ON-LINE ALGORITHMS FOR PATH SELECTION IN A NONBLOCKING NETWORK*

SANJEEV ARORA †, F. THOMSON LEIGHTON ‡, AND BRUCE M. MAGGS §

Abstract.
This paper presents the first optimal-time algorithms for path selection in an optimal-size nonblocking network. In particular, we describe an N-input, N-output, nonblocking network with $O(N \log N)$ bounded-degree nodes, and an algorithm that can satisfy any request for a connection or disconnection between an input and an output in $O(\log N)$ bit steps, even if many requests are made at once. Viewed in a telephone switching context, the algorithm can put through any set of calls among N parties in $O(\log N)$ bit steps, even if many calls are placed simultaneously. Parties can hang up and call again whenever they like; every call is still put through $O(\log N)$ bit steps after being placed. Viewed in a distributed memory machine context, our algorithm allows any processor to access any idle block of memory within $O(\log N)$ bit steps, no matter what other connections have been made previously or are being made simultaneously.

Key words. nonblocking network, multibutterfly network, multi-Beneš network, routing algorithm

AMS subject classifications. 68M10, 90B12, 94C10

1. Introduction.

1.1. Definitions. Nonblocking networks arise in a variety of communications contexts. Common examples include telephone systems and network architectures for parallel computers. In a typical application, there are $2N$ *terminals* (usually thought of as N *inputs* and N *outputs*) interconnected by switches that can be set to link the inputs to the outputs with node-disjoint paths according to a specified permutation. (Switches are also called nodes.) In a *nonblocking* network, the terminals and nodes are interconnected in such a way that any unused input–output pair can be connected by a path through unused nodes, no matter what other paths exist at the time. The 6-terminal graph shown in Figure 1.1, with inputs Bob, Ted, and Pat and outputs Vanna, Carol, and Alice, for example, is nonblocking because no matter which input–output pairs are connected by a path, there is a node-disjoint path linking any unused input–output pair. In particular, if Bob is talking to Alice and Ted is talking to Carol, then Pat can still call Vanna.

The notion of a nonblocking network has several variations. The nonblocking network in Figure 1.1 is an example of the most commonly studied type. This network is called a *strict-sense nonblocking connector*, because no matter what paths are established in the network, it is possible to establish a path from any unused input to any unused output. A slightly weaker notion is that of a *wide-sense nonblocking*

*This research was conducted while the first and third authors were affiliated with the Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, MA 02139. It was supported in part by the Defense Advanced Research Projects Agency under Contracts N00014–87–K–825 and N00014–89–J–1988, the Air Force under Contract AFOSR–89–0271, and the Army under Contract DAAL–03–86–K-0171. A preliminary version of this paper appeared in the *Proceedings of the 22nd Annual ACM Symposium on Theory of Computing,* pages 149–158, May 1990.

†Department of Computer Science, Princeton University, Princeton, NJ 08544. (arora@cs.princeton.edu).

‡Mathematics Department and Laboratory for Computer Science, Massachusetts Institute of Technology,Cambrige, MA 02139. (ftl@math.mit.edu).

§School of Computer Science, Carnegie Mellon University, Pittsburgh, PA 15213. (bmm@cs.cmu.edu).

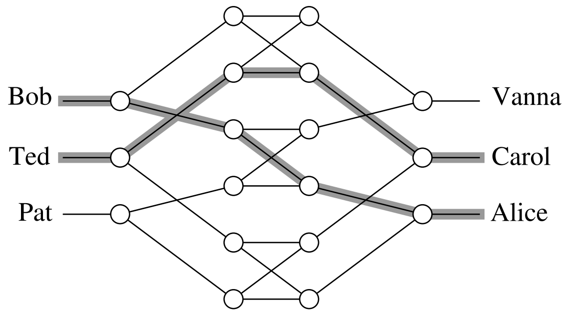

FIG. 1.1. *A nonblocking network with 3 inputs and 3 outputs.*

connector. A wide-sense nonblocking connector does not make the same guarantee as a strict-sense nonblocking connector. A network is a wide-sense nonblocking connector if there is an algorithm for establishing paths in the network one after another so that after each path is established, it is still possible to connect any unused input to any unused output. Still weaker is the notion of a *rearrangeable connector*. A rearrangeable connector is capable of realizing any 1–1 connection of inputs to outputs with node-disjoint paths provided that all the connections to be made are known in advance. A nonblocking or rearrangeable connector is a called a *generalized connector* if it has the additional property that each input can be simultaneously connected to an arbitrary set of outputs, provided that every output is connected to just one input. Generalized connectors are useful for multiparty calling in a telephone network as well as for broadcasting in a parallel machine.

1.2. Previous work. Nonblocking and rearrangeable networks have a rich and lengthy history. See [30] for an excellent survey and [9, 10] for more comprehensive descriptions of previous results. In 1950, Shannon [35] proved that any rearrangeable or nonblocking connector with N-inputs and N-outputs must have $\Omega(N \log N)$ edges[1]. Further work on lower bounds can be found in [4, 11, 32, 33]. In 1953, Clos constructed a strict-sense nonblocking connector with $O(N^{1+1/j})$ edges and depth j, for fixed j. (The degree of the nodes is not bounded). Bounded-depth nonblocking networks have subsequently been studied extensively [8, 10, 24, 25, 29, 33]. In the early 1960's, Beizer [5] and Beneš [6] independently discovered bounded-degree rearrangeable connectors with depth $O(\log N)$ and size $O(N \log N)$, and Waksman [38] gave an elegant algorithm for determining how the nodes should be set in order to realize any particular permutation. Ofman [26] followed with a generalized rearrangeable connector of size $O(N \log N)$. Next, Cantor [7] discovered a bounded-degree $O(\log N)$-depth strict-sense nonblocking connector with $O(N \log^2 N)$ edges. The existence of a bounded-degree strict-sense nonblocking connector with size $O(N \log N)$ and depth $O(\log N)$ was first proved by Bassalygo and Pinsker [3]. Although the Bassalygo and Pinsker proof is not constructive, subsequent work on the explicit construction of expanders [23] yielded a construction.

More recent work has focused on the construction of generalized nonblocking connectors. In 1973, Pippenger [28] constructed a wide-sense generalized nonblocking connector with $O(N \log^2 N)$ edges. This result was later improved to $O(N \log N)$ edges by Feldman, Friedman, and Pippenger [10]. Recently, Turner suggested cascad-

[1]Throughout this paper $\log N$ denotes $\log_2 N$.

ing two of the asymptotically larger Clos or Cantor networks as a more practical way to construct a generalized nonblocking connector [36]. This method requires that all the parties in a multiparty call are known at the time that the call is placed.

Unfortunately, there has not been as much progress on the problem of setting the nodes to realize the connection paths. Indeed, several of the references cited previously show that there exists a way of setting the nodes to realize the desired paths, but are unable to provide any reasonable algorithms for actually finding the right node settings. For example, no polynomial time algorithm is known for finding the paths in the wide-sense generalized nonblocking connector of [10]. There are a few exceptions. On the naive nonblocking networks of size $\Theta(N^2)$ (e.g. an $N \times N$ mesh of trees [15]), a simple greedy algorithm suffices to find the paths on-line in $O(\log N)$ time. (An algorithm that finds the settings for the nodes is called a *circuit switching* algorithm. An algorithm that is performed by the nodes themselves using only local information is called an *on-line* algorithm; an *off-line* algorithm is one that uses more global information.) Also, Lin and Pippenger recently found polylogarithmic time off-line parallel algorithms for path selection in $O(N \log^2 N)$-size strict-sense nonblocking connectors using one processor per request [22]. On any strict-sense nonblocking connector, an on-line version of breadth-first search can be used to find a path from an unused input to an unused output on-line. Unfortunately, this algorithm cannot efficiently cope with simultaneous requests for connections. Nevertheless, no better algorithm, either on-line or off-line, was previously known for any $O(N \log N)$-size nonblocking network.

1.3. Models and conventions. The running times of the algorithms in this paper are described in two models, the *bit model* and the *word model*. In the *bit model*, each network node can be thought of as a finite automaton. In each *bit step*, the node can receive a single bit of information along each of its incoming edges (of which there are at most a constant number), change to a new state, and output a single bit of information on each of its outgoing edges (of which there are at most a constant number). In the word model, each edge in an N-node network can transmit a word consisting of up to $O(\log N)$ bits in a single step.

To simplify the explanation of the algorithms and results in this paper, we have adopted some conventions that may differ from the way that this material is treated in the more applied literature. For example, we generally route paths in a node-disjoint fashion. In practice, however, it may be desirable to route paths in an edge-disjoint manner instead. Our definitions and results can also be applied in this setting, as demonstrated in Section 5.3.3. Note that node-disjoint paths are automatically edge-disjoint, and any algorithm for routing edge-disjoint paths on a degree-d network can be converted into one for routing node-disjoint paths by replacing each node with a $d \times d$ complete bipartite graph.

1.4. Our results. In this paper, we describe an $O(N \log N)$-node nonblocking network for which each connection can be made on-line in $O(\log N)$ bit steps. The path selection algorithm works even if many calls are made at once — every call still gets through in $O(\log N)$ bit steps, no matter what calls were made previously and no matter what calls are currently active, provided that no two inputs try to access the same output at the same time. (If many inputs inadvertently try to access the same output at the same time, all but one of the inputs will receive a busy signal. The busy signals are also returned in $O(\log N)$ bit steps, but, at present, we require the use of a sorting circuit [2, 20] to generate the busy signals. Alternatively, we could merge the calling parties together, but this also requires the use of a sorting circuit.)

In all scenarios, the size of the network and the speed of the path selection algorithm are asymptotically optimal.

In addition to providing the first optimal solution to the abstract telephone switching problem, our results significantly improve upon previously known algorithms for bit-serial packet routing. Previously, $O(\log N)$-bit-step algorithms for packet routing were known only for the special case in which all packet paths are created or destroyed at the same time, and even then only by resorting to the AKS sorting circuit [2], or by using randomness on the hypercube [1]. In many circuit-switched parallel machines, however, packets are of varying lengths and packet paths are created and destroyed at arbitrary times, thereby requiring that paths be routed in a nonblocking fashion – which is something that previously discovered algorithms were not capable of doing. Even without worrying about the nonblocking property, our results provide the first non-AKS $O(\log N)$-bit-step algorithms for bit-serial packet routing on a bounded-degree network. (Since this work first appeared, Leighton and Plaxton have developed an $O(\log N)$-bit-step randomized sorting algorithm for the butterfly [20].)

1.5. Our approach. The networks that we use to obtain these results are constructed by combining expanders and Beneš networks in much the same way that expanders and butterflies are combined to form the multibutterfly networks described by Upfal [37]. We refer to these networks as *multi-Beneš networks*. The nonblocking networks of Bassalygo and Pinsker [3] are similar. The details of the construction are provided in Section 2 of the paper.

The techniques in this paper can also be applied to bandwidth-limited switching networks such as fat-trees [21]. These networks may be more useful in the context of real telephone systems, where there are limitations on the number of calls based on the proximity of the calls (e.g., it is unlikely that everyone on the East Coast will call everyone on the West Coast at the same time).

The description and analysis of the path selection algorithm is divided into three sections. In Section 3, we prove that the multi-Beneš network is a strict-sense nonblocking connector. A similar approach was used in [17] to show that the multibutterfly is capable of routing in the presence of many faulty nodes. Indeed, we can think of currently-used nodes as being faulty since they cannot be used to form new connections. Similarly, the algorithms we describe for routing in nonblocking networks can easily be extended to be highly tolerant to faults in the network. In Section 4, we describe an $O(\log N)$-bit-step algorithm for bit-serial routing in a multibutterfly. This algorithm relies on an unshared-neighbor property possessed by all highly-expanding graphs. By implementing this algorithm on the multi-Beneš network and combining it with the methods of Section 3, we produce an algorithm that can handle many calls at the same time, independent of what calls have been made previously and what calls are currently connected.

In Section 5, we describe algorithms for handling multiparty calls, and situations where many inputs try to reach the same output simultaneously. Some of these algorithms rely on sorting circuits and are not as practical as those described in Section 4. We also show how to remove the distinction between terminals and non-terminals.

2. The multi-Beneš and multibutterfly networks. Our nonblocking network is constructed from a Beneš network in much the same way that a multibutterfly network [37] is constructed from a butterfly network. We start by describing the butterfly, Beneš, and multibutterfly networks.

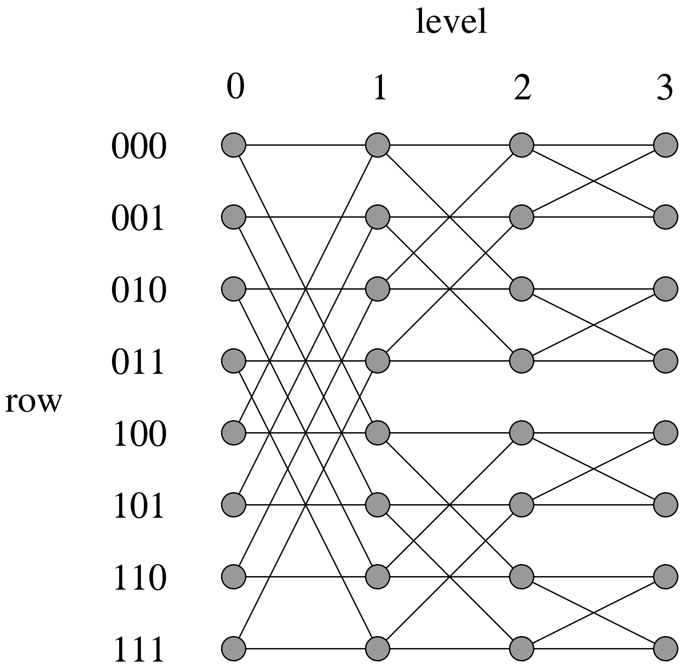

FIG. 2.1. *An 8-input butterfly network.*

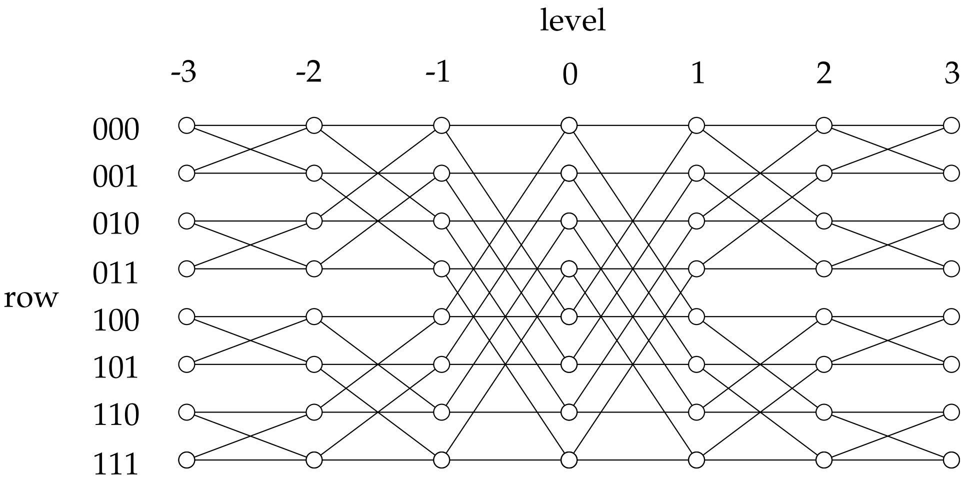

FIG. 2.2. *An 8-input Beneš network.*

An N-input butterfly has $\log N + 1$ levels, each with N-nodes. An example is shown in Figure 2.1. The Beneš network is a $(2 \log N + 1)$-level network consisting of back-to-back butterflies. The network in Figure 2.2 is a Beneš network. Although Beneš networks are usually drawn with the long diagonal edges at the first and last levels rather than in the middle (see e.g., [16, Figure 3-27]), the networks are isomorphic.

A multibutterfly is formed by gluing together butterflies in a somewhat unusual

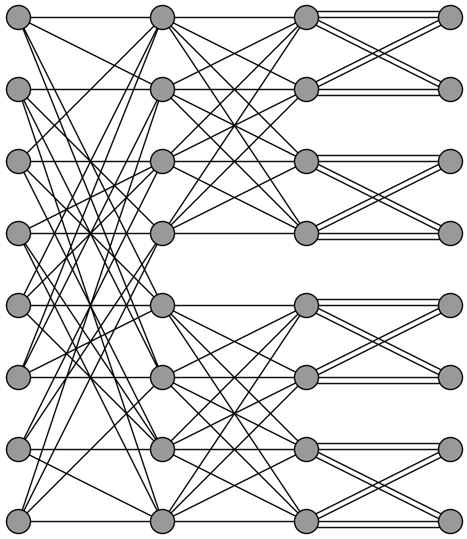

FIG. 2.3. *An 8-input 2-butterfly network.*

way. In particular, given 2 N-input butterflies G_1 and G_2 and a collection of permutations $\Pi = \langle \pi_0, \pi_1, \ldots, \pi_{\log N} \rangle$ where $\pi_l : [0, \frac{N}{2^l} - 1] \to [0, \frac{N}{2^l} - 1]$, a 2-butterfly is formed by merging the node in row $\frac{jN}{2^l} + i$ of level l of G_1 with the node in row $\frac{jN}{2^l} + \pi_l(i)$ of level l of G_2 for all $0 \le i \le \frac{N}{2^l} - 1$, all $0 \le j \le 2^l - 1$, and all $0 \le l \le \log N$. The result is an N-input $(\log N + 1)$-level graph in which each node has 4 inputs and 4 outputs. Of the 4 output edges at a node, two are *up* outputs and two are *down* outputs (with one up edge and one down edge coming from each butterfly). For example, see Figure 2.3. Multibutterflies (i.e., d-butterflies) are composed from d butterflies in a similar fashion using $d-1$ sets of permutations, $\Pi^{(1)}, \ldots, \Pi^{(d-1)}$, where $\Pi^{(i)} = \{\pi_l^{(i)}, 0 \le l \le \log N\}$, resulting in a $(\log N + 1)$-level network with $2d \times 2d$ nodes.

In a butterfly or multibutterfly, for each output v there is a distinct *logical* (up-down) path from the inputs to v. In order to reach v from any input u, the path from u to v must take an up-edge from level l to level $l+1$ if the lth bit in the row number of v is 0, and a down-edge if the bit is 1. (The bits are counted starting with the most significant, which is in position 0). Figure 2.4 shows the logical path from any input to output 011. Let us use the term *physical path* to denote our usual notion of a path through the network, i.e., a physical path consists of a sequence of nodes $w_0, w_1, \ldots, w_{\log N}$ such that node w_i resides on level i of the network, and nodes w_i and w_{i+1} are connected by an edge, for $0 \le i < \log N$. In a butterfly network, the logical path can be realized by only one physical path through the network. In a multibutterfly, however, each step of the logical path can be taken on any one of d edges. Hence, for any logical path there are many physical paths through the network.

The notion of up and down edges can be formalized in terms of splitters. More precisely, the edges from level l to level $l+1$ in rows $\frac{jN}{2^l}$ to $\frac{(j+1)N}{2^l} - 1$ in a multibutterfly form a *splitter* for all $0 \le l < \log N$ and $0 \le j \le 2^l - 1$. Each of the 2^l splitters starting at level l has $\frac{N}{2^l}$ inputs $\frac{N}{2^l}$ and outputs. The outputs on level $l+1$ are naturally divided into $\frac{N}{2^{(l+1)}}$ *up* outputs and $\frac{N}{2^{(l+1)}}$ *down* outputs. By definition, all splitters on the same level l are isomorphic, and each input is connected to d up outputs and d down outputs

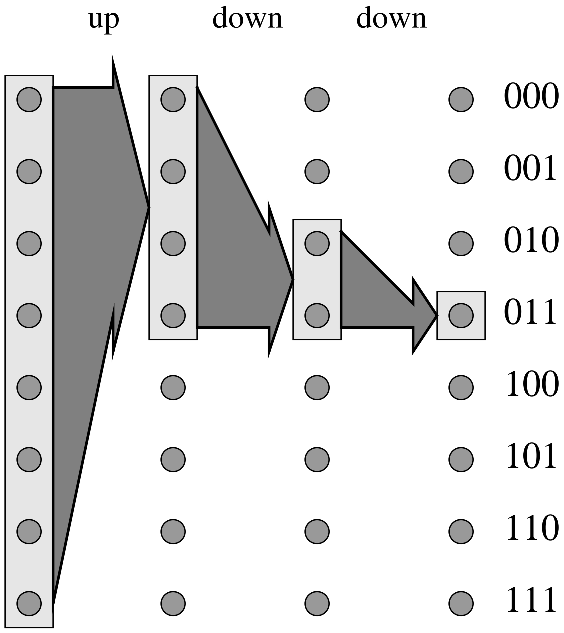

FIG. 2.4. *The logical up-down path from an input to output 011.*

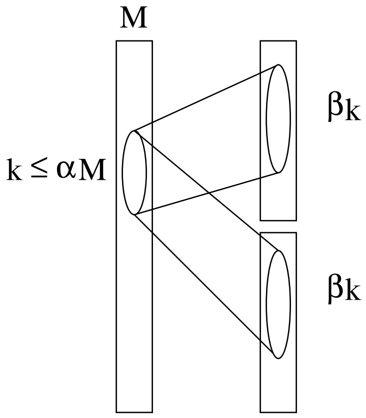

FIG. 2.5. *A splitter with expansion property* (α, β).

according to the butterfly and the permutations $\pi_l^{(1)}, \ldots, \pi_l^{(d-1)}$ and $\pi_{l+1}^{(1)}, \ldots, \pi_{l+1}^{(d-1)}$.

The most important characteristic of a multibutterfly is the set of permutations $\Pi^{(1)}$, ..., $\Pi^{(d-1)}$ that prescribe the way in which the component butterflies are to be merged. For example, if all the permutations are the identity map, then the result is the *dilated butterfly* (i.e., a butterfly with d copies of each edge). We are most interested in multibutterflies that have expansion properties. In particular, we say that an M-input splitter has *expansion property* (α, β) if every set of $k \leq \alpha M$ inputs is connected to at least βk up outputs and βk down outputs for $\beta > 1$. Similarly, we say that a multibutterfly has *expansion property* (α, β) if each of its component splitters has expansion property (α, β). For example, see Figure 2.5.

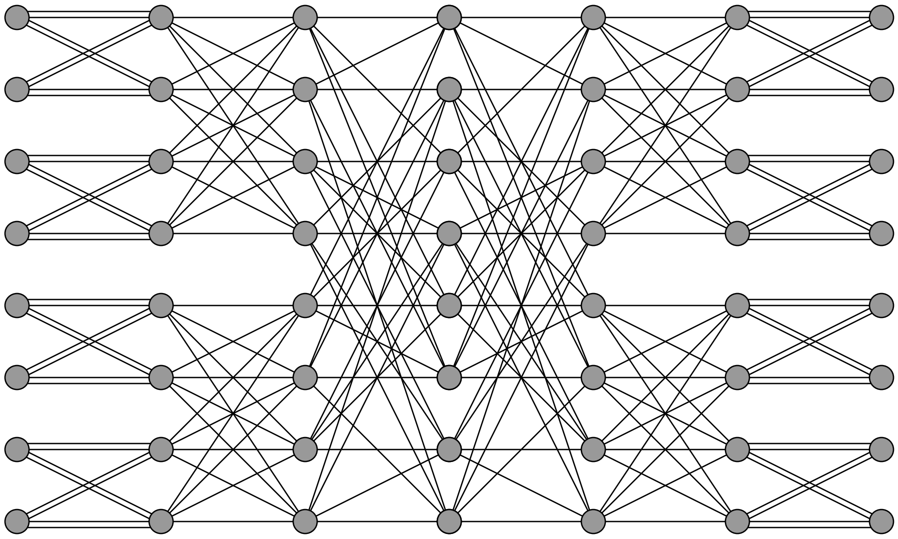

FIG. 2.6. *An 8-input 2-multi-Beneš network.*

Although the constants α, β, and d do not appear in the expressions for the running times of our algorithms, e.g., $O(\log N)$, as a practical matter they are crucial. In general, the larger β is, the fewer bit steps an algorithm will require. However, since $d \geq \beta$, a network with large β must also have large d, and in practice it may be difficult to build a node that can receive and transmit along all d of its edges simultaneously if d is large. Furthermore, most of the algorithms require $\beta > d/2$, which (as far as we know) can only be achieved for small α. As we shall see, the fraction of network nodes that are actually used by paths is at most $1/\alpha$, so if α is small, the network is not fully utilized.

If the permutations $\Pi^{(1)}, \ldots, \Pi^{(d-1)}$ are chosen randomly, then with non-zero probability, the resulting d-butterfly has expansion property (α, β) for any d, α, and β for which $2\alpha\beta < 1$ and

$$d > \beta + 1 + \frac{\beta + 1 + \ln 2\beta}{\ln(\frac{1}{2\alpha\beta})}. \tag{2.1}$$

This bound appears as Corollary 2.1 in [37]. A derivation can be found in [18]. Roughly speaking, the bound says that the expansion, β, can be almost as large as $d - 1$, provided that α is small enough. Furthermore, for any α, β can be made arbitrarily close to $1/2\alpha$, by making d large. It is not known if β can be made close to both $d - 1$ and $1/2\alpha$ simultaneously. Constructions for splitters and multibutterflies with good expansion properties are known although the expansion properties are generally not as good as those obtained from randomly-generated graphs.

Like a multibutterfly, a multi-Beneš network is formed from Beneš networks by merging them together. A 2-multi-Beneš network is shown in Figure 2.6. An N-input multi-Beneš network has $2\log N + 1$ levels labeled $-\log N$ through $\log N$. Levels 0 through $\log N$ form a multibutterfly, while levels $-\log N$ through 0 form the mirror image of a multibutterfly.

As in the multibutterfly, the edges in levels 0 through $\log N$ are partitioned into splitters. Between levels $-\log N$ and 0, however, the edges are partitioned into *mergers*. More precisely, the edges from level l to level $l+1$ in rows $j2^{l+\log N+1}$ to $(j+1)2^{l+\log N+1}-1$ form a *merger* for all $-\log N \leq l < 0$ and $0 \leq j \leq N/2^{l+\log N+1}-1$. Each of the $N/2^{l+\log N+1}$ mergers starting at level l has $2^{l+\log N+1}$ inputs and outputs. The inputs on level l are naturally divided into $2^{l+\log N}$ *up* inputs and $2^{l+\log N}$ *down* inputs. All mergers on the same level l are isomorphic, and each input is connected to $2d$ outputs. There is a single, trivial, logical path from any input of a multi-Beneš network through the mergers on levels $-\log N$ through -1 to the single splitter on level 0. (Any physical path will do.) From level 0 there is a single logical up-down path through the splitters to any output on level $\log N$. In both cases, the logical path can be realized by many physical paths.

We say that an M-output merger has expansion property (α, β) if every set of $k \leq \alpha M$ inputs (up or down or any combination) is connected to at least $2\beta k$ outputs, $\beta > 1$. With nonzero probability, a random set of permutations yields a merger with expansion property (α, β) for any d, α, and β for which $\alpha\beta < 1/2$ and

$$2d > 2\beta + 1 + \frac{2\beta + 1 + \ln 2\beta}{\ln(\frac{1}{2\alpha\beta})}. \tag{2.2}$$

This inequality can be derived by making a small number of substitutions in the derivation of Inequality 2.1 found in [18]. We say that a multi-Beneš network has expansion property (α, β) if each of its component mergers and splitters has expansion property (α, β). The multibutterflies and multi-Beneš networks considered throughout this paper are assumed to have expansion property (α, β).

It is worth noting that all the results in this paper hold for a broader class of networks than multibutterflies and multi-Beneš networks. In particular, each basic butterfly component used to make a multibutterfly or multi-Beneš network can be replaced by any Delta network. A *Delta* network is a regular network formed by splitters like the butterfly, but for which the individual connections within each splitter can be arbitrary [14].

3. A proof that the multi-Beneš network is nonblocking. In this section we prove that the multi-Beneš network is a strict-sense nonblocking connector. As a consequence, a simple algorithm like breadth-first search can be used to establish a single path from any unused input to any unused output in $O(\log N)$ bit steps, where N is the number of rows. Algorithms that handle simultaneous requests for connections and multiparty calls are deferred to Sections 4 and 5.

In order for the algorithm to succeed, the multi-Beneš network must be "lightly loaded" by some fixed constant factor L, where we will choose L to be a power of 2. Thus, in an N-row multi-Beneš network, we only make connections between the N/L inputs and outputs in rows that are multiples of L. Since the other inputs and outputs are not used, the first and last $\log L$ levels of the network can be removed, and the N/L inputs and outputs can each be connected directly to their L descendants and ancestors on levels $-\log N + \log L$ and $\log N - \log L$, respectively.

The basic idea is to treat the nodes through which paths have already been established as if they were faulty and to apply the fault propagation techniques from [17] to the network. In particular, we define a node to be *busy* if there is a path currently routing through it. We recursively define a node in the second half of the network to be *blocked* if all of its up outputs or all of its down outputs are busy or blocked. More precisely, nodes are declared to be blocked according to the following rule. Working backwards from level $\log N - \log L - 1$ to level 0, a node is declared blocked if either all d of its up edges or all d of its down edges lead to busy or blocked nodes. From level -1 to level $-\log N + \log L$, a node is declared blocked if all $2d$ of its outgoing edges lead to busy or blocked nodes. A node that is neither busy nor blocked is said to be *working*.

The following pair of lemmas bound the fraction of input nodes that are blocked in every splitter and merger.

LEMMA 3.1. *For $L > 1/2\alpha(\beta - 1)$, at most a 2α fraction of the inputs in any splitter are declared to be blocked. Furthermore, at most an α fraction of the nodes are blocked because of busy and blocked nodes from the upper outputs, and at most an α fraction are blocked because of busy and blocked nodes from the lower outputs.*

Proof. The proof is by induction on level number, starting at level $\log N - \log L$ and working backwards to level 0. The base case is trivial since there are no blocked nodes on level $\log N - \log L$. Suppose the inputs of an M-input splitter contain more than αM nodes that are blocked because of the upper (say) outputs. Consider the set U of busy or blocked upper outputs. Since all of the edges out of a blocked input lead to busy or blocked outputs, we can conclude that $|U| \geq \alpha\beta M$. Since every path passing through the upper outputs must lead to one of $M/2L$ terminals, there can be at most $M/2L$ busy nodes among the upper outputs of the splitter. Furthermore, by induction there are at most αM blocked nodes among the upper outputs. Thus, $|U| \leq \alpha M + M/2L$. For $L > 1/2\alpha(\beta - 1)$ we have a contradiction. Hence, at most an α fraction of the nodes are blocked, as claimed. □

LEMMA 3.2. *For $L > 1/2\alpha(\beta - 1)$, at most a 2α fraction of the upper inputs and a 2α fraction of the lower inputs in any merger are blocked.*

Proof. The proof is like that of Lemma 3.1 □

After the fault propagation process, every working node in the first half of the network has an output that leads to a working node, and every working node in the second half has both an up output and a down output that lead to working nodes. Furthermore, since at most a 2α fraction of the nodes in each merger on level $-\log N + \log L$ are blocked, and $2\alpha L < L - 1$ for $L > 1/2\alpha(\beta - 1)$ and $2\alpha\beta < 1$, each of the N/L inputs has an edge to a working node on level $-\log N + \log L$. As a consequence, we can establish a path through working nodes from any unused input to any unused output in $O(\log N)$ bit steps using a simple greedy algorithm. Since the declaration of blocked nodes takes just $O(\log N)$ bit steps, and since the greedy routing algorithm is easily accomplished in $O(\log N)$ bit steps, the entire process takes just $O(\log N)$ bit steps.

The preceding algorithm for establishing paths one after another in the multi-Beneš network implies that it is a wide-sense nonblocking connector. The proofs of Lemmas 3.1 and Lemmas 3.2, however, do not make any assumptions about the strategy used to make previous connections between inputs and outputs. Indeed, the only requirement is that there are at most M/L paths through each M-input splitter or M-output merger, which holds for any path selection strategy. Therefore, no matter how the paths for the previous connections were found, there is still at least one working node in each block at level $-\log N + \log L$, and as a consequence, at least one path between any unused input and unused output. Thus the multi-Beneš network is also a strict-sense nonblocking connector. As such, it is not really necessary to label the nodes as blocked or working; a simple on-line algorithm like breadth-first search is guaranteed to find a path. When simultaneous requests are dealt with in Section 4.4, however, a proper labeling will be important.

4. Establishing many paths at once. In this section, we describe an on-line algorithm for routing an arbitrary number of additional calls in $O(\log N)$ bit steps.

As before, we assume for the time being that each input and each output is involved in at most one two-party call. Extensions to the algorithm for handling multiparty calls are described in Section 5. We also assume that paths are established between inputs and outputs on rows congruent to 0 mod L in the multi-Beneš network, where L is a power of 2 and $L \geq 1/\alpha$. This will insure that no splitter or merger is ever overloaded.

To simplify the exposition of the algorithm, we start by describing an on-line algorithm for routing any initial set of paths in a multibutterfly (i.e., we don't worry about the nonblocking aspect of the problem for the time being). This comprises the first known circuit-switching algorithm for the multibutterfly. (Previous routing algorithms for the multibutterfly [17, 37] only worked for the store-and-forward model of routing.) The existence of the circuit-switching algorithm provides another proof that the multibutterfly is a rearrangeable connector. We conclude by modifying the definitions of busy and blocked nodes from Section 3 and showing how to implement the circuit-switching algorithm on a multi-Beneš network so that it works even in the presence of previously established calls.

4.1. Unshared neighbors. Our circuit-switching algorithm requires the splitters in the multibutterfly to have a special "unshared-neighbors" property defined as follows.

DEFINITION 4.1. *An M-input splitter is said to have the (α, δ) unshared neighbor property if in every subset X of $k \leq \alpha M$ inputs, there are δk nodes in X that have an up-output neighbor that is not adjacent to any other node in X, and there are δk nodes in X that have a down-output neighbor that is not adjacent to any other node in X (i.e., δk nodes in X have an unshared up-neighbor, and δk nodes have an unshared down-neighbor).*

LEMMA 4.2. *Any splitter with the (α, β) expansion property also has the (α, δ) unshared-neighbors property where $\delta = 2\beta/d - 1$, provided that $\beta > d/2$.*

Proof. Consider any set X of $k \leq \alpha M$ inputs in an M-input splitter. These nodes have at least βk neighbors among the up (down) outputs. Let n_1 denote the number of these neighbors incident to precisely one node of X, and let n_2 denote the number of neighbors incident to two or more nodes of X. Then $n_1 + n_2 \geq \beta k$ and $n_1 + 2n_2 \leq dk$. Solving for n_1 reveals that $n_1 \geq (2\beta - d)k$. Hence at least $(2\beta/d - 1)k$ of the nodes in X are adjacent to an unshared neighbor. □

By Equation 2.1, we know that randomly-generated splitters have the (α, δ) unshared-neighbors property where δ approaches 1 as d gets large and α gets small. Explicit constructions of such splitters are not known, however. Nevertheless, we will consider only multibutterflies with the (α, δ) unshared-neighbors property for $\delta > 0$ in what follows.

Remark: The (α, β) expansion property ($\beta > d/2$) is a sufficient condition for the unshared-neighbors property, but by no means necessary. In fact, we can easily prove the existence of random splitters which have a fairly strong (α, δ) unshared-neighbors property for small degree. For such graphs, the routing algorithm we are about to describe is more efficient in terms of hardware required. However, multibutterflies with expansion properties will remain the object of our focus.

4.2. A level-by-level algorithm. Our first algorithm extends the paths from level 0 to level $\log N$ by first extending all the paths from level 0 to level 1, then from level 1 to level 2, and so on. As we shall see, extending the paths from one level to the next can be done in $O(\log N)$ bit steps, so the total time is $O(\log^2 N)$ bit steps.

In a multibutterfly with the (α, δ) unshared-neighbors property, it is relatively easy to extend paths from one level to the next because paths at nodes with unshared neighbors can be extended without worrying about blocking any other paths that are trying to reach the next level. The remaining paths can then be extended recursively. In particular, all the paths can be extended from level l to level $l+1$ (for any l), by performing a series of "steps", where each step consists of:

1. every path that is waiting to be extended sends out a "proposal" to each of its output (level $l+1$) neighbors in the desired direction (up or down),
2. every output node that receives precisely one proposal sends back its acceptance to that proposal,
3. every path that receives an acceptance advances to one of its accepting outputs on level $l+1$.

Note that each step can be implemented in a constant number of bit steps.

Since the splitters connecting level l to level $l+1$ have $M = N/2^l$ inputs, and at most $M/2L$ paths must be extended to the upper (or lower) outputs, for $L > 2/\alpha$, the number of inputs containing these paths is at most αM. Thus, we can apply the (α, δ) unshared-neighbors property to these nodes. As a consequence, in each step the number of paths still remaining to be extended decreases by a $(1-\delta)$ factor. After $\log(N/L2^{l+1})/\log(1/(1-\delta))$ steps, no paths remain to be extended.

By using the path-extension algorithm just described to extend all of the paths from level 0 to level 1, then all of the paths from level 1 to level 2, and so on, we can construct all the paths in

$$\sum_{l=0}^{\log \frac{N}{L}-1} \frac{\log \frac{N}{L2^{l+1}}}{\log \frac{1}{1-\delta}} \leq \frac{\log^2 \frac{N}{2L}}{\log \frac{1}{1-\delta}} = O(\log^2 N)$$

steps.

4.3. A faster algorithm. To construct the paths in $O(\log N)$ bit steps we modify the first algorithm as follows. Given a set of at most αM paths that need to be extended at an M-input splitter, the algorithm does not wait $\Theta(\log M)$ time for every path to be extended before it begins the extension at the next level. Instead, it waits only $O(1)$ steps, in which time the number of unextended paths falls to a fraction ρ of its original value. We will choose ρ to be less than $1/d$. Now the path extension process can start at the next level. The only danger here is that the ρ fraction of paths left behind may find themselves blocked by the time they reach the next level, and so we need to ensure that this won't happen. Therefore, stalled paths send out *placeholders* to all of their neighbors at the next level, and henceforth the neighbors with placeholders participate in the path extension process at the next level, as if they were paths. Thus, a placeholder not only reserves a spot that may be used by a path at a future time, but also helps to chart out the path by continuing to extend ahead. Since a placeholder doesn't know which path will ultimately use it, a node holding a placeholder must extend paths into both the upper and lower output portions of its splitter. A placeholder that first extends a path into the upper output portion of its splitter continues to attempt to extend a path into the lower portion, and vice versa. We will call a path from the inputs of the network to the inputs of any splitter in the network a *real path* if it contains no placeholders. The goal of the algorithm, of course, is to extend real paths all the way through the network. Any path that contains at least one placeholder is called a *placeholder path*.

Since each stalled path generates up to $2d$ placeholders at the next level, and these placeholders might later become stalled themselves, there is a risk that the network will become clogged with placeholders. In particular, if the fraction of inputs in a splitter that are trying to extend rises above α, the path extension algorithm ceases to work. Thus, in order to prevent placeholders from clogging the system, whenever a stalled path, either real or a placeholder, gets extended into either the upper or lower output portion of a splitter, it sends a *cancellation signal* to each of the nodes in that portion of the splitter that are holding placeholders for it. When a placeholder is replaced by a real path, one of the two directions (up or down) into which the placeholder has been attempting to extend becomes unnecessary. If the placeholder has already extended its path in that direction, a single cancellation is sent along the edge that the path uses. Otherwise, a cancellation is sent to each of the d placeholding neighbors in that direction. When a placeholding node gets cancellations from all of the nodes that had requested it to hold their places, it ceases its attempts to extend. It also sends cancellations to any nodes ahead of it that may be holding a place for it. Note that a placeholding node that has received cancellations from all but one of the nodes that had requested it to hold their places continues to try to extend into both the upper and lower output portions of the splitter. As we shall see, this scheme of cancellations prevents placeholders from getting too numerous.

The $O(\log N)$-step algorithm for routing paths proceeds in *phases*. Each path is restricted to extend forward by at most one level during each phase. We refer to the first wave of paths and placeholders to arrive at a level as the *wavefront*. The wavefront moves forward by one level during each phase. A phase consists of the following three parts:

(i) C steps of passing cancellation signals. These cancellation signals travel at the rate of one level per step.

(ii) T steps of extending paths from one level to the next. In this time, the number of stalled (i.e., unextended) paths at each splitter drops by least a factor of ρ, where $\rho \leq (1-\delta)^T$.

(iii) 1 step of sending placeholders to all neighbors of paths in the wavefront that were not extended during the preceding T steps

Note that for constant T and C, each phase can be performed in $O(1)$ bit steps. We will assume that $C \geq 2$ so that cancellation signals have a chance to catch up with the wavefront, and that $d \geq 3$.

The key to our analysis of the algorithm is to focus on the number of stalled paths (corresponding to real paths *or* placeholders) at the inputs of each splitter. In phase t of the algorithm, where the first phase is phase 0, the wavefront advances from level t to level $t+1$. Let P_i denote the maximum fraction of inputs containing wavefront paths (real and placeholder) in a level i splitter that wish to extend to the upper (or similarly, to the lower) outputs at the end of phase $i-1$, i.e., when the wavefront arrives at level i, and let $S(i,t)$ denote the maximum fraction of inputs that contain stalled paths that wish to extend to the upper (or similarly, to the lower) outputs of any splitter at level i at the end of phase t. Note that $S(i,t)=0$ for $t<i$, since there are no paths to extend at level i before phase i. Also, note that $S(i,i) \leq \rho P_i$.

The following lemmas will be useful in proving that every path is extended to completion in $\log N$ phases provided that $L \geq 1/\alpha$ and $\rho < 1/14d$.

LEMMA 4.3. *If $P_i \leq \alpha$ then $S(i,t) \leq \rho^{t-i} S(i,i) \leq \rho^{t+1-i} P_i$ for $t \geq i$.*

Proof. In each phase of the algorithm, the number of stalled paths at the inputs drops by a factor of ρ, provided that the number of paths trying to extend is never greater than an α fraction of the inputs of the splitter. Since the number of paths reaching the inputs never increases after the wavefront arrives, this condition is always satisfied. □

The following lemma bounds the size of the wavefront in terms of the number of stalled paths behind it.

LEMMA 4.4.

$$P_i \leq \frac{1}{2L} + 2dS(i-1,i-1) + \sum_{l=0}^{\infty}\sum_{k=1}^{C} 2^{Cl+k+1} dS(i-1-Cl-k, i-l-2).$$

Proof. The first term, $1/2L$, is an upper bound on the fraction of inputs through which real paths that wish to extend to the upper outputs (or similarly to the lower outputs) will ever pass. The $2dS(i-1,i-1)$ term represents the fraction of inputs that could hold placeholders generated by stalled paths at level $i-1$ (the factor of 2 comes in because the number of inputs in a splitter at level $i-1$ is twice as many as those in a level i splitter). The $4dS(i-2,i-2)$ term ($l=0$, $k=1$) is an upper bound on the fraction of inputs containing placeholders that were generated by paths stalled at level $i-2$ when the wavefront was extended to level $i-1$ in phase $i-2$. Next, for $C \geq 2$, the contribution of placeholders from level $i-3$ is $8dS(i-3,i-2)$ (here $l=0$, $k=2$), not $8dS(i-3,i-3)$, since paths that are stalled at level $i-3$ during phase $i-3$, but get through during phase $i-2$, send cancellation signals to levels $i-2$ and $i-1$ during the first part of phase $i-1$. Hence, these paths do not contribute placeholders to the wavefront when it is extended from level $i-1$ to level i. The contribution from level $i-C-2$ is $2^{C+2}dS(i-C-2,i-3)$ (here $l=1$, $k=1$), since paths that are extended during the second part of phase $i-3$ send cancellations that reach level $i-2$ during the first part of phase $i-2$. These cancellations then reach level $i-1$ during the first part of phase $i-1$. The rest of the terms in the summation may be counted similarly. Although our summation seems to have infinitely many terms, only finitely many of them are non-zero. □

The next lemma, Lemma 4.5, presents a weaker bound on P_i. The difference between this lemma and the previous lemma is that in Lemma 4.5 we assume that a cancellation signal must reach level i rather than $i-1$ before the start of the path extension part of phase $i-1$ in order for it to have an effect on the size of the wave propagating from level $i-1$ to level i. The reason for this assumption is that we will later speed up the algorithm by overlapping the cancellation passing and path extension parts of each phase.

LEMMA 4.5.

$$\begin{aligned} P_i \leq{}& \frac{1}{2L} + 2dS(i-1,i-1) \\ &+ \sum_{k=2}^{C} 2^k dS(i-k,i-2) \\ &+ \sum_{l=1}^{\infty}\sum_{k=1}^{C} 2^{Cl+k} dS(i-Cl-k, i-l-2). \end{aligned}$$

Proof. The proof is similar to that of Lemma 4.4. □

The following lemma shows that for the right choices of L, ρ, d, and C, no splitter ever receives too many paths (real or placeholders) that want to extend to the upper outputs (and similarly, to the lower outputs).

LEMMA 4.6. *For $L \geq \frac{1}{\alpha}$, $\rho \leq \frac{1}{14d}$, $d \geq 3$, and $C \geq 3$, $P_i \leq \alpha$, for $0 \leq i \leq \log(N/L)$.*

Proof. We prove by induction on i that for $\gamma = \frac{\alpha}{14d}$, $P_i \leq \alpha$, and $S(i,i) \leq \rho P_i \leq \gamma$. For the base case, observe that $P_0 \leq 1/2L$, and $S(0,0) \leq \rho P_0$ (by applying Lemma 4.3 with $i = 0$ and $t = 0$). Hence, $S(0,0) \leq \alpha/28d = \gamma/2$. For the inductive step, we apply Lemma 4.3 to the recurrence of Lemma 4.5, which yields:

$$\begin{aligned}
P_i &\leq \frac{1}{2L} + 2d\gamma + \sum_{k=2}^{C} 2^k d\gamma\rho^{k-2} \\
&\quad + \sum_{l=1}^{\infty}\sum_{k=1}^{C} 2^{Cl+k} d\gamma\rho^{(C-1)l+k-2} \\
&= \frac{1}{2L} + 2d\gamma + \frac{4d\gamma(1-(2\rho)^{C-1})}{1-2\rho} \\
&\quad + \frac{d\gamma 2^{C+1}\rho^{C-2}(1-(2\rho)^C)}{(1-2^C\rho^{C-1})(1-2\rho)} \\
&\leq \frac{1}{2L} + 2d\gamma + 4.2d\gamma + .5d\gamma.
\end{aligned}$$

Note that in the last inequality we have used the fact that $d \geq 3$, $C \geq 3$, and $\rho \leq 1/14d$. (We really only needed $C \geq 2$, but the constants are better for $C \geq 3$.) Thus if $\gamma = \alpha/14d$ and $L \geq 1/\alpha$, then $P_i \leq \alpha$. Also, by Lemma 4.3, $S(i,i) \leq \rho P_i$ and if $\rho \leq 1/14d$, we have: $S(i,i) \leq \alpha/14d = \gamma$, thereby establishing the induction. ∎

From Lemma 4.6, it is clear that no splitter ever has more than an α fraction of its inputs containing paths to be extended to the upper (or lower) outputs. Therefore the path-extension algorithm is never swamped by placeholders and always works as planned at each level, cutting down the number of stalled paths by a factor of ρ during each phase. Hence, $\log(\alpha M)/\log(1/\rho)$ phases after the wavefront arrives at a splitter of size M, all paths are extended. Since the wavefront arrives at level i during phase $i-1$, the algorithm establishes all real paths to level $\log(N/L)$ (recall that the last $\log L$ levels have been removed) by phase

$$\max_{0\leq i<\log(N/L)} \max\left\{\left(i - 1 + \frac{\log\frac{\alpha N}{2^i}}{\log\frac{1}{\rho}} + \frac{\log\frac{N}{L} - i}{C}\right), \log\frac{N}{L} - 1\right\} =$$

$$\max_{0\leq i<\log(N/L)} \max\left\{\left(\frac{\log\alpha N}{\log\frac{1}{\rho}} + \frac{\log\frac{N}{L}}{C} + i\left(1 - \frac{1}{\log\frac{1}{\rho}} - \frac{1}{C}\right) - 1\right), \log\frac{N}{L} - 1\right\}$$

phases, since a path that is last stalled at level i extends to level $i+1$ by phase $i - 1 + \log(\alpha N/2^i)/\log(1/\rho)$, and if the wavefront reaches level $\log(N/L)$ before its cancellation signals do, then these signals arrive $(\log(N/L) - i)/C$ phases later. Otherwise, if the cancellation signals catch up to the wavefront (but the path is never again stalled), then the path extends to level $\log(N/L)$ by phase $\log(N/L) - 1$ For $C \geq 2$ and $\rho < 1/4$, this expression takes on a maximum value of $\log(N/2L) - 1 + \log(2\alpha L)/\log(1/\rho) + 1/C$. At first, this result seems too good to be true, but stalled real paths catch up to the wavefront very quickly once they get through, and they get through at a very high rate. Hence, all real paths get through to the final level along with the wavefront!

Since the number of phases required is basically $\log(N/L)$, the overall time for the algorithm depends mainly on the parameters C and T. By propagating the cancellations at the same time that paths are extended, a single phase can be implemented in $\max(C, 2T+1)$ steps. As long as $\rho < 1/14d$, the algorithm will work for $C \geq 3$. Since $\beta < d-1$, and Lemma 4.2 gives us $\delta = 2\beta/d - 1$, and we need $\rho = (1-\delta)^T < 1/14d$, T must be at least 2. In general, in order to make T small, we need δ to be large. In order to achieve large δ, we need β to be close to d, which requires α to be small (and consequently L to be large) and d to be large. By using good splitters ($\delta \approx 1$), α small, d large, $C = 5$, and $T = 2$, and replacing each edge with a small constant number of edges, we can obtain a $(5+\varepsilon)\log N$-step algorithm for routing all the paths. Unfortunately, d and L need to be quite large to achieve this bound. For more reasonable values of d (less than 10) and L (less than 150), we can achieve provable routing times of about $100 \log N$. Fortunately, the algorithms appear to run faster in simulations [19].

It is worth noting that each node only needs to keep track of a few bits of information to make its decisions. This is because only the ith bit of the destination is needed to make a switching decision at level i, and therefore a node at that level looks at this bit, strips it off, and passes the rest of the destination address onward. The path as a whole snakes forward through the network. If it ever gets blocked, the entire snake halts behind it. The implementation details for this scheme are straightforward. Previously, only the AKS sorting circuit was known to achieve this performance for bounded-degree networks, but at a much greater cost in complexity and constant factors. Recently, Leighton and Plaxton have also developed a randomized algorithm for sorting on the butterfly in $O(\log N)$ bit steps [20].

4.4. Routing many paths in a nonblocking fashion on a multi-Beneš network. It is not difficult to implement the circuit-switching algorithm just described on a multi-Beneš network. The main difference between routing through a multi-Beneš network and a multibutterfly network is that in the first half of the multi-Beneš network, a path at a merger input is free to extend to any of the $2d$ neighboring outputs. As the following definition and lemma show, the mergers have an unshared-neighbor property analogous to that of the splitters.

DEFINITION 4.7. *An M-input merger is said to have the (α, δ) unshared neighbor property if in every subset X of $k \leq \alpha M$ inputs (either up or down or any combination), there are δk nodes in X which have an output neighbor that is not adjacent to any other node in X.*

LEMMA 4.8. *Any merger with the (α, β) expansion property also has the (α, δ) unshared-neighbors property where $\delta = 2\beta/d - 1$, provided that $\beta > d/2$.*

Proof. The proof is essentially the same as that of Lemma 4.2. ◻

In order to route around existing paths in a multi-Beneš network, we combine the circuit-switching algorithm with the kind of analysis used in Section 3. To do so, we need to modify the definition of being blocked. A splitter input on level l, $0 \leq l < \log N - \log L$, is blocked if more than $2\beta - d - 1$ of its d up (or down) neighbors on level $l+1$ are busy or blocked. A merger input on level l, $-\log N + \log L \leq l < 0$, is blocked if more than $4\beta - 2d - 2$ of its $2d$ neighbors on level $l+1$ are either busy of blocked. Any node that is not blocked is considered to be working.

4.4.1. The subnetwork of working nodes. The following pair of lemmas show that for $\beta > (d+1)/2$, an unshared-neighbor property is preserved on the working nodes.

LEMMA 4.9. *For $\beta > (d+1)/2$, the working splitter inputs have an $(\alpha, 1/d)$ unshared-neighbor property.*

Proof. In the proof of Lemma 4.2 we show that every set X of $k \leq \alpha M$ nodes in an M-input splitter has at least $(2\beta - d)k$ neighbors in the upper and lower outputs with only one neighbor in X. If X is a set of working switches, then at most $(2\beta - d - 1)k$ of these unshared neighbors can be busy or blocked. Thus, at least k of the unshared neighbors must be working. ∎

LEMMA 4.10. *For $\beta > (d+1)/2$, the working merger inputs have an $(\alpha, 1/d)$ unshared-neighbor property.*

Proof. The proof is similar to that of Lemma 4.9. ∎

Of course, we must also check that the new blocking definition does not result in any inputs of the multi-Beneš network becoming blocked. This can be done with an argument similar to that in Lemmas 3.1 and 3.2.

LEMMA 4.11. *For $\beta > 2d/3 + 2/3$ and $L > 1/2\alpha(3\beta - 2d - 2)$, less than a 2α fraction of the inputs in any splitter are declared to be blocked. Furthermore, less than an α fraction of the inputs are blocked because of busy and blocked nodes from the upper outputs, and less than an α fraction are blocked because of the lower outputs.*

Proof. The proof is by induction on level number, working backwards from level $\log N - \log L$ to level 0. For the base case, observe that on level $\log N - \log L$ none of the nodes are blocked. Now suppose that αM of the inputs of some M-input splitter are blocked by upper outputs (say), and let $|U|$ be the set of busy or blocked upper outputs. Since the blocked inputs have at least $\alpha\beta M$ neighbors among the upper outputs, and at most $2d - 2\beta + 1$ edges out of each blocked node lead to working nodes, $|U| \geq \alpha M(\beta - (2d - 2\beta + 1)) = \alpha M(3\beta - 2d - 1)$. By induction, however, the number of blocked upper outputs is at most αM and thus $|U| \leq \alpha M + M/2L$. For $L > 1/2\alpha(3\beta - 2d - 2)$, we have a contradiction. ∎

LEMMA 4.12. *For $\beta > 2d/3 + 2/3$ and $L > 1/2\alpha(3\beta - 2d - 2)$, at most a 2α fraction of the up inputs and at most a 2α fraction of the down inputs in any merger are declared blocked.*

Proof. The proof is similar to that of Lemma 4.11. ∎

4.4.2. Routing new paths. Once the working nodes have been identified, new paths from the inputs to the outputs of the multi-Beneš network can be established using an algorithm that is essentially the same as the circuit-switching algorithm for multibutterflies described in Section 4.3. There are two main differences. First, in the multi-Beneš network, only working nodes are used. However, by Lemmas 4.9 and 4.10 the working switches have an $(\alpha, 1/d)$ unshared neighbors property. Hence, we can run the algorithm of Section 4.3 with $\delta = 1/d$. Second, routing in the first half of the multi-Beneš network is actually easier than in the second half, which is a multibutterfly, since there is no notion of up or down edges. The goal is simply to get each new path from an input on level $-\log N + \log L$ to any working node on level 0. The algorithm uses placeholders and cancellation signals in the first half in the same way that they are used in the second half.

4.5. Processing incoming calls. Since the working nodes must be identified before new paths can be routed, incoming calls are processed in batches. When a new call originates at an input, it waits until the paths are established for the batch that is currently being processed. When all of the calls in that batch have been established, the working nodes are identified, and then the paths for the new batch are established. Since identifying the working nodes and routing the new paths both take at most $O(\log N)$ bit steps, the time to process each batch is $O(\log N)$ bit steps, and no call waits for more than $O(\log N)$ bit steps before being established, including the time waiting for the previous batch to finish.

5. Extensions.

5.1. Multiparty calls. If all of the parties in a multiparty call are known to a caller at the start of the call, then it is possible to extend the algorithms in Sections 3 and 4 to route the call from the caller to all of the parties. As a call advances from level 0 to level $\log N$ of the multi-Beneš network, it simply creates branches where necessary to reach the desired output terminals. The bit complexity of the algorithm may increase, however, because more than $O(\log N)$ bits may be needed to specify the set of outputs that the call must reach.

The situation becomes more complicated if parties to a multiparty call are to be added after the call is already underway. One possible solution is to set up paths in the network from the caller to the parties in the call that make multiple passes through the network. To simplify the explanation, let us assume that the input in row i and the output in row i of the multi-Beneš network are actually the same node, for $0 \leq i \leq N-1$. (Thus each input/output can be involved in at most one call.) A multiparty call is established by constructing a binary tree whose root is the caller and whose internal nodes and leaves are the parties in the call. Each node of the binary tree is embedded at an input of the multi-Beneš network, and each edge in the tree from a parent to a child is implemented by routing a path through the multi-Beneš network from the input at which the parent is embedded to the output (which is also an input) at which the child is embedded. To add a new party to the call, we add a new node to the binary tree wherever its depth will be minimum. This ensures that the depth of a tree with l parties will be $O(\log l)$. Since each edge of the binary tree corresponds to a path of length $\log N$ in the network, the path from the root to any other node in the tree has length at most $O(\log^2 N)$ in the network. It's easy to see that a new party can be added in $O(\log^2 N)$ bit steps, but with a little work the time can be brought down to $O(\log N)$ bit steps. One problem with this scheme is that the parties corresponding to internal nodes of the binary tree cannot hang up without also disconnecting all of their descendants. Although this solution is not as elegant as those proposed in [10] for wide-sense generalized nonblocking connectors, no polynomial time routing algorithms are known for those constructions.

5.2. Multiple calls to the same output. If many parties want to call the same output terminal, then we have two options: merging the callers into a single multiparty call, or giving busy signals to all but one of the callers.

In either case, the first thing to do is to sort the calls according to their destinations. Unfortunately, no deterministic $O(\log N)$-bit-step sorting algorithm is known for the multibutterfly network at present, although $O(\log N)$ word- and bit-step randomized algorithms are known for the butterfly [20, 34]. If a deterministic $O(\log N)$-bit-step algorithm is required, the multibutterfly could be augmented with a sorting circuit such as the AKS sorting circuit [2]. The AKS sorting circuit will provide us with a set of edge-disjoint paths from its inputs to its outputs. If node-disjoint paths are desired, then each 2×2 comparator in the circuit can be replaced by a 2×2 complete bipartite graph. Note that in neither case is the sorting circuit a nonblocking network, since adding new calls at the inputs may alter the sorted order, thus disrupting existing paths. In the remainder of this section, we will use a sorting circuit either in conjunction with a butterfly network to route calls in a rearrangeable fashion, or in conjunction with a multibutterfly to route calls in a nonblocking fashion. In the latter case, the sorting circuit is used only to help compute the routes that the calls take, and not to route the calls themselves.

Once the calls have been sorted, a parallel prefix computation is applied to the sorted list of calls. For each destination, one of the calls is marked as a winner, and the others as losers. For a description of prefix operations, and how they can be implemented in $O(\log N)$ bit steps on a complete binary tree (which is a subgraph of the butterfly), see [16, Section 1.2].

If it suffices to send a busy signal to all of the callers except one, then these signals can be sent back to the losers along their paths through the sorting circuit, and the winning path can be established (in a nonblocking fashion) in a multibutterfly network.

If the calls are to be merged into a single call, then the next step is to label the winners according to their positions in the sorted order, and to give each loser the label of the winner for its destination. This is also prefix computation.

To route calls in a rearrangeable fashion, we identify the outputs of the sorting circuit with the inputs of a butterfly network. Each call is routed greedily in the butterfly network to the output in the row with the same number as the winner's index. This type of routing problem is called a *packing* problem. Surprisingly, only calls with the same destination will collide during the routing of any packing problem [16, Section 3.4.3]. After this step, all of the calls to the same destination have been merged into a single call. Since the calls remain sorted by destination, the problem of routing them to their destinations is called a *monotone routing* problem. Any monotone routing problem can be solved with a single pass through two back-to-back butterfly networks without collisions [16, Section 3.4.3].

To route calls in a nonblocking fashion, we can either assume that all callers are known at the time that a call is established or not. If all of the callers are known, we can route the calls backwards through a multibutterfly from the shared output to each of the inputs of the callers using the first scheme described in Section 5.1. Otherwise, we can use the second scheme of Section 5.1 in reverse to route the calls using paths of length $O(\log^2 N)$.

5.3. Removing the distinction between terminals and non-terminals. In this section we generalize the routing algorithm of Section 4 by removing the distinction between nodes that are terminals and nodes that are not. The algorithm in this section requires $O(\log N)$ word steps, not bit steps. Recall that in the word model, each edge can transmit a word of $O(\log N)$ bits in a single step. The goal of the algorithm is to establish a set of disjoint paths, each of which may start or end at any node in the network. The following similar problem was studied by Peleg and Upfal [27].

> Given an expander graph, G, K *source* nodes, $a_1, \ldots, a_K$ in G, and K *sink* nodes, $b_1, \ldots, b_K$ in G, where the sources and sinks are all distinct (i.e., $a_i \neq a_j$ and $b_i \neq b_j$ for $i \neq j$, and $a_i \neq b_j$ for all i and j), construct a path in G from each source a_i to the corresponding sink b_i, so that no two paths share an edge.

Peleg and Upfal presented polylogarithmic time algorithms for finding K edge-disjoint paths in any n-node expander graph, provided that $K \leq n^\rho$, where ρ is a fixed constant less one. In this section we show that if we are allowed to specify the network (but not the locations of the sources and sinks) then it is possible to construct even more paths. In particular, we describe an n-node bounded-degree network, R, and show how to find K edge-disjoint paths in it in $O(\log n)$ time, provided that $K \leq O(n/\log n)$.

Furthermore, we show how to find node-disjoint paths between $\Theta(K)$ of the sources and sinks.

5.3.1. The network. The network R consists of four parts, each of which contains $\log N + 1$ levels of N nodes. Each of the first three parts shares its last level with the first level of the next part, so the total number of levels is $4\log N + 1$, and the total number of nodes in the network is $n = N(4\log N + 1)$.

The first part is a set of $\log N + 1$ levels labeled $-2\log N$ through $-\log N$. For $-2\log N \le i < -logN$, the edges connecting level i to $i+1$, form an N-input merger. Hence, every set of $k \le \alpha N$ nodes on one level has at least $2\beta k$ neighbors on the next level, where α, β, and d are related as in Equation 2.2.

The second part consists of a multibutterfly whose levels are labeled $-\log N$ through 0. The multibutterfly has expansion property (α, β), where α, β, and d are related as in Equation 2.1.

The third and fourth parts are the mirror images of the first and second parts. The levels of these parts are labeled 0 through $2\log N$.

Although any node in R can be chosen to be a source or a sink, it would be more convenient if all of the sources were to reside in the first part, and all the sinks in the fourth. Thus, the node on level $-i$ of the second part, i of the third part, and $2\log N - i$ of the fourth part each have an edge called a *cross* edge to the corresponding node on level $-2\log N + i$ of the first part. Similarly, each node in the fourth part has cross edges to the corresponding nodes in first, second, and third parts. If a node in any part other than the first is chosen to be a source, then its path begins with its cross edge to the first part. If a node in any part other than the fourth is chosen to be a sink, then the path to it ends with a cross edge from the fourth part. At this point, each node in the first part may represent up to four sources, and each node in the fourth part may represent up to four sinks.

5.3.2. Constructing node-disjoint paths. For the paths to be node-disjoint, each path must avoid the sources and sinks in the second and third parts as it passes from the first part to the fourth part. To avoid these sources and sinks, we declare them to be *blocked*. We then apply the technique of [17] for tolerating faults in multibutterfly networks to the second and third parts, treating blocked nodes as if they were faulty. The technique of [17] can be summarized as follows. First, any splitter (and all nodes that can be reached from that splitter) that contains more than a $2\alpha(\beta' - 1)$ fraction of blocked inputs is *erased*, meaning that its nodes cannot be used for routing, where $\beta' = \beta - d/4$. Next, working backwards from the outputs to the inputs, a node is declared to be blocked if more than $d/4$ of its up or down neighbors at the next level are blocked (and not erased). (Note that it is not possible for all of a node's up and down neighbors to be erased unless that node is also erased.) Upon reaching the inputs of the network, all the blocked nodes are erased. The switches that are not erased are said to be *working*. The expansion property of the network of working switches is reduced from β to β'.

The following lemmas bound the number of inputs (on levels $-\log N$ and $\log N$) and outputs (on level 0) that are erased in the second and third parts. Note that Lemma 5.1 bounds the number of inputs that are erased, but are not themselves blocked. (All the blocked inputs are erased.) Note also that since the two parts share level 0, the number of erased nodes on that level may be as large as twice the bound given in Lemma 5.2.

LEMMA 5.1. *In addition to the (at most) K blocked inputs, at most $K/(\beta' - 1)$ nonblocked inputs are erased in the second and third parts.*

Proof. This lemma is essentially the same as Lemma 3.3 of [17]. ☐

LEMMA 5.2. *At most $K/2\alpha(\beta' - 1)$ outputs are erased in each of the second and third parts.*

Proof. This lemma is essentially the same as Lemma 3.1 of [17]. ☐

In both networks at least $N - O(K)$ of the inputs and $N - O(K)$ of the outputs are left working, where K is the number of sources (and sinks). Suppose that $K \leq \gamma N$, where γ is some constant. By choosing γ to be small, we can ensure that at least K of the nodes on level 0 are not erased in either the second or third parts. We call these K nodes the *rendezvous points*. By making β' (and hence d) large, we can also ensure that the number of nodes on levels $-\log N$ and $\log N$ that are erased, but are not themselves sources or sinks, is ϵK, where ϵ can be made to be an arbitrarily small constant.

The reconfiguration technique described in [17] requires off-line computation to count the number of blocked inputs in each splitter. In another paper, Goldberg, Maggs, and Plotkin [12] describe a technique for reconfiguring a multibutterfly on-line in $O(\log N)$ word steps.

The next step is to mark some of the nodes in the first part as blocked. We begin by declaring any node in the first part to be *reserved* if it is a neighbor of a source in the second, third, or fourth part via a cross edge. Now, working backwards from level $-\log N - 1$ to $-2\log N$, a node is declared *blocked* if at least $d/2$ of its $2d$ neighbors at the next level are either sources, sinks, blocked, reserved, or erased. We call a node that is not a source or a sink, and is not reserved, blocked, or erased, a *working* node.

Where did the $d/2$ bound on non-working neighbors come from? In order to be apply the routing algorithm of Section 4.3, the subnetwork of working nodes must have an (α, δ) unshared neighbor property. Let β' be the largest value such that the subnetwork of working nodes has an (α, β') expansion property (where (α, β) is the original expansion property of the first part). To show that the subnetwork of working nodes has an (α, δ) unique neighbors property, we need $\beta' > d/2$. If every working node has at most $d/2$ non-working neighbors, then the subnetwork of working nodes has expansion property $(\alpha, \beta - d/4)$. (Recall that we multiply the β' parameter by 2 to get the actual expansion in each merger.) Thus $\beta' > \beta - d/4$. If $\beta > 3d/4$, then $\beta' > d/2$. By restricting a working switch to have fewer non-working neighbors, we could have reduced the required expansion from $3d/4$ down to nearly $d/2$. As the following lemma shows, however, if a working switch can have $d/2$ non-working neighbors, then we also need $\beta > 3d/4$ in order to ensure that there aren't too many blocked nodes. If we were to allow a working switch to have fewer (or more) than $d/2$ non-working neighbors, then one of the two "$\beta > 3d/4$" lower bounds would increase, and the network would require more expansion.

LEMMA 5.3. *Let f denote the total number of nodes declared blocked in the first part, let $K \leq \gamma N$ denote the number of sources and sinks, and let ϵK denote the number of nodes on level $-\log N$ that are not sources or sinks, but are erased. Then if $(2+\epsilon)\gamma < (2\beta - 3d/2 - 1)\alpha$, then $f \leq \frac{2+\epsilon}{2\beta - 3d/2 - 1}K$.*

Proof. First, suppose that the total number of blocked nodes in the first part is at most αN. Then the f blocked nodes must have at least $(2\beta - 3d/2)f$ neighbors that are either sources, sinks, blocked, reserved, or erased, since each blocked node has at most $3d/2$ neighbors that are working. Since there are a total of at most K sources and reserved nodes in the first part, at most K sinks, and at most ϵK nodes on level $-\log N$ that are erased, but are not sources or sinks, we have

$$f + 2K + \epsilon K \geq (2\beta - 3d/2)f,$$

which implies that $f \leq \frac{2+\epsilon}{2\beta-3d/2-1}K$.

Otherwise, suppose that there are more than αN blocked nodes in the first part. Let us rank the nodes according to the levels that they appear on (breaking ties within a level arbitrarily), with the nodes on level $\log N - 1$ having highest rank, and those on level $-2\log N$ the lowest. Since the αN blocked nodes with highest rank must have at least $(2\beta - 3d/2)\alpha N$ neighbors that are sources, sinks, blocked, reserved, or erased, we have $\alpha N + 2K + \epsilon K \geq (2\beta - 3d/2)\alpha N$, a contradiction for $(2+\epsilon)\gamma < (2\beta - 3d/2 - 1)\alpha$. □

An identical process is applied to the fourth part, with blocked nodes propagating from level $\log N$ to level $2\log N$, and a lemma analogous to Lemma 5.3 can be proven, showing that there are at most $((2+\epsilon)/(2\beta - 3d/2 - 1))K$ blocked nodes in this part.

Because each node in the first part may be reserved by one source in each of the second, third, and fourth parts, it may not be possible for all the sources to establish their paths. If several sources wish to begin their paths at the same node, then one is locally and arbitrarily selected to do so, and the others give up. Since at most four paths start at any node in the first section, at least $K/4$ of the sources are able to begin their paths. Each source then sends a message to the corresponding sink. A message first routes across the row of its source to level $-\log N$ (recall that in every merger there is an edge from each input to the output in the same row), then uses the multibutterfly store-and-forward packet routing algorithm from [17, 37] to route to the row of its sink on level 0, then routes across that row in the third and fourth parts until it either reaches its sinks or reaches the cross edge to its sink. The entire routing can be performed in $O(\log N)$ word steps. Note that we can't use the circuit-switching algorithm of Section 4.3 here because there may be as many as $\log N$ sinks in a single row. The $K/4$ or more sinks that receive messages then each pick one of these messages (there are at most 4), and send an acknowledgement to the source of that message. At least $K/16$ sources receive acknowledgements, and these sources are the ones that will establish paths. A source that doesn't receive an acknowledgement gives up on routing its path.

Some of the nodes at which the remaining sources and sinks wish to begin or end their paths may have been declared blocked. None of these nodes will be used. By making β (and hence d) large, however, the number of blocked nodes in the first and fourth parts, $((2+\epsilon)/(2\beta - 3d/2 - 1))K$, can be made small relative to $K/16$. Thus, we are left with $\Theta(K)$ source-sink pairs.

The paths from the sources and the paths from the sinks are routed independently through the first two and last two parts, respectively. The path from a source a_i then meets the path from the corresponding sink b_i at a rendezvous point r_i on level 0.

The rendezvous points are selected as follows. First, the sources route their paths to distinct nodes on level $-\log N$ in $O(\log N)$ time using the algorithm from Section 4 on the working switches. Then the sources are numbered according to the order in which they appear on that level using a parallel prefix computation. A parallel prefix computation can be performed in $O(\log N)$ word (or even bit) steps on an N-leaf complete binary tree, and hence also on a butterfly. For a proof, see [16, Section 1.2]. (Note that although we are treating some of the nodes as if they were faulty, there are no actual faults in the network, so non-working nodes can assist in performing prefix computations.) The rendezvous points are also numbered according to the order in which they appear on level 0 using another prefix computation.

Next, using a packing operation, a packet representing the ith rendezvous point r_i is routed from r_i to the node in the ith row of level 0. At the same time, a packet representing the ith source a_i is routed from level $-\log N$, where a_i's path has reached so far, to the ith node of level 0. These two routings can be implemented in $O(\log N)$ word (or bit) steps on a butterfly [16, Section 3.4.3].

Once the packets for a_i and r_i are paired up, a packet is sent back to a_i's node on level $-\log N$ informing it of the position of r_i on level 0. (This is an unpacking operation.) The path for source a_i is then extended from level $-\log N$ to level 0 using the algorithm from Section 4.3 on the working switches. Then a packet containing the location of r_i is sent from a_i's node on level $-\log N$ to the node on level 0 that is in the same row that b_i lies on in the fourth part. This routing can be performed in $O(\log N)$ word steps using the store-and-forward multibutterfly routing algorithm of [17]. (We can't use the circuit switching algorithm because there may be as many as $\log N$ sinks in the same row.)

In $O(\log N)$ time, the packet works it way across the row from level 0 to b_i, which lies somewhere between levels $\log N$ and $2\log N$. (Note that although there may be as many as $\log N$ $b_i's$ in the same row, the total time is still at most $O(\log N)$.)

Finally, a path is extended from b_i to any working node on level $\log N$ and from there to r_i using the algorithm of Section 4.3 on the working switches.

5.3.3. Establishing edge-disjoint paths. It is easier to establish edge-disjoint paths in R than node-disjoint paths. In particular, it is not necessary to apply the technique of [17] for tolerating faults in multibutterflies to the second and third parts of the network as we did in order to establish the node-disjoint paths. The main thing that must be done is to modify the algorithm from Section 4 for locking down node-disjoint paths in a multibutterfly so that it allows a constant number of edge-disjoint paths to pass through each node. Let r be the maximum number of paths that may pass through a node. In order to replace the unshared neighbors protocol with one that allows r paths to pass through a node, we define the following r-neighbors property for splitters. Similar definitions hold for mergers, or for pairs of consecutive levels like those in the first and fourth parts of R.

DEFINITION 5.4. *An M-input splitter is said to have an (α, δ) r-neighbors property if in every subset X of $k \leq \alpha M$ inputs, there are subsets X_U and X_D of X such that $X_U \geq \delta k$ and $X_D \geq \delta k$, and every node in X_U (X_D) has at least r up-output (down-output) neighbors, each of which has at most r neighbors in X.*

The following lemma shows that a splitter with a sufficient expansion property also has an r-neighbors property.

LEMMA 5.5. *A splitter with an (α, β) expansion property has an (α, δ) r-neighbors property where*

$$\delta = \frac{\frac{r+1}{r}\beta - \frac{d}{r} - r + 1}{d - r + 1}.$$

Proof. The proof is similar to the proof of Lemma 4.2. Let X be a set of $k \leq \alpha M$ inputs in an M-input splitter, let n_r denote the number of up (down) outputs that have at least one, but at most r, neighbors in X, and let n_+ denote the number of up (down) outputs that have more than r neighbors in X. Then $n_r + n_+ \geq \beta k$, and $n_r + (r+1)n_+ \leq dk$. Solving for n_r yields

$$n_r \geq \left(\frac{r+1}{r}\beta - \frac{d}{r}\right)k.$$

Let δk denote the number of nodes in X with at least r up-output (down-output) neighbors, each of which has at most r neighbors in X. Then $\delta k d + (1-\delta)k(r-1) \geq n_r$, which implies that

$$\delta \geq \frac{\frac{r+1}{r}\beta - \frac{d}{r} - r + 1}{d - r + 1}.$$

□

The algorithm for routing edge-disjoint paths in a multibutterfly is nearly identical to the algorithm described in Section 4 for routing node-disjoint paths. First, each node that has at least one path to extend in either the up (or down) direction sends a proposal to each of his output neighbors in the up (down) direction. Then, every output node that receives at most r proposals sends back acceptances to all of those proposals. (Notice that this step limits the number of paths passing through a node to at most r.) Finally, each node that receives enough acceptances to extend *all* of its paths does so. In a network with an (α, δ) r-neighbors property, a constant fraction of the paths on each level are extended at each step. Thus, the time to extend a set of N paths from one level to the next is $O(\log N)$, and the total time to route a set of N paths from the inputs to the outputs is $O(\log^2 N)$. As in Section 4, this time can be improved to $O(\log N)$ using place-holders and cancellation signals.

Note that for $r \approx \sqrt{d}$, only $\beta \approx \sqrt{d}$ expansion is required in order to have an (α, δ) r-unshared neighbors property, where $\alpha > 0$ and $\delta > 0$. Since an algorithm for finding edge-disjoint paths can be converted to an algorithm for finding node-disjoint paths by replacing each degree-$2d$ node with a $2d \times 2d$ complete bipartite graph, the algorithm of this section reduces the expansion required for finding either edge- or node-disjoint paths from $\beta > d/2$ to $\beta \approx \sqrt{d}$. The difference is important because explicit constructions of expander graphs are known for $\beta > \sqrt{d}$ [13], but not for $\beta > d/2$. The algorithms for tolerating faults in [17] and the algorithms for routing paths in a nonblocking fashion in this paper still seem to require $\beta > d/2$. Recently, however, Pippenger has shown how to perform all of these tasks using only expansion $\beta > 1$ [31].

In order to use this routing algorithm in network R, we must make one modification. The paths from the sources do not necessarily start on level $-2\log N$ of the first part. In fact as many as four paths may start at any node in the first part. (Recall that sources in the second, third, and fourth parts start their paths in the first part.) Thus, the routing algorithm must be modified so that a node in the first part sends acceptances to the nodes at the previous level only if it receives at most $r - 4$ proposals. The impact on the performance of the algorithm will be negligible if r is large relative to 4.

6. Acknowledgments. The authors thank Nick Pippenger for suggesting that a nonblocking network might be constructed by treating busy switches as if they were faulty, and Ron Greenberg and an anonymous referee for suggesting many improvements to the paper.

REFERENCES

[1] W. A. Aiello, F. T. Leighton, B. M. Maggs, and M. Newman, *Fast algorithms for bit-serial routing on a hypercube*, Mathematical Systems Theory, 24 (1991), pp. 253–271.

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

[3] L. A. Bassalygo and M. S. Pinsker, *Complexity of an optimum nonblocking switching network without reconnections*, Problems of Information Transmission, 9 (1974), pp. 64–66.
[4] ———, *Asymptotically optimal networks for generalized rearrangeable switching and generalized switching without rearrangement*, Problemy Peredachi Informatsii, 16 (1980), pp. 94–98.
[5] B. Beizer, *The analysis and synthesis of signal switching networks*, in Proceedings of the Symposium on Mathematical Theory of Automata, Brooklyn, NY, 1962, Brooklyn Polytechnic Institute, pp. 563–576.
[6] V. E. Beneš, *Optimal rearrangeable multistage connecting networks*, Bell System Technical Journal, 43 (1964), pp. 1641–1656.
[7] D. G. Cantor, *On construction of non-blocking switching networks*, in Proceedings of the Symposium on Computer Communication Networks and Teletraffic, Brooklyn, NY, 1972, Brooklyn Polytechnic Institute, pp. 253–255.
[8] D. Dolev, C. Dwork, N. Pippenger, and A. Widgerson, *Superconcentrators, generalizers and generalized connectors with limited depth*, in Proceedings of the 15th Annual ACM Symposium on Theory of Computing, Apr. 1983, pp. 42–51.
[9] P. Feldman, J. Friedman, and N. Pippenger, *Non-blocking networks*, in Proceedings of the 18th Annual ACM Symposium on Theory of Computing, May 1986, pp. 247–254.
[10] ———, *Wide-sense nonblocking networks*, SIAM Journal of Discrete Mathematics, 1 (1988), pp. 158–173.
[11] J. Friedman, *A lower bound on strictly non-blocking networks*, Combinatorica, 8 (1988), pp. 185–188.
[12] A. V. Goldberg, B. M. Maggs, and S. A. Plotkin, *A parallel algorithm for reconfiguring a multibutterfly network with faulty switches*, IEEE Transactions on Computers, 43 (1994), pp. 321–326.
[13] N. Kahale, *Better expansion for Ramanujan graphs*, in Proceedings of the 32nd Annual Symposium on Foundations of Computer Science, IEEE Computer Society Press, Oct. 1991, pp. 398–404.
[14] C. P. Kruskal and M. Snir, *A unified theory of interconnection network structure*, Theoretical Computer Science, 48 (1986), pp. 75–94.
[15] F. T. Leighton, *Parallel computation using meshes of trees*, in 1983 Workshop on Graph-Theoretic Concepts in Computer Science, Linz, 1984, Trauner Verlag, pp. 200–218.
[16] ———, *Introduction to Parallel Algorithms and Architectures: Arrays • Trees • Hypercubes*, Morgan Kaufmann, San Mateo, CA, 1992.
[17] 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 (1992), pp. 578–587.
[18] T. Leighton, C. E. Leiserson, and D. Kravets, *Theory of parallel and VLSI computation*, Research Seminar Series Report MIT/LCS/RSS 8, MIT Laboratory for Computer Science, May 1990.
[19] T. Leighton, D. Lisinski, and B. Maggs, *Empirical evaluation of randomly-wired multistage networks*, in Proceedings of the 1990 IEEE International Conference on Computer Design: VLSI in Computers & Processors, IEEE Computer Society Press, Sept. 1990, pp. 380–385.
[20] T. Leighton and G. Plaxton, *A (fairly) simple circuit that (usually) sorts*, in Proceedings of the 31st Annual Symposium on Foundations of Computer Science, IEEE Computer Society Press, Oct. 1990, pp. 264–274.
[21] C. E. Leiserson, *Fat-trees: universal networks for hardware-efficient supercomputing*, IEEE Transactions on Computers, C–34 (1985), pp. 892–901.
[22] G. Lin and N. Pippenger, *Parallel algorithms for routing in non-blocking networks*, in Proceedings of the 3rd Annual ACM Symposium on Parallel Algorithms and Architectures, July 1991, pp. 272–277.
[23] G. A. Margulis, *Explicit constructions of concentrators*, Problems of Information Transmission, 9 (1973), pp. 325–332.
[24] G. M. Masson and B. W. Jordan, Jr., *Generalized multi-stage connection networks*, Networks, 2 (1972), pp. 191–209.
[25] D. Nassimi and S. Sahni, *Parallel permutation and sorting algorithms and a new generalized connection network*, Journal of the ACM, 29 (1982), pp. 642–667.
[26] Y. P. Ofman, *A universal automaton*, Transactions of the Moscow Mathematical Society, 14 (1965), pp. 186–199.
[27] D. Peleg and E. Upfal, *Constructing disjoint paths on expander graphs*, in Proceedings of the 19th Annual ACM Symposium on Theory of Computing, May 1987, pp. 264–273.
[28] N. Pippenger, *The complexity theory of switching networks*, PhD thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, Cambridge,

MA, 1973.

[29] ———, *On rearrangeable and nonblocking switching networks*, Journal of Computer and System Sciences, 17 (1978), pp. 145–162.

[30] ———, *Telephone switching networks*, in Proceedings of Symposia in Applied Mathematics, vol. 26, 1982, pp. 101–133.

[31] ———, *Self-routing superconcentrators*, in Proceedings of the 25th Annual ACM Symposium on the Theory of Computing, May 1993, pp. 355–361.

[32] N. Pippenger and L. G. Valiant, *Shifting graphs and their applications*, Journal of the ACM, 23 (1976), pp. 423–432.

[33] N. Pippenger and A. C. Yao, *Rearrangeable networks with limited depth*, SIAM Journal of Algebraic and Discrete Methods, 3 (1982), pp. 411–417.

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

[35] C. E. Shannon, *Memory requirements in a telephone exchange*, Bell System Technical Journal, 29 (1950), pp. 343–349.

[36] J. S. Turner, *Practical wide-sense nonblocking generalized connectors*, Technical Report WUCS–88–29, Department of Computer Science, Washington University, St. Louis, MO, 1988.

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

[38] A. Waksman, *A permutation network*, Journal of the ACM, 15 (1968), pp. 159–163.