leq n - 1$. Thus, an $n \times n$ mesh has $N = n^2$ nodes. For $x < n - 1$, node (x, y) is connected to $(x + 1, y)$, and for $y < n-1$, node (x, y) is connected to $(x, y+1)$. A 5×5 mesh is illustrated in Figure 2. Sometimes *wraparound* edges are included, so that a node labeled $(x, n - 1)$ is connected to $(x, 0)$ and a node labeled $(n - 1, y)$ is connected to $(0, y)$.

Theorem 3.1 *With high probability, an N-node mesh can route any permutation of N packets in $O(\sqrt{N})$ steps using constant-size queues.*

Proof: The algorithm consists of four phases. In the first phase only those packets that need to route up and to the right are sent. The paths of the packets are selected greedily with each packet first traveling to the correct row, and then to the correct column. The level of a node is the sum of its row and column numbers. This simple strategy guarantees that both the congestion and the number of levels of the phase are $O(\sqrt{N})$. The packets are scheduled using the $O(c+L+\log N)$-step algorithm from Section 2. The up–right phase is followed by up–left, down–right, and down–left phases. □

4 Routing on butterflies

In this section we apply the scheduling algorithm from Section 2 to route N packets in $O(\log N)$ steps on an N-node butterfly using constant size queues. We essentially duplicate the result of Ranade [29], but the proof is simpler.

In a *butterfly network*, each node has a distinct label $\langle l, r\rangle$, where l is its level and r is its row. In an n-input butterfly, l is an integer between 0 and $\log n$, and r is a $\log n$-bit binary number. The nodes on level 0 and $\log n$ are called the inputs and outputs, respectively. Thus, an n-input butterfly has $N = n(\log n + 1)$ nodes. For $l < \log n$, a node labeled $\langle l, r\rangle$ is connected to nodes $\langle l+1, r\rangle$ and $\langle l+1, r^{(l)}\rangle$, where $r^{(l)}$ denotes r with the lth bit complemented. An 8-input butterfly network is illustrated in Figure 3. Sometimes the input and output nodes in each row are identified as the same node. In this case the number of nodes is $N = n \log n$. The butterfly has several natural recursive decompositions. For example, removing the nodes on level 0 (or $\log n$) and their incident edges leaves two $n/2$-input subbutterflies.

Theorem 4.1 *With high probability, an N-node butterfly can route any permutation of N packets in $O(\log N)$ steps using constant size queues.*

Proof: Routing is performed on a logical network consisting of $4\log n + 1$ levels. The first $\log n$ levels of the logical network are linear arrays. The packets originate in these arrays, one to a node. Levels $\log n$ through $2\log n$ form a butterfly network. Levels $2\log n$ through $3\log n$ consist of a butterfly with its levels reversed. The last $\log n$ levels are again linear arrays. Each packet has its destination in one of the arrays spanning levels $3\log n$ to $4\log n$. Packets with the same destination are combined. The butterfly simulates each step of this logical network in a constant number of steps. Paths for the packets are selected using Valiant's paradigm; each packet travels to a random intermediate destination on level $2\log n$ before moving on to its final destination. This strategy ensures that with high probability, say at least $1 - 1/N^{k_1}$, where k_1 is a constant, the congestion is $O(\log N)$. Since the paths are chosen independently of the ranks for the packets, the scheduling algorithm can treat the paths as if they were fixed. Assuming

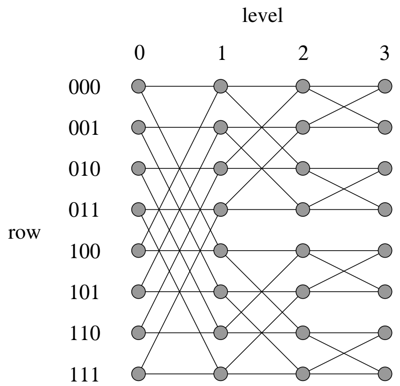

Figure 3: An 8-input butterfly network. Each node has a level number between 0 and 3, and a 3-bit row number. A node on level l in row r is connected to the nodes on level $l+1$ in rows r and $r^{(l)}$, where where $r^{(l)}$ denotes r with the lth bit complemented.

that the paths have congestion $O(\log N)$, by Theorem 2.9 the scheduling algorithm delivers all of the packets in $O(\log N)$ steps, with high probability, say at least $1-1/N^{k_2}$. Thus, the probability that either the congestion is too large or that the scheduling algorithm takes too long to deliver the packets is at most $1/N^{k_1} + 1/N^{k_2}$. □

Theorem 4.2 *With high probability, an n-input butterfly can route a random permutation of n packets from its inputs to its outputs in* $\log n + o(\log n)$ *steps.*

Proof: If each input sends a single packet, the congestion will be $O(\log n/\log\log n)$, with high probability. Given paths with congestion $O(\log n/\log\log n)$, by Theorem 2.9 the delay is $O(\log n/\log\log n)+o(\log N)$, with high probability. □

5 Routing on shuffle-exchange graphs

In this section, we present a randomized algorithm for routing any permutation of N packets on an N-node shuffle-exchange graph in $O(\log N)$ steps

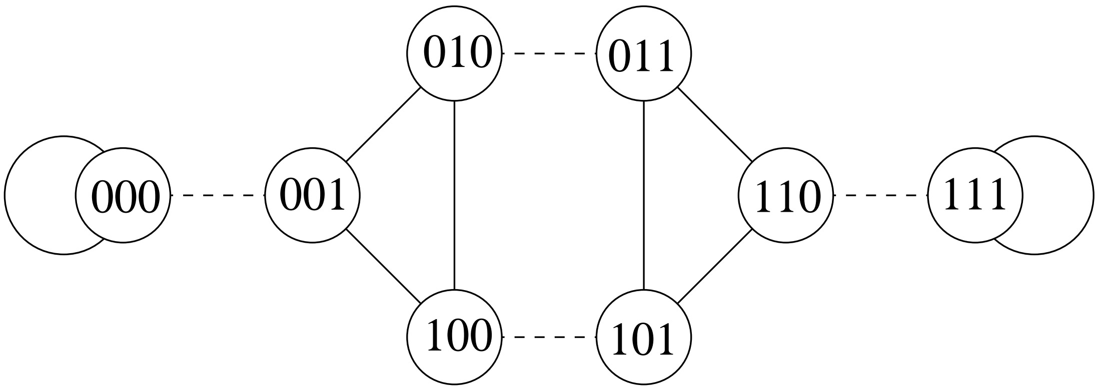

Figure 4: An 8-node shuffle-exchange graph. Shuffle edges are solid, exchange edges dashed.

using constant-size queues. The previous $O(\log N)$-time algorithms [2] required queues of size $\Omega(\log N)$.

Figure 4 shows an 8-node shuffle-exchange graph. Each node is labeled with a unique $\log N$-bit binary string. A node labeled $a = a_{\log N-1} \cdots a_0$ is linked to a node labeled $b = b_{\log N-1} \cdots b_0$ by a *shuffle* edge if rotating a one position to the left or right yields b, i.e., if either $b = a_0 a_{\log N-1} a_{\log N-2} \cdots a_1$ or $b = a_{\log N-2} a_{\log N-3} \cdots a_0 a_{\log N-1}$. Two nodes labeled a and b are linked by an *exchange* edge if a and b differ in only the least significant (rightmost) bit, i.e., $b = a_{\log N-1} \cdots a_1 \overline{a}_0$. In the figure, the shuffle edges are solid, and the exchange edges are dashed.

The removal of the exchange edges partitions the graph into a set of connected components called *necklaces*. Each necklace is a ring of nodes connected by shuffle edges. If two nodes lie on the same necklace, then their labels are rotations of each other. Due to cyclic symmetry, the number of nodes in the necklaces differ. For example, in a 64-node shuffle-exchange graph, the nodes 010101 and 101010 form a 2-node necklace, while 011011, 110110, and 101101 form a 3-node necklace. For each necklace, the node with the lexicographically minimum label is chosen to be the necklace's *representative*.

5.1 Good and bad nodes

Unlike the mesh and butterfly networks, the shuffle-exchange graph cannot emulate a leveled network in a transparent fashion. Nevertheless, it is still possible to apply the $O(c + L + \log N)$ scheduling algorithm for leveled networks to the problem of routing on the shuffle-exchange graph. The key idea is that a large subset of the shuffle-exchange graph (at least $N/5$ nodes) can emulate a leveled network. We call these nodes *good* nodes. The rest of the nodes are *bad*.

A node can be classified as bad for one of three reasons:

1. its label does not contain a substring of $\log\log N$ consecutive 0's (we consider the rightmost and leftmost bits in a label to be consecutive) (*type 1*),

2. its label contains at least two disjoint longest substrings of at least $\log\log N$ consecutive 0's (*type 2*), or

3. its label is $0 \cdots 0$ (*type 3*).

Thus, the label of every good node contains a unique longest substring of 0's with length at least $\log\log N$. For simplicity, we assume that $\log\log N$ is integral, and that $\log N \gg \log\log N$.

Since the length of a substring of consecutive 0's in a label is not changed by rotation, a necklace consists either entirely of good nodes or entirely of bad nodes. Furthermore, each good necklace consists of $\log N$ good nodes since a unique longest substring of consecutive 0's precludes cyclic symmetry.

In order to route packets among all N nodes of the shuffle-exchange graph, we associate the bad nodes with good nodes. A type-1 bad node is associated with a good node by changing the least significant bit of its label to a 1 and the $\log\log N$ most significant bits to 0's. Each bad necklace of type 2 is associated with a good necklace by changing the two bits following the leftmost group of 0's in its representative's label to 01. Finally, the node $0 \cdots 0$ is associated with its neighbor $0 \cdots 01$.

Lemma 5.1 *At most $4\log N$ bad nodes are associated with any good necklace.*

Proof: Each type-1 bad node is associated with the representative of a good necklace since, after the transformation, the longest string of consecutive 0's begins with the most significant bit. Only type-1 bad nodes whose labels differ from the representative's label in at most $\log\log N+1$ bits are associated with it, so at most $2^{\log\log N+1} = 2\log N$ type-1 bad nodes are associated with any good necklace.

To assess the number of type-2 bad nodes associated with a good necklace, we consider the label of the representative of the good necklace and notice that only a bad necklace whose representative's label differs in the last bit of its leading block of 0's and possibly the bit after that can be mapped to the good necklace. Thus, at most two type-2 bad necklaces are associated with any good necklace.

Finally, no bad nodes of either type 1 or 2 are associated with the necklace of node $0\cdots01$. □

Corollary 5.2 *At least $N/5$ of the nodes are good.*

Proof: By Lemma 5.1 at most $4\log N$ bad nodes are associated with any good necklace. Since every good necklace contains exactly $\log N$ nodes, at least $N/5$ of the nodes are good. □

The remainder of this section provides the details of the routing algorithm. We begin by describing a logical leveled network that the good nodes can easily emulate with constant overhead. Next, we show that for any routing problem, choosing random intermediate destinations yields paths with congestion and dilation $O(\log N)$ in this network, with high probability. Thus, by applying the analysis of Section 2, routing on the logical network takes $O(\log N)$ steps with high probability, and uses constant-sized queues. We conclude by describing a deterministic algorithm for routing between good and bad nodes.

5.2 A leveled network

The level of a node is determined by the distance to the representative node in its necklace. An alternate way to write a node's label is to place a line under its least significant bit (which we call the *current* bit), and then rotate the label until it matches its representative's label. For example, 110001 can also be written $000\underline{1}11$. The *level* of a node is the position of the current bit, starting with zero and counting from the left. For example, $00\underline{0}111$ lies on level 2. (Note that the representative node lies on level $\log N-1$.)

The problem with this leveling scheme is that although it induces a leveling of the shift edges, it does not necessarily induce a leveling of the exchange edges. An exchange edge may create a new longest substring of

0's by appending two substrings separated by a single 1, and thus connect two levels that are very far apart.

To overcome this difficulty, we replace the exchange edges with *flip* edges. A flip edge links nodes labeled a and b if both are good, $a = a_{\log N-1} \cdots \underline{a_j} \cdots a_0$, $b = a_{\log N-1} \cdots \overline{a}_j \underline{1} \cdots a_0$, $j > 0$, and a_j is not in the longest block of 0's of a. Note that a flip edge extends a group of 0's by at most one. Thus no flip edge can create a new leading group of 0's, because if it grew a shorter group to be as long as the leading group, then it would lead to a bad node of type 2, a contradiction since flip edges occur only between good nodes by definition. Thus flip edges are leveled. The operation of the flip edges can be emulated by the shuffle-exchange graph with only a constant factor of slowdown; each flip edge is composed of an exchange edge, a shuffle edge, and possibly another exchange edge.

We denote by A the network composed of the good nodes, the shuffle edges (excluding the shuffle edges from level $\log N - 1$ to 0), and the flip edges. Note that in network A, from the level 0 node of any necklace it is possible to reach any other necklace whose longest string of 0's has the same or greater length by correcting bits starting from the end of the leading block of 0's.

In fact, we wish to be able to get from the level 0 node of a necklace to *any* other necklace. Thus we append a mirror image of A to itself so that from any level 0 node it is possible to reach necklaces with fewer 0's in the longest string. The leveling is extended in the natural manner. We call this network AA^r, and note that network A can emulate it with constant slowdown.

We denote by L the network consisting of the shuffle edges on the good nodes, again excluding shuffle edges from level $\log N - 1$ to level 0. Our method of path selection consists of routing from a good node to the level 0 node in its necklace, then routing to a random intermediate necklace, then routing to the destination necklace, and finally routing to the appropriate good node. Thus, we route in a leveled network composed of network L, network AA^r, another network AA^r, and another network L. We extend the leveling in the natural manner and note that network A can emulate the whole thing with constant slowdown.

5.3 Path selection and congestion

For each packet we choose its path by uniformly choosing a random good necklace for it to route through before it goes to its final destination. So the path for a packet consists of a path through L to the node on level 0 of its necklace, a path through AA^r to its random intermediate necklace, a path through the second AA^r to its destination necklace, and a path through the second L to the proper node of its destination necklace.

The following lemma shows that if at most $O(\log N)$ packets originate and terminate in each good necklace, then this method yields paths with congestion $O(\log N)$ with high probability.

Lemma 5.3 *Suppose that each good necklace sends and receives at most $b \log N$ packets, where b is a fixed constant. Then for any constant k_1, there is a constant k_2 such that the probability that more than $k_2 \log N$ packets use any edge is at most $1/N^{k_1}$.*

Proof: We observe that for the paths in the copies of L, we have congestion at most $b \log N$, since at most $b \log N$ packets start or end in any good necklace. By symmetry we claim that the analysis of the path portions in both copies of AA^r is the same. Finally we recall that in AA^r, we route packets going to intermediate destination necklaces with fewer 0's straight across (i.e, without using any flip edges) in network A. Thus, the congestion of the straight across paths in A is at most $b \log N$. Also, we route packets going to intermediate necklaces with the same or more 0's straight across in network A^r. We will show that any intermediate necklace gets $O(\log n)$ packets with high probability, so the straight across portion of the paths in A^r will have $O(\log N)$ congestion. To finish, we analyze the congestion in network A due to packets routing to intermediate necklaces with the same or more 0's, and claim that the arguments will hold by symmetry for AA^r.

Consider a shuffle or flip edge e in the first copy of network A. Suppose that e traverses levels m and $m+1$. Let x be the length of the longest string of 0's in the necklace to which e goes. If $m < x$, then e must be a shuffle edge and no packet from any other necklace can use e, since we only map to a necklace via flip edges after its longest string of 0's. Otherwise ($m \geq x$) we consider the number of packets from other necklaces that can use e. We know that only packets from at most 2^l other necklaces with $l = m - \log\log N$ can use e since at most l bits can change by level $m+1$ (there are no flip edges in the first $\lg\lg N$ levels of any necklace). Thus the number of packets that can use e is at most $b \cdot 2^l \log N$ since each necklace starts with at most $b \log N$ packets. The probability that a specific packet uses e is the number of necklaces that can be reached using e, at most $2^{\log N - m} = 2^{\log N - \log\log N - l}$ (i.e., necklaces that match e's necklace in the first $m = l + \log\log N$ bits), divided by the total number of good necklaces, at least $N/5\log N$, which is just $5/2^l$.

The probability that more than $k_2 \log N$ packets use e is at most

$$\binom{b \cdot 2^l \log N}{k_2 \log N} \left(\frac{5}{2^l}\right)^{k_2 \log N},$$

since there are $b \cdot 2^l \log N$ Bernoulli trials, each succeeding with probability $5/2^l$. The probability that any of the $O(N)$ edges of this stage has congestion more than $k_2 \log N$ is $O(N)$ times this probability. Using the inequality $\binom{a}{b} \leq (ae/b)^b$, for any k_1 we can bound the product by $1/N^{k_1}$ by choosing k_2 large enough. □

Because the congestion and number of levels are $O(\log N)$, with high probability, the time to route the packets between the good nodes is also $O(\log N)$, with high probability, and the queue size is constant.

5.4 Packets from bad nodes

In this section we show how to deterministically route the packets from the bad nodes to their associated good necklaces.

Lemma 5.4 *Packets from bad nodes can be routed to the associated good necklaces deterministically in $O(\log N)$ time using constant-size queues.*

Proof: Recall that we associate a bad node of type 1 with the necklace represented by a 1 in the least significant or current bit plus $\log\log N$ 0's in the most significant bits. We route these packets in the shuffle exchange graph by changing the current bit to a 1 (if it is 0) and changing $\log\log N$ bits to the right to 0's. Thus we map a bad node to a good necklace at its level $\log\log N$ node.

For any necklace, the shuffle-exchange graph emulates binary tree whose leaves are mapped to the necklace. Each edge of the tree is emulated by either a shuffle edge or an exchange edge followed by a shuffle edge. Each level of the tree corresponds to one of the $\log\log N+1$ bits that were changed. Therefore, we can route packets from the binary tree leaves to the necklace, and distribute them along the necklace deterministically. This is easily done in $O(\log N)$ time with constant queues. The routing from the necklace to the tree is equally trivial. But, we need to ensure that traffic from the separate binary trees does not interfere too much. This is easy since any bad node is in at most two binary trees; in at most one as a leaf since any node is mapped to exactly one good node, and in at most one as an internal node since the number of 0's between the current bit and the closest 1 to the left determines a unique level and the rest of the bits determine a unique tree.

To finish, we consider bad nodes of type 2. These are nodes without a unique longest string of 0's. Here we extend one of the groups of 0's by one 0, making sure not to join two groups of 0's by inserting a 1, thus mimicking the flip operation. For any good necklace whose representative is $0^k1...$ only the necklaces represented by $0^{k-1}10...$ and $0^{k-1}11...$ can be mapped to it. Again, at most two bad necklaces are associated with any good necklace.

For each packet in such a bad necklace we route it through the node connecting it to the appropriate good necklace. We perform this movement by pipelining the packets through the flip edge that connects the two necklaces. We see that this mapping maps at most one packet from the bad necklace to a node in the good necklace. Since we are basically routing on linear arrays of length $O(\log N)$, $O(\log N)$ steps are sufficient to route the packets from the two bad necklaces.

This finishes the description of the maps to and from all the bad nodes except for node $0 \cdots 0$, which is adjacent to node $0 \cdots 01$. □

5.5 Summary

The main result of this section is summarized in the following theorem.

Theorem 5.5 *With high probability, an N-node shuffle-exchange graph can route any permutation of N packets in $O(\log N)$ steps using constant-size queues.*

Proof: There are three phases to the algorithm. First, packets originating at bad nodes are deterministically routed to the good nodes with which they are associated. By Lemma 5.4 this phase requires $O(\log N)$ steps. Next, packets are routed between the good nodes on the logical network. Since at most $4 \log N$ bad nodes are associated with each good necklace, with high probability the congestion of the paths on the logical network is $O(\log N)$, by Lemma 5.3. Thus, this phase requires $O(\log N)$ steps, with high probability. The packets are routed in $O(\log N)$ steps using the scheduling algorithm from Section 2. Finally, packets destined for bad nodes are deterministically routed from the good nodes to bad. By an analysis similar to that of Lemma 5.4, this phase also requires $O(\log N)$ steps. □

6 Routing on multidimensional arrays

In this section we describe a randomized algorithm for routing kN packets on an N-node k-dimensional array in $O(kM)$ steps using constant-size queues, where M is the maximum of the k side lengths of the array. Special cases include the mesh ($k = 2$) and the hypercube ($M = 2$). For arrays of dimension greater than two, no asymptotically-optimal constant-queue-size routing algorithms were previously known.

A *k-dimensional array* with side lengths $M_i \geq 2$, for $1 \leq i \leq k$, has $N = \prod_{i=1}^{k} M_i$ nodes and $2kN$ edges. Each node has a distinct label $(w_1, \ldots, w_k)$, where $0 \leq w_i \leq M_i - 1$, for $1 \leq i \leq k$. A node has two edges for each dimension; for $1 \leq i \leq k$, $(w_1, \ldots, w_k)$ has an edge to $(w_1, \ldots, w_i + 1 \bmod M_i, \ldots, w_k)$ and to $(w_1, \ldots, w_i - 1 \bmod M_i, \ldots, w_k)$. (If $M_i = 2$, as in the hypercube, then the node has only one edge in dimension i. In this case, the total number of edges is only kN.) We assume that at each step, a node may simultaneously transmit and receive a packet on each of its $2k$ edges, even if k is not constant.

In order to apply the scheduling algorithm from Section 2, routing is performed on a bounded-degree leveled *logical network* that the array emulates. (Note that the degree of the array itself is not necessarily constant.) The logical network consists of $(2k+1)$ *plateaus* labeled 0 through $2k$, each consisting of N logical nodes. Each node in the logical network has a label $(w_1, \ldots, w_k)$, where $0 \leq w_j \leq M_j$ for $1 \leq j \leq k$, that is distinct from the labels of the other nodes on the same plateau. Each node in the logical network has at most two incoming edges and two outgoing edges. We begin by describing the edges in plateaus 0 through k. A node on plateau i has edges only in dimensions i and $i + 1$. If $i > 0$ and $w_i < M_i - 1$, then the node labeled $(w_1, \ldots, w_k)$ has an edge to the node in the same plateau with label $(w_1, \ldots, w_i + 1, \ldots, w_k)$. Also, if $i < k$ and $w_{i+1} < M_{i+1} - 1$, then the node has an edge to $(w_1, \ldots, w_{i+1} + 1, \ldots, w_k)$ on the same plateau. The only connections to plateau $i + 1$ come from nodes with $w_{i+1} = M_{i+1} - 1$. For $i < k$, $(w_1, \ldots, w_i, M_{i+1} - 1, w_{i+2}, \ldots, w_k)$ is connected to $(w_1, \ldots, w_i, 0, w_{i+2}, \ldots, w_k)$ on plateau $i + 1$. Plateau k is connected to plateau $k + 1$ by dimension 1 edges. Plateaus $k + 1$ through $2k$ are essentially a copy of plateaus 1 through k. The edges on plateau i, $k + 1 \leq i \leq 2k$ are given by the same rules as the edges on on plateau $i - 2k$. The level of node $(w_1, \ldots, w_k)$, $0 \leq i \leq k$, is $\sum_{j=1}^{k} w_j + \sum_{j=1}^{i} M_j$. For $k \leq i \leq 2k$, the level is $\sum_{j=1}^{k} w_j + \sum_{j=1}^{i-k} M_j + \sum_{j=1}^{k} M_j$. The network is leveled because each edge connects a pair of nodes on adjacent levels.

Each step of the logical network can be emulated by the array in a constant number of steps. The array node labeled $(w_1, \ldots, w_k)$ emulates the $2k+1$ logical nodes labeled $(w_1, \ldots, w_k)$, one on each plateau. The array edge from $(w_1, \ldots, w_i, \ldots, w_k)$ to $(w_1, \ldots, w_i + 1 \bmod M_i, \ldots, w_k)$ emulates at most four logical edges, one each on plateaus $i-1$, i, $k+i-1$ and $k+i$. Note that even though k may not be a constant, we are assuming that each node can process k packets in a single step.

Paths for the packets are selected using Valiant's paradigm. Initially each node on plateau 0 holds k packets in an initial queue. A packet travels from its origin on plateau 0 to a random destination on plateau k, then continues on to its true destination on plateau $2k$. Suppose that a packet originating at $(x_1, \ldots, x_k)$ on plateau 0 is to pass through $(r_1, \ldots, r_k)$ on plateau k on its way to $(y_1, \ldots, y_k)$ on plateau $2k$. In the first half of the path plateau i is used to make the ith component of the packet's location match the ith component of its random destination, for $1 \leq i \leq k$. The packet enters plateau $i \geq 1$ at node $(r_1, \ldots, r_{i-1}, 0, x_{i+1}, \ldots, x_k)$ and traverses dimension i edges to $(r_1, \ldots, r_i, x_{i+1}, \ldots, x_k)$. The packet then traverses dimension $i+1$ edges to $(r_1, \ldots, r_i, M_{i+1} - 1, x_{i+2}, \ldots, x_k)$ and crosses over to node $(r_1, \ldots, r_i, 0, x_{i+2}, \ldots, x_k)$ on plateau $i+1$. In the second half of the path, plateau $k+i$ is used to make the ith component of the packet's location match the ith component of the true destination in a similar fashion. The following lemma shows that with high probability, the congestion c of the paths is at most $O(kM)$, where M is the maximum side length.

Lemma 6.1 *For any constant k_1, there is a constant k_2 such that the probability that $c > k_2 kM$ is at most $1/N^{k_1}$.*

Proof: We analyze congestion in the first half of the network only. The calculation for the second half is identical.

We begin by bounding the probability that a particular edge is congested. There are two parts to the calculation: counting the number of packets that can possibly use the edge and bounding the probability that an individual packet actually does so. First, we count packets that can use the edge. Consider an edge on plateau $i-1$ or i from $(w_1, \ldots, w_k)$ to $(w_1, \ldots, w_i + 1 \bmod M_i, \ldots, w_k)$. Since a packet does not use any dimension $i+1$ through k edges before it uses a dimension i edge, any packet that uses the edge must come from an origin whose last $k-i$ components x_{i+1} through x_k match w_{i+1} through w_k. There are at most $M_1 \cdots M_i$ such origins, each transmitting k packets. Next we bound the probability that each of these packets actually uses the edge. A packet uses the edge only if components r_1 through r_{i-1} of its random destination match w_1 through w_{i-1}. The probability that these components match is $1/M_1 \cdots M_{i-1}$.

Since the random destinations are chosen independently, the number of packets S that pass through the edge has a binomial distribution. The probability that more than $k_2 kM$ packets use an edge is at most

$$\Pr[S > k_2 kM] \leq \binom{kM_1 \cdots M_i}{k_2 kM} \left(\frac{1}{M_1 \cdots M_{i-1}}\right)^{k_2 kM}.$$

Using the inequalities $M_i \leq M$ for $1 \leq i \leq k$, and $\binom{a}{b} \leq \left(\frac{ae}{b}\right)^b$, we have $\Pr[S > k_2 kM] \leq \left(\frac{e}{k_2}\right)^{k_2 kM}$.

To bound the probability that any edge is congested, we simply sum the probabilities that each particular edge is congested, i.e.,

$$\Pr[c > k_2 kM] \leq 4kN \left(\frac{e}{k_2}\right)^{k_2 kM}.$$

Since $kM > \log N$, for any k_1, there is a k_2 such that this probability is at most $1/N^{k_1}$. □

Theorem 6.2 *For any constant k_1, there is a constant k_2 such that the probability that any packet is not delivered by step $k_2 kM$ is at most $1/N^{k_1}$.*

Proof: With high probability, the scheduling algorithm from Section 2 delivers all packets in $O(c + L + \log N)$ steps. The number of levels L is $O(kM)$, and by Lemma 6.1 with high probability the congestion c is $O(kM)$. Also, $\log N < kM$. □

7 Construction of area and volume-universal networks

In this section we construct a class of networks that are *area-universal* in the sense that a network in the class with N nodes has area $O(N)$ and can, with high probability, simulate in $O(\log N)$ steps each step of any network of area $O(N)$. The networks are based on the fat-trees of Greenberg and Leiserson [10] and the simulation uses the packet routing algorithm from Section 2.

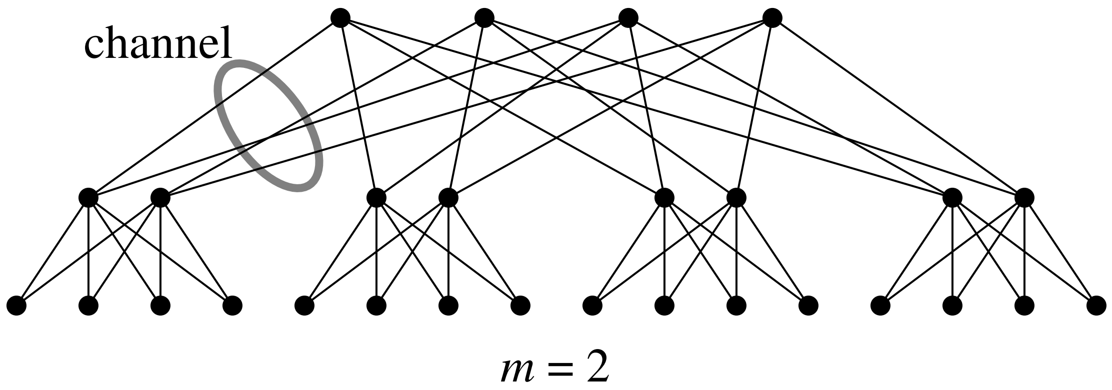

Figure 5: A fat-tree.

Leiserson was the first to display a class of networks that could efficiently simulate any other network of the same area or volume. In [21] he showed that a fat-tree of area $O(N)$ can simulate in $O(\log^3 N)$ bit-steps each bit-step of any network of area $O(N)$. (In the *bit model* an edge can transmit a single bit in each time step. All of the algorithms in this paper are described in terms of the packet model, in which an edge can transmit a packet of at least $\log N$ bits in one step.) Leiserson's simulation used an off-line routing algorithm for fat-trees. On-line routing algorithms were later developed by Greenberg and Leiserson [10].

A fat-tree network is shown in Figure 5. Its underlying structure is a complete 4-ary tree. Each edge in the 4-ary tree corresponds to a pair of oppositely directed groups of edges called *channels*. The channel directed from the leaves to the root is called an up channel; the other is called a down channel. The *capacity* of a channel C, $\text{cap}(C)$, is the number of edges In the channel. We call the tree "fat" because the capacities of the channels grow by a factor of 2 at every level. A fat-tree of height m has $M = 2^m$ nodes at the root and $M^2 = 2^{2m}$ leaves.

It will prove useful to label the nodes at the top and bottom of each channel. Let the level of a node be its distance from the leaves. Suppose a channel C connects $\text{cap}(C)/2 = 2^l$ nodes at level l with $\text{cap}(C) = 2^{l+1}$ nodes at level $l+1$. Give the nodes at level l labels 0 through $2^l - 1$ and the nodes at level $l+1$ labels 0 through $2^{l+1} - 1$. Then node k at level l is connected to nodes k and $k + 2^l$ at level $l+1$. The following lemma relates the labels of the nodes on a packet's path from a leaf to the root.

Lemma 7.1 *There is a unique shortest path from any leaf to a node labeled k at the root, for $0 \leq k \leq M-1$, and that path passes through a node labeled $k \bmod 2^l$ at level l, for $0 \leq l \leq m$.* $\square$

The simplest way to route packets in a leveled fashion on a fat-tree is to route every packet all of the way up to the root before routing it down to its destination. The problem with this strategy is that it may cause unnecessary congestion at the root. For example, suppose that every leaf wants to send a single packet to its sibling. If these packets are sent to the root, then $M^2/2$ packets will pass through each channel at the root. On the other hand, if each packet goes up just one level before turning around, then at most one packet will pass through any channel in the entire network. Thus, we will route every packet from its origin to its destination along a path that passes through as few channels as possible.

For a set Q of packets to be delivered between the leaves of the fat-tree, we define the *load* of Q on a channel C, $\text{load}(Q, C)$, to be the number of different destinations of packets in Q for which at least one packet must pass through C. (A packet must pass through C only if every path from the packet's origin to its destination passes through the C.) Note that even if many packets with the same destination must pass through a channel, that destination contributes at most one to the load of the channel. The routing algorithm will combine all packets with the same destination that attempt to pass through the channel. We define the *load factor* of Q on C, $\lambda(Q, C)$, to be the ratio of the load of Q on C to the capacity of C, $\lambda(Q, C) = \text{load}(Q, C)/\text{cap}(C)$. The load factor $\lambda(Q)$ on the entire network is simply the maximum load factor on any channel $\lambda(Q) = \max_C \lambda(Q, C)$. The load factor is a lower bound on the the number of steps required to deliver Q. We shall sometimes write λ to denote $\lambda(Q)$ when the set of packets to be delivered is clear from the context.

In a *leveled fat-tree* a node at the top of an up channel at level l is connected to itself at the top of the corresponding down channel by a linear chain of nodes of length $2(m-l)$. A packet may only make a transition from an up channel to a down channel by traversing a chain. Thus all shortest paths between leaves in a leveled fat-tree have length $2m$. Note that the load of a set of packets on a channel of the leveled fat-tree is identical to the load on the corresponding channel in the fat-tree.

The path that a packet for destination x takes through a leveled fat-tree is determined by the m-universal hash function [7]

$$\text{path}(x) = \left(\left(\sum_{i=0}^{m-1} a_i x^i \right) \bmod P \right) \bmod M,$$

where P is a prime number larger than the number of possible different destinations, and the $a_i \in \mathbf{Z}_P$ are chosen at random off-line. A packet with destination x follows up channels along the unique shortest path to the node labeled path(x) at the root until it can reach x without using any more up channels. It then crosses over to a down channel via a chain, and follows down channels to x. Note that a packet only passes through a channel if all paths from its origin to its destination pass through that channel. Also, all packets with destination x that pass through channel C pass through node (path(x) mod cap(C)) at the top of C and through node (path(x) mod (cap$(C)/2$)) at the bottom of C.

The following lemma shows that we can use the scheduling algorithm from Section 2 to route packets in a fat-tree.

Lemma 7.2 *For any constant c_1, there is a constant c_2 such that the probability that the number of steps required to deliver a set Q of N packets with load factor λ is more than $c_2(\lambda + \log M)$ is at most $1/M^{c_1}$, provided that N is polynomial in M.*

Proof: The paths of the packets are first randomized using the universal hash function path. With high probability, the resulting congestion is $c = O(\lambda + \log M)$. Each packet travels a distance of $L = 2m = 2\log M$. The packets are then scheduled using the algorithm from Section 2. □

Let us now consider the VLSI area requirements [34] of fat-trees. A fat-tree with root capacity M and $\Theta(M^2)$ nodes has a layout with area $O(M^2 \log^2 M)$ that is obtained by embedding the fat-tree in the tree of meshes [15]. The nodes of the tree of meshes in this layout are separated by a distance of $\log M$ in both the horizontal and vertical directions. Thus, the $\Theta(\log M)$ space for the chain associated with each node in the leveled fat-tree can be allocated without increasing the asymptotic area of the layout. (In fact, it is possible to attach a chain of size $O(\log^2 M)$ to each fat-tree node without increasing the area by more than a constant factor.) The leaves of the fat-tree are separated in the layout from each other by a distance of $\log M$ in each direction. We can improve the density of nodes without increasing the asymptotic area of the layout by connecting a $\log M \times \log M$ mesh of nodes to each leaf. The resulting network has $\Theta(M^2 \log^2 M)$ nodes and area $\Theta(M^2 \log^2 M)$. The N-node network in this class has root capacity $\Theta(\sqrt{N}/\log N)$, $\Theta(N/\log^2 N)$ leaves, and area $\Theta(N)$.

The following theorem shows that this class of networks is area-universal.

Theorem 7.3 *With high probability, an N-node network U of area $\Theta(N)$ can simulate in $O(\log N)$ steps each step of any network B of area $O(N)$.*

Proof: The nodes of network B are mapped to the nodes of the area-universal network U off-line using a recursive decomposition technique as in [21]. In each step, an edge of B is simulated by routing packets between the nodes that it connects. At each level of the recursion at most $O(\text{cap}(C) \cdot \log N)$ edges connect the nodes mapped below a channel C with the rest of the network. This property of the mapping ensures that the load factor of each set of packets used in the simulation of B is at most $O(\log N)$. At the bottom of the decomposition tree, a $O(\log N) \times O(\log N)$ region of the layout of B is mapped to each leaf of the fat-tree. The $O(\log N) \times O(\log N)$ mesh connected to the leaf in U simulates this region of B with $O(\log N)$ slowdown using a mesh routing algorithm such as the one in Section 3. $\square$

The study of fat-tree routing algorithms that perform combining was motivated in part by an abstraction of the volume and area-universal networks called the distributed random-access machine (DRAM). A host of conservative algorithms for tree and graph problems for the exclusive-read exclusive-write (EREW) DRAM are presented in [22]. Recently we discovered conservative concurrent-read concurrent-write (CRCW) algorithms that require fewer steps for some of these problems [23]. Until now, however, no efficient fat-tree routing algorithms that perform combining were known. The $O(\lambda + \log N)$ step routing algorithm presented here fills the void.

Only slight modifications to the area-universal fat-tree are necessary to make it volume universal [10]. The underlying structure of the volume-universal fat-tree is a complete 8-ary tree. Instead of doubling at each level, the channel capacities increase by a factor of 4. The tree has m levels, root capacity $M = 2^{2m}$, and $M^{3/2} = 2^{3m}$ leaves. The nodes at the top of a channel at level l are labeled 0 through $4^l - 1$. Node k at level l is connected to nodes k, $k + 4^l$, $k + 2 \cdot 4^l$, and $k + 3 \cdot 4^l$ at level $l + 1$. A layout with volume $O(M^{3/2} \log^{3/2} M)$ for the fat-tree can be obtained by embedding it in the three-dimensional tree of meshes. As before, a chain of size $O(\log^{3/2} M)$

can be attached to each node of the fat-tree without increasing the asymptotic layout area and the density of nodes can be improved by connecting a $\log^{1/2} M \times \log^{1/2} M \times \log^{1/2} M$ mesh to each leaf.

The simulation scheme in this section can also be used to simulate shared-bus networks. In a shared-bus network, an edge is allowed to connect more than two nodes. If several nodes attempt to send packets on the same edge in the same step, then the packets are combined using some simple rule to form a single packet. Combining is assumed to require a single step, regardless of the number of packets combined or the rule used. Because the fat-tree routing algorithm is capable of combining, it is no more difficult for the fat-tree to simulate shared-bus networks than to simulate point-to-point networks (i.e., networks in which each edge connects a pair of nodes). The simulation is optimal because a point-to-point network may require $\Omega(\log N)$ steps to simulate one step of a shared-bus network. Previously, no fat-tree routing algorithms were capable of combining packets to the same destination. As a consequence, no scheme for simulating shared-bus networks was known. A network that can simulate in $O(1)$ steps each step of any shared-bus network area of equal area was presented in [24]. However, the connections in that network are not fixed, but instead nodes communicate via reconfigurable buses.

8 Sorting on butterflies

In this section we present a randomized algorithm for sorting $N \log N$ packets on an $N \log N$-node butterfly network in $O(\log N)$ steps using constant-size queues. The algorithm is based on the *Flashsort* algorithm of Reif and Valiant [32]. The main difference is that we use the algorithm for scheduling packets on leveled networks in place of their scheduling algorithm, which requires queues of size $O(\log N)$. A similar approach has been suggested previously by Pippenger [26].

8.1 The algorithm

The basic outline of the algorithm is the same as that of Flashsort. The first step is to randomly select a small set of *splitters* from among the packets that are to be sorted. Next the splitters are sorted deterministically. The splitters partition the packets into *intervals*. The ith interval consists of those packets whose keys are larger than the key of the $(i-1)$st largest splitter, and smaller than the key of the ith largest splitter. (We assume without loss of generality that all of the keys are distinct.) Using the splitters as guides, each interval of packets is routed to a different subbutterfly, where it is sorted recursively.

We begin by describing a recursive algorithm for sorting $N/\log^{\alpha} N$ packets in $O(\log N)$ time on an $N \log N$-node butterfly, where α is some fixed constant greater than one. The butterfly is "lightly loaded" by this factor of $\log^{\alpha+1} N$ to ensure that, with high probability, at the lower levels of the recursion the number of packets to be sorted by each subbutterfly does not exceed the number of inputs to that subbutterfly. When the algorithm is invoked, each packet must reside at a distinct input. As we shall see, this algorithm can be combined with Leighton's *Columnsort* algorithm [16] to sort all $N \log N$ packets in $O(\log N)$ time.

The steps taken by a subbutterfly with M inputs are presented in some detail in Figure 6.

The first step in the algorithm is to count the number of packets entering the subbutterfly. Since the packets reside in distinct inputs, the total number of packets can be computed via a parallel prefix computation. The prefix computation can be performed in $O(\log M)$ time deterministically.

Next each packet independently chooses to be a splitter candidate with probability $\sqrt{M}/n$. As we shall see, with high probability the number of candidates is between $\sqrt{M}/2$ and $3\sqrt{M}/2$. This step requires only constant time.

The candidates are then sorted in $O(\log M)$ time using a simple deterministic algorithm based on counting due to Nassimi and Sahni [25].

After the candidates are sorted, every $(\log N)$th one in the sorted order is chosen to be a splitter. This *oversampling* technique, due to Reif, ensures that each of the intervals contains approximately the same number of splitters, with high probability. Note that we oversample by a factor of $\log N$, where N is the number of inputs in the entire network, independent of the number of inputs, M, of the subbutterfly on which the algorithm is invoked. Since with high probability there are at least $\sqrt{M}/2 \log N$ splitters, the subbutterflies at the next level of recursion should have at most $2\sqrt{M} \log N$ inputs.

Next the splitters are distributed throughout the M-input subbutterfly so that they can direct each interval of packets to the appropriate smaller subbutterfly. We distribute a copy of the median splitter to each node in level 0 of the M-input subbutterfly. Then we divide the splitters into upper and lower halves. We distribute a copy of the median splitter from the upper half to each node in the upper half of level 1. Similarly, we distribute a copy of the

1. Count the number of packets entering the M-input subbutterfly. Let the number of packets be denoted by n.
2. Randomly and independently, make each packet a candidate with probability $\sqrt{M}/n$.
3. Sort the candidates deterministically.
4. Select every $\log N$th candidate to be a splitter.
5. Distribute the splitters for splitter-directed routing.
6. Route each packet to a random row of the M-input subbutterfly.
7. Route each interval to a smaller subbutterfly via splitter-directed routing.
8. Distribute the packets in each interval to distinct inputs of the corresponding smaller subbutterflies.
9. Sort the intervals recursively.

Figure 6: The steps performed by an M-input subbutterfly in the recursive algorithm for sorting $N/\log^{\alpha} N$ packets in $O(\log N)$ time on an $N \log N$-node butterfly using constant-size queues.

median splitter from the lower half to each node in the lower half of level 1. The process continues in this fashion until all of the splitters are used up. At this point, every node in the first $\Theta(\log(\sqrt{M}/\log N))$ levels of the butterfly has a copy of a splitter. This step can be performed deterministically in $O(\log M)$ time.

After the splitters are positioned, each packet is routed to a random row of the M-input subbutterfly. The packets are scheduled using the algorithm for routing on leveled networks.

Each interval of packets is then routed to a different smaller subbutterfly. This step is called *splitter-directed routing* [32]. The paths of the packets are determined as follows. At level 0, each packet compares itself to the median splitter. If it is larger, it moves to the upper half of the second level, otherwise it moves to the lower half. The process is repeated at the level 1, with each packet being directed to the appropriate quarter of level 2, and so on. The packets are scheduled using the algorithm for routing on leveled networks. When all of the packets have been routed along in the butterfly as deeply as the splitters are assigned, each subbutterfly at that level picks new splitters and proceeds recursively.

The last step before the recursive call is to position the packets in each smaller subbutterfly in distinct inputs. On an M-input butterfly where at most c packets enter each input, M packets can be distributed to distinct inputs deterministically in $O(c + \log M)$ time.

The recursion continues until either the number of inputs, M, is smaller than $2^{\sqrt{\log N}}$, or the number of packets, n, is smaller than $\sqrt{M}$. In the first case, the sort is completed using Batcher's odd-even merge sort. An M-input butterfly can sort M packets in $O(\log^2 M)$ time using odd-even merge sort. For $M = 2^{\sqrt{\log N}}$, the time is $O(\log N)$. In the second case, the packets can be sorted deterministically in $O(\log M)$ time using Nassimi and Sahni's sorting algorithm, as in step four.

We can now make a rough estimate of the running time of this algorithm. Steps 1 and 2 are performed deterministically in $O(\log M)$ time. Assuming that there are $O(\sqrt{M})$ candidates, Steps 3, 4, and 5 also require $O(\log M)$ time. As we shall see, the expected time for Steps 6, 7 and 8 is $O(\log M)$. Although these steps sometimes take longer than expected, let us assume for now that they do not. In this case, the running time is given by the recurrence

$$T(M) \quad \leq \quad \begin{cases} T(2\sqrt{M}\log N) + O(\log M) & M > 2^{\sqrt{\log N}} \\ O(\log N) & M \leq 2^{\sqrt{\log N}} \end{cases}$$

which has solution $T(N) = O(\log N)$.

8.2 Analysis

The analysis of the algorithm is broken into three parts, each corresponding to a different use of randomization in the algorithm. We first examine the use of randomization in selecting the splitters. We show that, with high probability, the number of splitters chosen by each butterfly is within a constant factor of the expectation and the number of packets in each interval is smaller than the number of inputs of the subbutterfly to which it is assigned. Next, we bound the probability that the congestion is large at any particular switch in Steps 6 and 7. Finally, we show that if the packets are scheduled using the randomized algorithm for leveled networks, then it is unlikely that a delay of more than $O(\log N)$ will accumulate over the course of the algorithm.

8.3 Bounding the load

The first step in the analysis is to show that, with high probability, the number of splitter candidates chosen by each butterfly is within a constant factor of the expectation. We say that an M-input butterfly is *well-partitioned* if the number of splitter candidates chosen is between $\sqrt{M}/2$ and $3\sqrt{M}/2$. The $3\sqrt{M}/2$ upper bound ensures that the candidates can be sorted deterministically by the butterfly in $O(\log M)$ time and the $\sqrt{M}/2$ lower bound implies that the subbutterflies at the next level of recursion will have at most $2\sqrt{M} \log N$ inputs. If all of the butterflies are well-partitioned, then the algorithm terminates after $O(\log \log N)$ levels of recursion. (The choice of $1/2$ and $3/2$ as the coefficients of $\sqrt{M}$ are not particularly important. Other constants would serve equally well.)

Lemma 8.1 *For any fixed constant k_1 there is a constant k_2 such that the probability that any butterfly with at least $k_2 \log^2 N$ inputs is not well-partitioned is at most $1/N^{k_1}$.*

Proof: We begin by considering a single M-input butterfly that is to sort n packets. Since each packet chooses independently to be a candidate, the number of candidates has a binomial distribution. Let S be the number of successes in r independent Bernoulli trials where each trial has probability p of success. Then we have $\Pr[S = s] = \binom{r}{s} p^s (1-p)^{r-s}$. We estimate the area under the tails of this binomial distribution using a Chernoff-type bound [8]. Following Angluin and Valiant [3] we have

$$\begin{aligned}\Pr[S \leq \gamma_1 rp] &\leq e^{-(1-\gamma_1)^2 rp/2} \\ \Pr[S \geq \gamma_2 rp] &\leq e^{-(1-\gamma_2)^2 rp/3}\end{aligned}$$

for $0 \leq \gamma_1 \leq 1$ and $1 \leq \gamma_2 \leq 2$. In our application $r = n$, $p = \sqrt{M}/n$, $\gamma_1 = 1/2$, and $\gamma_2 = 3/2$. For any fixed constant k_3, there is a constant k_2 such that the right-hand sides of the two inequalities sum to at most $1/N^{k_3}$ for $M \geq k_2 \log^2 N$.

To bound the probability that any butterfly is not well-partitioned, we sum the probabilities for all of the individual butterflies. Over the course of the algorithm, the algorithm is invoked on at most $N \log N$ individual butterflies. Thus, the sum is at most $\log N/N^{k_3-1}$. For any k_1, there is a k_3 such that this sum is at most $1/N^{k_1}$. $\square$

The next lemma shows that, with high probability, the number of packets in each interval is at most a constant factor times its expectation. We say that an M-input butterfly that is assigned n packets to sort is δ*-split* if every interval has size at most $\delta n \log N/\sqrt{M}$. As we shall see, if every butterfly is $O(1)$-split and there are $O(\log \log N)$ levels of recursion, then by lightly loading the butterfly we can ensure that no butterfly is assigned too many packets to sort.

Lemma 8.2 *For any fixed constant k_1 there is a constant k_2 such that the probability that every butterfly is k_2-split is at least $1 - 1/N^{k_1}$.*

Proof: We begin by examining a single packet in a single M-input butterfly that is to sort n-packets. To show that a packet lies in an interval of size at most $k_2 n \log N/\sqrt{M}$ it is sufficient to show that both following and preceding it in the sorted order at least $\log N$ of the next $k_2 n \log N/2\sqrt{M}$ packets are candidates.

First we consider the packets that follow in the sorted order. The number of candidates in a sequence of $k_2 n \log N/2\sqrt{M}$ packets has a binomial distribution. For $r = k_2 n \log N/2\sqrt{M}$, $p = \sqrt{M}/n$, $rp = k_2 \log N/2$, $\gamma_1 = 2/k_2$, and $k_2 > 2$, we have $\Pr[S \leq \log N] \leq e^{-k_2(1-2/k_2)^2 \log N/4}$. For any k_3 we can make the right-hand side smaller than $1/N^{k_3}$ by choosing k_2 large enough.

The calculation for the packets that precede in the sorted order are identical. The probability that fewer $\log N$ of the preceding $k_2 n \log N/2\sqrt{M}$ packets are candidates is at most $1/N^{k_3}$. Thus, the probability that an individual packet lies in an interval of size greater than $k_2 n \log N/2\sqrt{M}$ is at most $2/N^{k_3}$.

To bound the probability that any interval in the butterfly is too large we sum the probabilities that each individual packet lies in an interval that is too large. Since there are at most $N \log N$ packets, this sum is at most $2 \log N/N^{k_3-1}$.

To bound the probability that any butterfly is not k_2-split, we sum the probabilities that each individual butterfly is not. Over the course of the algorithm, the algorithm is invoked on at most $N \log N$ butterflies. The sum of the probabilities is at most $2 \log^2 N/N^{k_3-2}$. For any constant k_1, we can make this sum at most $1/N^{k_1}$ by making k_3 large enough. $\square$

The remainder of the analysis is conditioned on the event that every subbutterfly is well-partitioned and $O(1)$-split, which occurs with high probability. Two technical points bear mentioning. First, Lemma 8.1 requires that the number of inputs to every subbutterfly be at least $k_2 \log^2 N$, where k_2 is some constant. Since the recursion terminates when the number of inputs is $2^{\sqrt{\log N}}$, N must be large enough that $2^{\sqrt{\log N}} > k_2 \log^2 N$. Second, both Lemmas 8.1 and 8.2 hold independent of the number of packets to be sorted by each subbutterfly. Thus, as the following lemmas show, we can adjust the load on the butterfly in order to ensure that each M-input subbutterfly receives at most M packets to sort.

Lemma 8.3 *If every subbutterfly is well-partitioned then the number of levels of recursion is $O(\log \log N)$.*

Proof: At each level of recursion the number of inputs drops from M to at most $2\sqrt{M}/\log N$, until the number of inputs reaches $2^{\sqrt{\log N}}$. $\square$

Lemma 8.4 *If every subbutterfly is well-partitioned and $O(1)$-split then there is an $\alpha > 0$ such that if the number of packets to be sorted is $N/\log^\alpha N$, then the number of packets are assigned to any M-input butterfly is at most M.*

Proof: Since the ratio of packets to inputs is $1/\log^\alpha N$ at the top level of the recursion, and increases by at most a constant factor at each of $O(\log \log N)$ levels, it is possible to choose α such that at the bottom level it will be at most one. $\square$

8.4 Bounding the congestion at each switch

The second step in the analysis is to bound the probability that too many packets pass through any switch in Steps 6 and 7. The following lemma provides a bound on the probability that the congestion, c, in an M-input butterfly exceeds $\log M$ in either of of these steps.

Lemma 8.5 *There is a fixed constant β_1 such that for $s > \log M$,*

$$\Pr[c \geq s] \leq \left(\frac{\beta_1}{s}\right)^s.$$

Proof: For the sake of brevity, we examine Step 7 only. A similar (and simpler) analysis holds for Step 6.

We begin by counting the number of packets that can possibly use a switch. Let L denote the depth of an M-input butterfly, i.e., $L = \log M$. From a switch at level l, $0 \leq l \leq L$, 2^{L-l} rows can be reached. The splitters partition these rows into subbutterflies. By Lemma 8.4, the number of packets that enter each of these subbutterflies is at most the number of inputs, with high probability. Thus, at most 2^{L-l} packets can pass through the switch.

Next we determine the probability that a packet that can pass through the switch actually does so. A switch at level l can be reached from 2^l different inputs. Since each packet begins in a random input, the probability that it can reach the switch is 2^{l-L}.

The number of packets, S, that pass through a particular switch at level l has a binomial distribution. The number of trials is $r = 2^{L-l}$ and the probability of success is $p = 2^{l-L}$. Thus, $\Pr[S = s] = \binom{2^{L-l}}{s}\left(2^{l-L}\right)^s\left(1 - 2^{l-L}\right)^{2^{L-l}-s}$ and $\Pr[S \geq s] \leq \binom{2^{L-l}}{s}\left(2^{l-L}\right)^s$. Using the inequality $\binom{a}{b} \leq (ae/b)^b$, we have $\Pr[S \geq s] \leq (e/s)^s$.

We bound the congestion in the entire butterfly by summing the individual probabilities over all $2^{O(L)}$ switches in the butterfly. We have

$$\Pr[c \geq s] \leq 2^{O(L)}\left(\frac{e}{s}\right)^s.$$

For $s \geq L$, we have $\Pr[c \geq s] \leq (\beta_1/s)^s$ for some constant β_1. $\square$

8.5 Bounding the cumulative delay

Since a subbutterfly does not begin to execute its algorithm until the larger butterfly at the previous level of recursion is finished, delay in excess of the time allotted to each butterfly accumulates over the course of the algorithm. An M-input butterfly is allotted $O(\log M)$ time to perform its steps. However, Steps 6, 7, and 8 are not guaranteed to terminate in time $O(\log M)$. It is tempting to try to prove that these steps terminate quickly with high probability. This approach fails because at the lower levels of the recursion the problem size is so small that nothing can be ascertained with high probability. Instead we must argue that although delay may occur at any particular step, it is unlikely that a lot of delay will accumulate over a sequence of steps.

The delay from Step 8 is relatively easy to analyze. This step requires $O(c+L)$ time; the delay depends only on the congestion. Lemma 8.5 bounds the probability that the congestion is large.

There are two possible causes of delay in Steps 6 and 7. A poor set of random rows for the packets can cause congestion at some node, which guarantees that some packet will arrive at its destination late. On the other hand, even if the congestion is small, a poor choice for the random ranks used by the scheduling algorithm may delay a packet. The following pair of lemmas bounds the probability that the delay from these steps is large. The first is a restatement of the Theorem 2.9 in a slightly different form. It bounds the probability that a packet will be delayed when the congestion is small. The second puts this bound together with the bound that the congestion is large from Lemma 8.5.

Lemma 8.6 *For a bounded-degree leveled network with L levels and a set of $N = 2^{O(L)}$ packets whose paths have congestion c, there are fixed constants $\beta_2 > 1$ and $\beta_3 > 1$ such that the probability that any packet arrives at its destination after time $\beta_2 L + w$, $w > 0$, is at most $(\beta_3 c/w)^w$.*

Proof: The proof of Theorem 2.9 shows that the probability that any packet arrives at its destination after step $L + w'$ is at most

$$2^{k_1 L + \log N} \cdot \left(\frac{k_2 c}{w'}\right)^{w'},$$

where k_1 and k_2 are constants. Suppose that $w' > k_1 L + \log N$. Then this probability is at most $\left(\frac{2k_2 c}{w'}\right)^{w'}$, which is less than $\left(\frac{2k_2 c}{w'-(k_1 L+\log N)}\right)^{w'-(k_1 L+\log N)}$.

Now let $w = w' - (k_1 L + \log N)$. Then the probability that any packet arrives at its destination after step $L + w' = \beta_2 L + w$ (for some β_2) is at most $\left(\frac{\beta_3 c}{w}\right)^w$, where $\beta_3 = 2k_2$. $\square$

Lemma 8.7 *There is a constant $\beta_4 > 1$ such that the probability that Steps 6 and 7 require more than $(\beta_2 + 1)L + w$ time steps, $w > 0$, is at most $2(1/\beta_4)^w$.*

Proof: For the sake of brevity, we examine Step 7 only. A similar analysis holds for Step 6.

We break the analysis into two cases according to whether the congestion is small or large. Let T be the time at which the last packet arrives. Then

$$\Pr[T \geq (\beta_2 + 1)L + w] \quad \leq \quad \Pr[T \geq (\beta_2 + 1)L + w | c < (L + w)/\beta_3\beta_4] + \Pr[c \geq (L + w)/\beta_3\beta_4].$$

We use Lemma 8.6 to bound the first term on the right. Plugging in $(L + w)/\beta_3\beta_4$ for c yields $\Pr[T \geq (\beta_2 + 1)L + w | c < (L + w)/\beta_3\beta_4] \leq (1/\beta_4)^w$. We use Lemma 8.5 to bound the second term on the right. Plugging in $(L+w)/\beta_3\beta_4$ for c yields $\Pr[c \geq (L+w)/\beta_3\beta_4] \leq (\beta_1\beta_3\beta_4/(L+w))^{(L+w)/\beta_3\beta_4}$. Since $L \geq \sqrt{\log N}$, and β_1, β_3, and β_4 are constants, $\Pr[c \geq (L+w)/\beta_3\beta_4] \leq (1/\beta_4)^w$, for sufficiently large N. $\square$

The following lemma bounds the combined delay of Steps 6, 7, 8.

Lemma 8.8 *There are constants β_5 and $\beta_6 > 1$ such that the probability that Steps 6, 7, and 8 together require time $\beta_5 L + w$, $w > 0$, is at most $(1/\beta_6)^w$.*

Proof: Step 8 can be performed deterministically in time $O(c + L)$. From Lemma 8.5 we have $\Pr[c \geq s] \leq (\beta_1/s)^s$, for $s > L$. For our purposes, a weaker bound on this probability suffices. Since β_1 is a constant, there is a constant k_1 such that $(1/k_1)^s \leq (\beta_1/s)^s$ for sufficiently large L. Combining this bound with that of Lemma 8.7 yields the desired result. $\square$

To complete our analysis of the algorithm, we need to bound the probability that more than $O(\log N)$ delay accrues during the sort.

Lemma 8.9 *For any fixed constant k_1, there is a constant k_2 such that the probability that the cumulative delay is more than $k_2 \log N$ is at most $1/N^{k_1}$.*

Proof: The cumulative delay at the bottom level of the recursion is the sum of the delay at each of the butterflies on the branch of the recursion tree from the top level to the leaf. Let D_i be the delay beyond $\beta_5 L$ at the ith level of the recursion. Then $\Pr[D_i = w] \le (1/\beta_6)^w$ by Lemma 8.8. Notice that there is no dependence on i in this expression. Let D be the cumulative delay on a branch of the recursion from the top level to a leaf. Then $D = \sum_{i=0}^{O(\log\log N)} D_i$. Generating functions help us here. The generating function for D_i is

$$G_{D_i}(z) = \sum_{w=0}^{\infty} \Pr[D_i = w] z^w,$$

where z^w can be thought of as a place holder. Since the delay at each level of the recusion is independent of the delays at other levels, we can sum the delay by multiplying the generating functions. Thus, the generating function for the cumulative delay is $G_D(z) = \prod_{i=0}^{O(\log\log N)} G_{D_i}(z)$. The coefficient of z^w in $G_D(z)$ is at most $\binom{w+O(\log\log N)}{w}(1/\beta_6)^w$. For $w = O(\log\log N)$, this coefficient is at most $(O(1)/\beta_6)^w$. For any k_3, there is a k_2 such that $\sum_{w=k_2\log N}^{\infty}(O(1)/\beta_6)^w$ is at most $1/N^{k_3}$.

To bound the probability that the cumulative delay exceeds $k_2 \log N$ on any branch of the recursion, we sum the individual probabilities for all of the branches. There are at most N branches. Thus, the sum is at most $1/N^{k_3-1}$. For any k_1, there is a k_3 such that this sum is at most $1/N^{k_1}$. □

8.6 Putting it all together

Theorem 8.10 *With high probability, an $N \log N$-node butterfly can sort $N \log N$ packets in $O(\log N)$ steps using constant-size queues.*

Proof: The algorithm for sorting $N \log N$ packets on an $N \log N$-node butterfly uses the algorithm for sorting $N/\log^{\alpha} N$ packets as a subroutine. First each packet independently chooses to be a splitter with probability $1/\log^{\alpha+1} N$. With high probability, this leaves $\Theta(N/\log^{\alpha} N)$ candidates. The candidates are sorted using the subroutine. Then every $\log N$th candidate is selected to be a splitter, leaving $\Theta(N/\log^{\alpha+1} N)$ splitters. The splitters are distributed throughout the butterfly, and splitter-directed routing is used to route intervals of size $O(\log^{\alpha+2} N)$ to subbutterflies with $\Theta(\log^{\alpha+1} N)$ inputs. Now each interval of $O(\log^{\alpha+2} N)$ packets resides in a group of $\Theta(\log^{\alpha+1} N)$ butterfly rows. Each of these rows contains $O(\log N)$ packets. The packets in each row can be sorted in $O(\log N)$ time using an odd-even transposition sort [17, Section 1.6.1]. With a fixed number of row sorts and permutations, all of the packets in each interval can be sorted in $O(\log N)$ time using Columnsort. $\square$

9 Sorting on shuffle-exchange graphs

The butterfly sorting result from Section 8 does not directly extend to the shuffle-exchange graph because the shuffle-exchange graph does not have the nice recursive structure possessed by the butterfly. However we can use the following theorem to sort on the shuffle-exchange graph. A similar theorem was proven independently by Raghunathan and Saran [28].

Theorem 9.1 *There is an embedding of $M/\log M$ distinct $M \log M$-node butterfly graphs in an $N = M^2$-node shuffle-exchange graph with load $l = 2$, congestion $c = O(1)$, and dilation $d = 3$.*

Proof: We assume that $M = 2^k$. Thus each row of each butterfly can be represented by a k-bit string, and each node of the shuffle-exchange can be represented by a $2k$-bit string.

To map $M/\log M$ butterflies to the shuffle-exchange graph, we use the following lemma [15, Lemma 1-1].

Lemma 9.2 *The set of $k = \log M$-bit strings has at least $M/2\log M$ disjoint subsets each containing $\log M$ distinct strings that are cyclic shifts of each other (for sufficiently large k).*

For each of these subsets we pick the lexicographically minimum string to represent the subset. We associate the $M/\log M$ butterflies in a two to one fashion with the $M/2\log M$ representative strings. Suppose butterfly i is associated with string w^i. We map a node $\langle l, r\rangle$ in butterfly i to a shuffle-exchange node by shuffling the bits of w^i with the bits of r's representation, and choosing the current bit to be r_l. That is, node $\langle l, r\rangle$ in butterfly i is mapped to shuffle-exchange node $r_1 w_1^i \ldots \underline{r_l} w_l^i \ldots r_k w_k^i$.

From a shuffle-exchange node we can recover the representative string w^i by picking out every other bit and shifting to the lexicographically minimum string. We find the row string by picking out the other bits and shifting by the same amount. The position in the row is clearly the number of shifts we used to get to w^i and the row number.

To finish, we observe that each edge in any of the butterflies is mapped to a path of length at most three in the shuffle-exchange graph since we either shift twice to reach $\langle l+1, r\rangle$'s image, or we exchange the current bit and shift twice to reach $\langle l+1, r_1 \ldots \overline{r_l} \ldots r_k\rangle$'s image.

Thus we can embed $\sqrt{N}/\log\sqrt{N}$ ($\sqrt{N}\log\sqrt{N}$)-node butterflies in an N-node shuffle-exchange with load 2, congestion $O(1)$, and dilation 3. $\Box$

This technique can be extended to prove that for any constant $0 < \epsilon < 1$, N^{ϵ} distinct $N^{1-\epsilon}$-node butterfly graphs can be embedded in an N-node shuffle-exchange with constant load, congestion, and dilation.

The following theorem shows that we can use Theorem 9.1 to sort on the shuffle-exhange graph in $O(\log N)$ steps.

Theorem 9.3 *With high probability, an N-node shuffle-exchange graph can sort N packets in $O(\log N)$ steps using constant-size queues.*

Proof: The N packets can be sorted using the Columnsort algorithm of [16]. Using Theorem 9.1, $M/\log M$ $M\log M$-node butterflies are embedded in the shuffle-exchange graph with constant load, congestion, and dilation, where $N = M^2$. Each of the butterflies can sort $M\log M$ packets in $O(\log M) = O(\log N)$ time using the butterfly sorting algorithm of Section 8. Columnsort sorts all N packets using a constant number of butterfly sorting steps and global permutation routing steps on the shuffle-exchange graph, each of which takes $O(\log N)$ time, with high probability. $\Box$

10 Sorting on multidimensional arrays

In this section we describe an algorithm for sorting kN packets on an N-node k-dimensional array with maximum side length M in $O(kM)$ steps. The algorithm closely follows the algorithm of Section 8 for sorting on the butterfly network. For the sake of brevity, we omit many of the details.

Theorem 10.1 *With high probability, an N-node k-dimensional array with maximum side length M can sort kN packets in $O(kM)$ steps using constant-size queues.*

Sketch of proof: The algorithm is essentially the same as the algorithm for sorting on the butterfly. For simplicity, we assume that $M = M_1 = \cdots = M_k$.

As before, there is a recursive subroutine for sorting $kN/2^{\Theta(\log k)}$ packets. We describe the action of the subroutine on a κ-dimensional subcube. (A κ-dimensional *subcube* of an array is a subgraph of the array consisting of M^κ nodes whose labels $(w_1, \ldots, w_k)$ share the same value in some set of $k-\kappa$ coordinates. For example, given a set $i_1, i_2, \ldots i_{k-\kappa}$ coordinates and a set $v_1, v_2, \ldots, v_{k-\kappa}$ of values, there is a subcube consisting of those nodes $(w_1, \ldots, w_k)$ such that $w_{i_j} = v_j$ for $1 \le j \le k-\kappa$.) The array is lightly loaded by the factor of $2^{\Theta(\log k)}$ to ensure that at each level of the recursion, the number of packets entering any κ-dimensional subcube is at most M^κ.

The first step is to choose a set of splitters. A set of $\Theta(M^{\kappa/2})$ packets are randomly chosen to be splitter candidates and then sorted by comparing every candidate with every other candidate, which can be done in $O(\kappa M)$ steps. Next, every $\log N$th candidate is selected to be a splitter.

The subcube routes packets on a leveled logical routing network as described in Section 6. The splitters are placed on the plateaus of the logical network, M^i of them on plateau i. Since there are only $\Theta(M^{\kappa/2})$ splitters, only the first $\beta\kappa$ plateaus get any, where β is some fixed constant less than one. On plateau 1, the jth splitter is assigned to all of the nodes labeled $(j, w_2, \ldots, w_\kappa)$, where $0 \le w_i \le M-1$ for $2 \le i \le \kappa$. A packet routes across dimension 1 edges on plateau 1 until until it reaches a splitter that is larger than itself. It then routes across dimension 2 edges to plateau 2. Because dimension 1 edges do not appear in any later plateaus, each of the M intervals is routed to a disjoint $k-1$ dimensional subgraph of the logical network. On each of these subgraphs, M splitters are positioned on plateau 2, and so on.

The subroutine calls itself recursively on subcubes with $\kappa - \beta\kappa$ dimensions. The recursion terminates when the number of packets entering a subcube is less than kM, in which case they are sorted sequentially in $O(kM)$ time. There are $O(\log k)$ levels to the recursion. As before, we can estimate the time required by the algorithm by assuming that the time spent on a κ-dimensional subcube is equal to the expectation, $O(\kappa M)$. The time is given by the recurrence

$$T(\kappa) \quad \le \quad \begin{cases} T(\kappa(1-1/\beta)) + O(\kappa M) & M^\kappa > kM \\ O(kM) & M^\kappa \le kM \end{cases}$$

which has solution $T(k) = O(kM)$. To rigorously bound the time we would have to argue that delay does accumulate across the levels of the recursion.

The algorithm for sorting kN packets calls the subroutine once. First, $kN/2^{\Theta(\log k)}$ packets are randomly chosen to be candidates and sorted using the subroutine. Next, every $\log N$th candidate is selected to be a splitter. The splitters partition the packets into intervals of size $2^{\Theta(\log k)} \log N$. Since $k \leq \log N$, each interval has at most $\log^{\Theta(1)} N$ packets. Since the 1-dimensional subcubes can each sort kM packets in $O(kM)$ steps using bubblesort, and $kM > \log N$, Columnsort can be applied to sort all of the packets in $O(kM)$ steps. □

11 Remarks

The scheduling algorithm from Section 2 can be used as a subroutine in algorithms for emulating shared-memory machines on bounded-degree networks. A shared-memory machine with a large address space can be emulated by randomly hashing the memory locations to the nodes of a butterfly as in [11] and [29]. The hashing ensures that the congestion of the packets implementing each memory access step is small. The algorithm from Section 2 is used to schedule the movements of these packets.

Given a set of N packets whose paths have congestion c on a leveled network with depth L, a setting of ranks that ensures delivery in time $O(c+L+\log N)$ can be found can be found off-line deterministically in time $2^{O(c+L+\log N)}$. The proof uses the Raghavan–Spencer technique [27, 33] to sequentially find a setting of the ranks so that no bad event corresponding to a delay sequence occurs.

One application is in preparing simulations by volume and area-universal networks off-line so that no random bits are needed. As before, the first step is to map the processors of the network to be simulated, B, to the processors of the area-universal network, U, from Section 7 using the recursive decomposition strategy from [21]. Network U has N processors, and B has area $O(N)$. To simulate each step of B, network U must route a set of $n = O(N/\log N)$ messages with load factor $\lambda = O(\log N)$. The second step is to find paths for the messages. Since these messages link the same processors at every step of B, it is sufficient to find paths once off-line. They can be reused over and over during the simulation. Given a set of n messages with load factor λ, it is possible to find a set of paths with congestion $c = O(\lambda + \log M)$ and dilation $d = O(\log M)$ in a fat-tree with root capacity M off-line deterministically in time polynomial in n and M. The final step is to find a set of ranks for the messages. These ranks can also be reused at each step of the simulation. Network U has root capacity $M = \Theta(\sqrt{N}/\log N)$. Thus, both the paths and the ranks for the packets can be determined off-line deterministically in time polynomial in N so that the time to simulate each step of B is $O(\log N)$.

By making minor modifications to the definition of a delay sequence, it is possible to prove that not only does the late arrival of some packet imply that a bad event occurs, but also if a bad event occurs then some packet is delayed. More precisely, some packet arrives at step $L + w$ where $w = m + qf$ if and only if a bad event corresponding to a delay sequence of length $l \leq L + 2f$ with $m + qf$ packets occurs.

References

[1] M. Ajtai, J. Komlós, and E. Szemerédi. Sorting in $c \log n$ parallel steps. *Combinatorica*, 3:1–19, 1983.

[2] R. Aleliunas. Randomized parallel communication. In *Proceedings of the ACM SIGACT–SIGOPS Symposium on Principles of Distributed Computing*, pages 60–72, August 1982.

[3] D. Angluin and L. G. Valiant. Fast probabilistic algorithms for hamiltonian circuits and matchings. *Journal of Computer and System Sciences*, 18(2):155–193, April 1979.

[4] K. Batcher. Sorting networks and their applications. In *Proceedings of the AFIPS Spring Joint Computing Conference*, volume 32, pages 307–314, 1968.

[5] Butterfly$^{\text{TM}}$ Parallel Processor Overview. BBN Report No. 6148, Version 1, BBN Advanced Computers, Inc., Cambridge, MA, March 1986.

[6] V. E. Beneš. *Mathematical Theory of Connecting Networks and Telephone Traffic*. Academic Press, New York, 1965.

[7] J. L. Carter and M. N. Wegman. Universal classes of hash functions. *Journal of Computer and System Sciences*, 18(2):143–154, April 1979.

[8] H. Chernoff. A measure of asymptotic efficiency for tests of a hypothesis based on the sum of observations. *American Mathematical Society*, 23:493–507, 1952.

[9] W. Dally and C. Seitz. Deadlock free message routing in multiprocessor interconnection networks. *IEEE Transactions on Computers*, C–36(5):547–553, May 1987.

[10] R. I. Greenberg and C. E. Leiserson. Randomized routing on fat-trees. In Silvio Micali, editor, *Randomness and Computation.* Volume 5 of *Advances in Computing Research*, pages 345–374. JAI Press, Greenwich, CT, 1989.

[11] A. R. Karlin and E. Upfal. Parallel hashing — an efficient implementation of shared memory. In *Proceedings of the 18th Annual ACM Symposium on Theory of Computing*, pages 160–168, May 1986.

[12] R. R. Koch. Increasing the size of a network by a constant factor can increase performance by more than a constant factor. In *Proceedings of the 29th Annual Symposium on Foundations of Computer Science*, pages 221–230. IEEE Computer Society Press, October 1988.

[13] D. Krizanc, S. Rajasekaran, and Th. Tsantilis. Optimal routing algorithms for mesh-connected processor arrays. In J. Reif, editor, *Aegean Workshop on Computing: VLSI Algorithms and Architectures.* Volume 319 of *Lecture Notes in Computer Science*, pages 411–422. Springer–Verlag, New York, NY, June 1988.

[14] M. Kunde. Routing and sorting on mesh-connected arrays. In J. Reif, editor, *Aegean Workshop on Computing: VLSI Algorithms and Architectures.* Volume 319 of *Lecture Notes in Computer Science*, pages 423–433. Springer–Verlag, New York, NY, June 1988.

[15] F. T. Leighton. *Complexity Issues in VLSI.* MIT Press, Cambridge, MA, 1983.

[16] F. T. Leighton. Tight bounds on the complexity of parallel sorting. *IEEE Transactions on Computers*, C–34(4):344–354, April 1985.

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

[18] F. T. Leighton and B. M. Maggs. Fast algorithms for routing around faults in multibutterflies and randomly-wired splitter networks. *IEEE Transactions on Computers*, 41(5):578–587, May 1992.

[19] F. T. Leighton, F. Makedon, and I. Tollis. A $2N-2$ step algorithm for routing in an $N \times N$ mesh. In *Proceedings of the 1989 ACM Symposium on Parallel Algorithms and Architectures*, pages 328–335, June 1989.

[20] T. Leighton, B. Maggs, and S. Rao. Universal packet routing algorithms. In *Proceedings of the 29th Annual Symposium on Foundations of Computer Science*, pages 256–271, October 1988.

[21] C. E. Leiserson. Fat-trees: universal networks for hardware-efficient supercomputing. *IEEE Transactions on Computers*, C–34(10):892–901, October 1985.

[22] C. E. Leiserson and B. M. Maggs. Communication-efficient parallel graph algorithms for distributed random-access machines. *Algorithmica*, 3(1):53–77, 1988.

[23] B. M. Maggs. *Locality in Parallel Computation.* PhD thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, Cambridge, MA, September 1989.

[24] R. Miller, V. K. Prasanna-Kumar, D. Reisis, and Q. F. Stout. Meshes with reconfigurable buses. In J. Allen and F. T. Leighton, editors, *Advanced Research in VLSI: Proceedings of the Fifth MIT Conference*, pages 163–178, Cambridge, MA, April 1988. MIT Press.

[25] D. Nassimi and S. Sahni. Parallel permutation and sorting algorithms and a new generalized connection network. *Journal of the ACM*, 29(3):642–667, July 1982.

[26] N. Pippenger. Parallel communication with limited buffers. In *Proceedings of the 25th Annual Symposium on Foundations of Computer Science*, pages 127–136, October 1984.

[27] P. Raghavan. Probabilistic construction of deterministic algorithms: approximate packing integer programs. *Journal of Computer and System Sciences*, 37(4):130–143, October 1988.

[28] A. Raghunathan and H. Saran. Is the shuffle-exchange better than the butterfly? Unpublished manuscript.

[29] A. G. Ranade. How to emulate shared memory. In *Proceedings of the 28th Annual Symposium on Foundations of Computer Science*, pages 185–194. IEEE Computer Society Press, October 1987.

[30] A. G. Ranade. *Fluent Parallel Computation.* PhD thesis, Yale University, New Haven, CT, 1988.

[31] A. G. Ranade, S. N. Bhatt, and S. L. Johnsson. The fluent abstract machine. In J. Allen and F. T. Leighton, editors, *Advanced Research in VLSI: Proceedings of the Fifth MIT Conference*, pages 71–94, Cambridge, MA, April 1988. MIT Press.

[32] J. H. Reif and L. G. Valiant. A logarithmic time sort for linear size networks. *Journal of the ACM*, 34(1):60–76, January 1987.

[33] J. Spencer. *Ten Lectures on the Probabilistic Method*. SIAM, Philadelphia, PA, 1987.

[34] C. D. Thompson. *A Complexity Theory for VLSI*. PhD thesis, Department of Computer Science, Carnegie-Mellon University, Pittsburgh, PA, 1980.

[35] C. D. Thompson and H. T. Kung. Sorting on a mesh-connected parallel computer. *Communications of the ACM*, 20(4):263–271, 1977.

[36] E. Upfal. Efficient schemes for parallel communication. In *Proceedings of the ACM SIGACT–SIGOPS Symposium on Principles of Distributed Computing*, pages 55–59, August 1982.

[37] E. Upfal. An $O(\log N)$ deterministic packet routing scheme. In *Proceedings of the 21st Annual ACM Symposium on Theory of Computing*, pages 241–250, May 1989.

[38] L. G. Valiant. A scheme for fast parallel communication. *SIAM Journal on Computing*, 11(2):350–361, May 1982.

[39] 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.

[40] A. Waksman. A permutation network. *Journal of the ACM*, 15(1):159–163, January 1968.