Work-Preserving Emulations of Fixed-Connection Networks

R.R. Koch, F.T. Leighton, B.M. Maggs,
S.B. Rao, A.L. Rosenberg and E.J. Schwabe

Computer and Information Science Department
University of Massachusetts

COINS Technical Report 90-114

Work-Preserving Emulations of Fixed-Connection Networks

Richard R. Koch[1]
F. T. Leighton[2,3]
Bruce M. Maggs[3]
Satish B. Rao[4]
Arnold L. Rosenberg[5]
Eric J. Schwabe[2,3]

[1]AT & T Bell Laboratories
Holmdel, New Jersey 07733

[2]Mathematics Department and
[3]Laboratory for Computer Science
Massachusetts Institute of Technology
Cambridge, Massachusetts 02139

[4]Aiken Computation Laboratory
Harvard University
Cambridge, Massachusetts 02138

[5]Department of Computer Science
University of Massachusetts
Amherst, Massachusetts 01003

Abstract

In this paper, we study the problem of emulating T_G steps of an N_G-node guest network, G, on an N_H-node host network, H. We call an emulation *work-preserving* if the time required by the host, T_H, is $O(T_G N_G / N_H)$, because then both the guest and host networks perform the same total work (i.e., processor-time product), $\Theta(T_G N_G)$, to within a constant factor. We say that an emulation occurs in *real-time* if $T_H = O(T_G)$, because then the host emulates the guest with constant slowdown. Although many isolated emulation results have been proved for specific networks in the past, and measures such as dilation and congestion were known to be important, the field has lacked a model within which general results and meaningful lower bounds can be proved. We attempt to provide such a model, along with corresponding general techniques and specific results in this paper. Some of the more interesting and diverse conseqeuences of this work include:

1. a proof that a linear array can emulate a (much larger) butterfly in a work-preserving fashion, but that a butterfly cannot emulate an expander (of any size) in a work-preserving fashion,
2. a real-time work-preserving emulation of a butterfly on a shuffle-exchange graph, and vice versa,
3. a proof that a mesh can be emulated in real time in a work-preserving fashion on a butterfly, even though any $O(1)$-to-1 embedding of a mesh in a butterfly has dilation $\Omega(\log N)$, and
4. simple $O(N^2/\log^2 N)$-area and $O(N^{3/2}/\log^{3/2} N)$-volume layouts for the N-node shuffle-exchange graph.

This research was supported by the Defense Advanced Research Projects Agency under Contract N00014-87-K-825, the Office of Naval Research under Contract N00014-86-K-0593, the Air Force under Contract OSR-86-0076, and the Army under Contract DAAL-03-86-K-0171. Tom Leighton is supported by an NSF Presidential Young Investigator Award with matching funds provided by IBM. Eric Schwabe is supported by an NSF Graduate Fellowship. Arnold Rosenberg is supported by NSF Grant CCR-88-12567.

1 Introduction

In this paper, we study the problem of emulating an N_G-node *guest* network $G = (V_G, E_G)$ on an N_H-node *host* network $H = (V_H, E_H)$ where $N_H \leq N_G$. Our goal is to emulate T_G steps of any computation on G in $T_H = ST_G$ steps on H where S (the *slowdown* of the emulation) is as small as possible.

The slowdown of the emulation must always be at least as large as N_G/N_H since G has N_G/N_H times as many processors as does H. If $S = O(N_G/N_H)$, then we say that the emulation is *work-preserving* because then the total *work* (i.e., the processor-time product) performed by the emulating network ($W_H = T_H N_H$) is within a constant factor of the work performed by the guest network ($W_G = T_G N_G$). Such emulations achieve optimal speedup (to within a constant factor) over sequential emulations of G since they use N_H processors to solve a problem $\Theta(N_H)$ times faster than is possible with a single processor.

More generally, we say that there is a *work-preserving emulation* of a class of networks $\mathcal{G}$ by a class of networks $\mathcal{H}$ with slowdown $S(N)$ if for every N and T, any N-node network in $\mathcal{H}$ can emulate T steps of any $N \cdot S(N)$-node network in $\mathcal{G}$ in $O(T \cdot S(N))$ steps. In the special case that $S(N) = O(1)$, we say that the emulation occurs in *real time*. Real-time emulations are the hardest to obtain since we require the host network to emulate a guest network of the same size with constant slowdown.

An efficient emulation scheme can often be devised by finding a good embedding of the guest network into the host. By an embedding of a graph G into a graph H, we mean a mapping $\phi : G \rightarrow H$ that takes the nodes of G to the nodes of H and the edges of G to paths in H. The *dilation* of an embedding is the length of the longest path $\phi(e)$ corresponding to an edge e of G. The *congestion* of an embedding is the largest number of paths $\phi(e)$ crossing a single edge of H. The *load* of an embedding is the maximum number of nodes of G mapped to a single node of H. In a one-to-one embedding, the load is 1. Throughout the paper we will make use of the fact that if there is an embedding of G in H with congestion c, dilation d, and load l, then there is an emulation of G by H with slowdown $O(l+c+d)$. This fact follows from the proof in [14] that for any set of packets whose paths have congestion c and dilation d, there is a schedule of length $O(c+d)$ in which at most one packet traverses each edge at each step. When H is an array, tree, butterfly, or shuffle-exchange graph, the schedule can be computed on-line

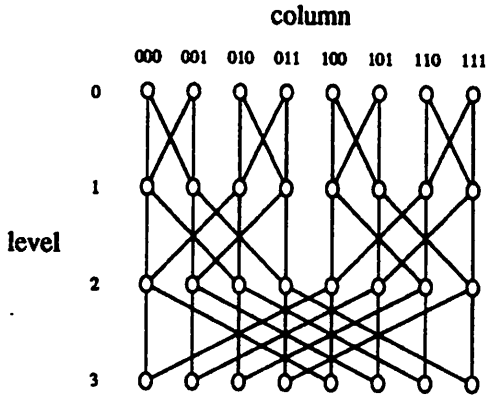

Figure 1: An 8-input butterfly without wraparound.

using an algorithm that works for all leveled networks [14, 12].

As a simple example, let $\mathcal{G}$ be the class of linear arrays, and $\mathcal{H}$ be the class of all bounded-degree connected networks. It is well known [20] that an N-node linear array can be embedded one-to-one in any connected bounded-degree N-node network with constant dilation and congestion. Hence any N-node bounded-degree connected network H can emulate any N-node linear array with constant slowdown, and thus there is a real-time emulation of the class $\mathcal{G}$ by the class $\mathcal{H}$.

As another simple example, consider the more interesting problem of emulating a butterfly on a linear array. We will prove that the class of butterflies cannot be emulated in real time by the class of linear arrays. (This should come as no surprise, although the proof is not entirely trivial.) However, there is a simple work-preserving emulation of the class of butterflies by the class of linear arrays with slowdown $O(2^N)$.

The M-node butterfly, where $M = r2^r$, has nodes consisting of all ordered pairs $\langle l, C\rangle$, where the *level* l is taken from the set $\{0, \ldots, r-1\}$ and the *column* C is an r-bit string. Node $\langle l, c_r \cdots c_{l+1} \cdots c_1\rangle$ is connected to node $\langle l+1 \bmod r, c_r \cdots c_{l+1} \cdots c_1\rangle$ by a *straight* edge, and to node $\langle l+1 \bmod r, c_r \cdots \overline{c_{l+1}} \cdots c_1\rangle$ by a *cross* edge. The butterfly *without wraparound* is defined similarly, but with $M = (r+1)2^r$ and the mod r removed from the edge definitions. In the butterfly without wraparound, the nodes in level 0 are called the *inputs* and the nodes in level r are called the *outputs*; note

that in the regular butterfly, the inputs and the outputs are identified into a single level. Furthermore, each of these networks can be embedded into the other with constant load, dilation and congestion, so that each can perform a real-time emulation of the other.

Given an N-node linear array, by mapping the 2^N nodes of the form $\langle l, C\rangle$ (where $C \in \{0,1\}^{N-1}$) to the lth node of the linear array, an N-node linear array can emulate an $N2^{N-1}$-node butterfly without wraparound with 2^{N-1} slowdown. A linear array can therefore also perform a work-preserving emulation of a regular butterfly with slowdown $O(2^N)$.

Seeing this elementary example, one is tempted to ask if there are faster work-preserving emulations of a butterfly on a linear array. In other words, can a linear array emulate a smaller butterfly (perhaps larger by only a polynomial factor) in a work-preserving fashion? Although the proof is not obvious, the answer is no. There is no work-preserving emulation of the class of butterflies by the class of linear arrays with polynomial slowdown. Any such emulation requires exponential slowdown. Alternatively, we might wonder if a linear array can emulate any bounded-degree network in a work-preserving fashion given enough slowdown. Again, the answer is no. Although the linear array can emulate a butterfly in a work-preserving fashion, it cannot emulate any expander, no matter how much blowup is allowed. In fact, by combining these results we can conclude that even a butterfly is not sufficiently powerful to emulate an expander in a work-preserving fashion.

We also consider emulations that are not work-preserving. Such emulations are (by definition) inefficient, and we define the inefficiency of such an emulation to be $I = W_H/W_G$. In these terms, an emulation is work-preserving if it has constant inefficiency. Many of our bounds will reflect tradeoffs between slowdown and inefficiency. In general,

$$I = \frac{S}{C}$$

where $C = N_G/N_H$ is the *contraction* of an emulation.

1.1 The motivation

There are several good reasons for studying the problem of emulating one network on another in a work-preserving fashion. First, this kind of analysis gives us an excellent means by which to compare the computational power

of one network relative to that of another. More importantly, it gives us an automatic way to compile and run algorithms designed for one kind of parallel architecture without loss of efficiency on another. This is provided, of course, that the ratio of the size of the problem to the size of the machine is large enough. For example, we have already seen that a small linear array (which has a very simple structure) is just as efficient in terms of work as a very large butterfly (which has a more complicated structure).

More generally, the study of work-preserving emulations lies at the heart of efficient parallel computing. Indeed, one of the central problems in efficient parallel computing is the task of mapping a collection of processes linked by precedence and/or communication constraints onto the processors and routing network of a parallel machine so that

1. the processing load imposed on the processors is balanced,
2. the communication between processors can be handled efficiently, and
3. the computation and communication can be scheduled so that the necessary inputs for a process are available where and when the process is scheduled to be computed.

In other words, we would like to schedule the communication and computation in a way that takes maximum advantage of the available hardware to minimize the completion time of the job.

In general, we can model the computation to be performed by a DAG. Each node of the DAG represents a process and each directed edge (u, v) represents a communication that must take place between processes u and v. Typically, this communication represents data output from u after u is completed which is to be input to v before v is started. The parallel machine can be modeled as an network. The nodes of the network correspond to processors, and the edges correspond to communication links between processors (and/or their associated memories). To implement the computation on a parallel machine, we must construct a schedule that specifies which processor executes each process, and how the outputs of these processes are communicated.

In many applications, the DAG possesses a very natural structure. For example, typical DAGs encountered in practice are derivatives of a binary tree, array, butterfly, or shuffle-exchange graph. This is often due to the fact

that the DAG is associated with an algorithm whose inherent underlying structure is a tree or array (as is the case for many problems in numerical analysis and linear algebra) or a butterfly or shuffle-exchange graph (as is the case for Fourier Transform and data manipulation problems). Alternatively, it could be that the DAG was constructed from an algorithm specifically designed for use on one of these common parallel architectures.

Similarly, parallel networks also tend to be very naturally structured and typically are configured as trees, arrays, butterflies, and the like. Hence, the mapping problem often consists of emulating T_G steps of one N_G-node network (represented as a DAG of depth T_G in which each level consists of a copy of G) on an N_H-node network with a different structure. Ideally, we would like to perform the computation in $O(T_G N_G / N_H)$ steps, which is precisely the problem of finding a work-preserving emulation of one network on another.

In practice, the guest network can be substantially larger than the host network. For example, it is not uncommon for a parallel machine with between 8 and 256 processors to be emulating array-based computations involving hundreds of thousands of data points. In such examples, even work-preserving emulations with exponential slowdown may be within the scope of practicality. Indeed, the most important feature of the computation is that it be work-preserving.

1.2 A closer look at the computational model

If we can find an embedding of a graph G into a graph H with constant dilation, congestion, and load, then it is fairly clear that H can emulate G with constant slowdown. Is the converse true? Somewhat surprisingly, it is not. For example, Bhatt, Chung, Hong, Leighton and Rosenberg [3] proved that any embedding of an N-node mesh into an N-node butterfly with constant load requires dilation $\Omega(\log N)$, the worst possible. At first glance, it might seem that this result implies that any emulation of an N-node mesh by and N-node butterfly must have slowdown at least $\Omega(\log N)$. However, we show that an N-node butterfly can emulate T-steps of an N-node mesh in $O(T)$ steps.

In order to understand how such a contradictory result is possible, we need to take a closer look at what it means to emulate T_G steps of one network in T_H steps on another. We model the emulation of the guest by

the host as a pebbling process on a DAG, Γ. In particular, Γ consists of $T_G + 1$ levels, one for each guest time step t, where $0 \leq t \leq T_G$. (Level 0 corresponds to the initial state of the guest.) On level t, there is a node (v,t) for every node, v, of G, and a node (e,t) for every edge, e, of G. Node (v,t) represents the state of guest processor v at the end of step t, while node (e,t) represents the data sent across guest edge e during step t. In addition, for $t > 0$, there are directed edges in Γ into node (v,t) from nodes $(v,t-1)$ and $(e_1,t-1),(e_2,t-1),\ldots,(e_k,t-1)$, where $e_1,e_2,\ldots,e_k$ are the edges into v. For each edge e leaving v in G, there are edges in Γ into (e,t) from the same nodes, $(v,t-1)$ and $(e_1,t-1)\ldots(e_k,t-1)$. The goal of the host is to create a pebble for each node in Γ. We call the pebbles for DAG nodes of the form (v,t) *node pebbles* and those for nodes of the form (e,t) *edge pebbles*. At each step in the emulation, a host node may perform the following operations.

1. Copy a single edge pebble that it contains.
2. Send a single edge pebble to a neighbor.
3. Create a node or edge pebble for a node in Γ if it contains pebbles for all of that node's predecessors in Γ.

The trick that makes it possible for a butterfly to emulate a mesh in real-time is to allow the host to create more than one pebble for each DAG node. (Note that in the emulation schemes based on embedding G in H, the host creates exactly one pebble for each DAG node.) Creating several pebbles for a node corresponds to performing the same computation more than once. By allowing redundant computation, we dramatically increase the number of ways that the host can emulate the guest. This makes it more likely that we can find a computation that can be efficiently emulated on some host network H, but it also makes the task of proving lower bounds more difficult. Indeed, at the very least, we must choose T_G to be large since by allowing redundant computations of pebbles, any $O(1)$ steps of any N-node bounded-degree graph G can be computed in $O(1)$ steps on any N-node graph H. (This is because if $T = O(1)$, then any output pebble can only depend on $O(1)$ input pebbles, which can be redundantly computed locally since every node of H is assumed to have access to all input pebbles.)

Note that when we prove a lower bound on the ability of a graph H to emulate a graph G, it does not necessarily mean that H cannot effectively

compute the same result as does G (possibly by using a different algorithm, for example). Rather, we are proving lower bounds on the ability of H to perform the same step-by-step computations as G when G is used in a general purpose way. Hence the term *emulation*. We suspect that our pebbling model is probably the most general model in which we could hope to prove lower bounds.

1.3 Network definitions

We now formally define the networks whose properties we will be studying (aside from the butterfly, which has already been defined).

The N-node *q-dimensional mesh* has vertex set $\{1, \ldots, \sqrt[q]{N}\}^q$. For each dimension i, node $(x_1, \ldots, x_i, \ldots, x_q)$ is connected to node $(x_1, \ldots, x_i - 1, \ldots, x_q)$ unless $x_i = 1$, and to node $(x_1, \ldots, x_i + 1, \ldots, x_q)$ unless $x_i = \sqrt[q]{N}$.

The N-node *complete binary tree*, where $N = 2^h - 1$, has vertex set $\{1, \ldots, N\}$, where each vertex $i \leq (N-1)/2$ is connected to vertices $2i$ and $2i + 1$.

The nodes of the N-node *shuffle-exchange graph*, where $N = 2^n$, consist of all N-bit strings. Node $x_n x_{n-1} \cdots x_2 x_1$ is connected to nodes $x_{n-1} x_{n-2} \cdots x_1 x_n$ and $x_1 x_n \cdots x_3 x_2$ by *shuffle* edges, and to node $x_n \cdots \overline{x_1}$ by an *exchange* edge.

1.4 Our results

The technical portion of this paper is divided into five sections. We commence in Section 2 with some general techniques for establishing the existence or nonexistence of a work-preserving emulation. In Sections 3 through 6, we focus on the special case of emulations by arrays, complete binary trees, butterflies, and shuffle-exchange graphs, respectively.

In Section 2, we describe two general methods for proving lower bounds on the slowdown of a work-preserving emulation. The first method is based on dilation considerations and appears in Section 2.1. As an application of this method, we prove that any class of low diameter networks (such as complete binary trees) cannot be emulated in real time on any class of networks that has poor expansion properties (such as arrays of bounded dimension). The second method is based on congestion properties and is presented in

Section 2.2. Here we describe a general method for proving that a work-preserving emulation requires a large amount of time, or that it is impossible altogether. As an example, we prove that any work-preserving emulation of a butterfly on an array of bounded dimension requires exponential time, and that it is not possible to emulate an expander on a butterfly in work-preserving fashion. These results provide a curious contrast between the power of a linear array, butterfly, and an expander. By most standards, it would seem that a butterfly is closer in power to an expander than it is to a linear array. Yet a linear array can emulate a butterfly in a work-preserving fashion, but a butterfly (or most any non-expander) cannot emulate an expander in a work-preserving fashion.

In Section 3, we prove tight bounds on the slowdown required for an array to emulate a tree, array or butterfly.

In Section 4, we prove that there is a work-preserving emulation of bounded-degree trees by complete binary trees with $O(\log \log N)$ slowdown. We also give evidence, but no proof, that there is no corresponding real-time emulation for this class. (Proving that a complete binary tree cannot emulate a complete ternary tree in real-time is one of several challenging questions left open in this paper.)

Section 5 explores emulations by butterfly networks. We begin by describing a real-time emulation of the shuffle-exchange graph on the butterfly. In Section 6, we prove the reverse, namely, that there is a real-time emulation of the butterfly on the shuffle-exchange graph. Taken together, these results resolve the long open question of whether the butterfly and shuffle-exchange graph are computationally equivalent. Next, we show that an N-node butterfly can emulate an N-node mesh in real-time. This result is interesting because any one-to-one embedding of an array (with dimension 2 or more) in a butterfly requires $\Omega(\log N)$ dilation [3], which suggests that any emulation must require slowdown $\Omega(\log N)$. The result takes on added significance given the fact that many parallel numerical algorithms are array-based while several parallel machines are butterfly-based. Finally, we describe a simple constant-congestion embedding of an N-node shuffle-exchange graph in an N-node butterfly. This result can be used to provide an elementary proof that the N-node shuffle-exchange graph can be laid out in $O(N^2/\log^2 N)$ area and in $O(N^{3/2}/\log^{3/2} N)$ volume. Both results are optimal. The area bound was known previously [9], but the proof was much more difficult (as were the proofs for several suboptimal layouts for the shuffle-exchange graph

[8, 11, 13, 21]). The 3-d layout bound is new and was not obtainable by any of the previous approaches to the 2-d layout problem.

The real-time emulation of the butterfly by the shuffle-exchange graph yields several new efficient algorithms for the shuffle-exchange graph. For example, we now know that a shuffle-exchange graph can sort N numbers in $O(\log N)$ steps with high probability. Previously, such an algorithm was known for the butterfly [14, 17, 19] but that algorithm made crucial use of the recursive structure of the butterfly, a structure not present in a shuffle-exchange graph. The emulation also yields a real-time emulation of bounded-degree arrays by the shuffle-exchange graph.

1.5 Previous work

The notion of work-preserving emulations was previously studied by Fishburn and Finkel [6]. They examined emulations in which both the guest and host are drawn from the same class of networks. Several of their results are included in this paper for completeness.

There has been a great deal of previous work on graph embeddings with the intent of showing that one network can or can't emulate another network efficiently [3, 4, 5, 7, 14, 18]. Many of the results were positive and proved things like "all N-node binary trees can be emulated in constant time on an N-node hypercube." There were also some negative results, but because of the lack of a good model, their significance is now less clear. For example, even though an embedding of a mesh into a butterfly requires dilation $\Omega(\log N)$, we now find that a butterfly can emulate a mesh with constant slowdown.

Work-preserving PRAM algorithms have previously been studied [10, 15] and served to motivate this work. Related problems of scheduling computations on fixed-connection networks have also been studied [16].

2 Lower bounds

In this section we present lower bounds on slowdown and inefficiency. Loosely speaking, these lower bounds apply when the guest graph expands faster than the host graph. The first lower bound can be used to show that any emulation of a complete binary tree by a linear array has slowdown $\Omega(N_H/\log N_H)$. The

second can be used to show that a butterfly cannot perform a work-preserving emulation of an expander graph, that any work-preserving emulation of a butterfly by a linear array H requires slowdown at least $2^{\Omega(N_H)}$, and that any work-preserving emulation of a $(k+1)$-dimensional mesh by a k-dimensional mesh H requires slowdown at least $\Omega(N_H^{1/k})$. All of these lower bounds on slowdown are tight up to constant factors in the Ω notation.

For the sake of proving lower bounds, we simplify the pebbling model somewhat. As described in Section 1, the goal of the host is to pebble a DAG, Γ, with $T_G + 1$ levels, but now each level t contains only the nodes of the form (v,t), and none of the form (e,t). The edges into (v,t) come from $(v,t-1)$, and $(v_1,t-1)$, $(v_2,t-1)$, ..., $(v_k,t-1)$, where $v_1,\ldots,v_k$ are the neighbors of v in G. As before, (v,t) represents the state of processor v at time step t. At each step, a host node may make a copy of a pebble that it creates, send a pebble to one of its neighbors, or create a pebble for a node of Γ provided that it contains pebbles for all of that node's predeccessors in Γ. Although it is not entirely realistic to allow a host edge to pass the entire state of a guest node in a single time step, the lower bounds that we prove in this simplified model hold in the more realistic model as well.

Before proving the lower bounds, we need to introduce some notation. For an undirected graph $G = (V,E)$, let $\delta(u,v)$ be the length (number of edges) of the shortest path between nodes u and v in G. Let $B_G(u,i)$ be the set of nodes within a distance i of u in G, i.e., $B_G(u,i) = \{v \in V | \delta(u,v) \leq i\}$, and let $b_G(u,i) = |B_G(u,i)|$. We call b_G the *growth function* of G.

2.1 Distance-based lower bound

The following theorem shows that if the guest graph grows faster than the host graph, then any emulation of the guest by the host must be slow.

Theorem 2.1 *Let $H = (V_H, E_H)$ be an N_H-node host graph and $G = (V_G, E_G)$ be an N_G-node guest graph, and suppose that there are integers τ_H and τ_G such that*

$$\max_{u \in V_H} \sum_{i=1}^{\tau_H} b_H(u,i) < \min_{v \in V_G} \sum_{j=1}^{\tau_G} b_G(v,j).$$

Then any emulation of $T_G \geq \tau_G$ steps of G by H has slowdown

$$S > \frac{\tau_H + 1}{2\tau_G}.$$

Proof: Our strategy will be to find a sequence of $\Omega(T_G/\tau_G)$ pebbles created by H such that no two are created within τ_H host time steps of each other. Such a sequence implies that the slowdown $S = T_H/T_G$ is at least $\Omega(\tau_H/\tau_G)$.

We start the sequence with the last pebble created by H. Suppose that at time T_H some node $u_0 \in V_H$ creates a pebble for DAG node (v_0, t_0), where $t_0 = T_G$. The pebble for (v_0, t_0) cannot be created by H until pebbles for all of its predecessors in the DAG are created. In particular, there are at least $\sum_{j=1}^{\tau_G} b_G(v_0, j)$ precedessors for time steps $t_0 - \tau_G$ through $t_0 - 1$. We want to show that the pebble for at least one of these predecessors must have been created by the host graph before time $T_H - \tau_H$. The pebble for every predecessor of (v_0, t_0) that is created at distance i from u_0 in H must be created at or before time $T_H - i$. Thus at most $\sum_{i=1}^{\tau_H} b_H(u_0, i)$ pebbles for predecessors of (v_0, t_0) are created by H between time steps $T_H - \tau_H$ and $T_H - 1$. Since $\max_{u \in V_H} \sum_{i=1}^{\tau_H} b_H(u, i) < \min_{v \in V_G} \sum_{j=1}^{\tau_G} b_G(v, j)$, the pebble for some predecessor (v_1, t_1), $t_1 \geq T_G - \tau_G$, must be created by the host graph at or before time $T_H - (\tau_H + 1)$.

We can repeat the argument to find a pebble for a predecessor (v_2, t_2), $t_2 \geq T_G - 2\tau_G$, of (v_1, t_1) that must be created by the host at or before time $T_H - 2(\tau_H + 1)$, and so on. Eventually we obtain a pebble (v_k, t_k) such that $\tau_G > t_k \geq T_G - k\tau_G$. This pebble must be created by the host at or before time $T_H - k(\tau_H + 1)$. We assume that input pebbles are created at host time step 0, and that the emulation begins with time step 1. Thus, $T_H - k(\tau_H + 1) \geq 0$. Combining these inequalities, we have

$$T_H/T_G > \frac{\tau_H + 1}{2\tau_G}$$

for $T_G \geq \tau_G$. □

Corollary 2.2 *Any such emulation has inefficiency*

$$I > \Omega\left(\frac{\tau_H N_H}{\tau_G N_G}\right).$$

Corollary 2.3 *For fixed k, any emulation of a complete binary tree, G, by a k-dimensional mesh, H, has slowdown at least $\Omega\left((N_G/\log^k N_G)^{1/(k+1)}\right)$.*

Proof: Apply Theorem 2.1 with $\tau_G = \Theta(\log N_G)$, and $\tau_H = \Theta\left((N_G \log N_G)^{1/(k+1)}\right)$. □

2.2 Congestion-based lower bound

The second lower bound requires a little more notation. Let $G = (V, E)$ be an undirected graph as before. For a set $U \subseteq V$, we define the *i-neighborhood* of U, $\mathcal{N}_i(U)$, to be the set of nodes not in U, but within a distance i of some node in U, $\mathcal{N}_i(U) = \cup_{u \in U} B_G(u, i) - U$. We define an *$(R, f(R))$-decomposition* of G to be a partition of V into $|V|/R$ sets of nodes (regions) such that each contains R nodes and has a 1-neighborhood of size at most $f(R)$.

The last graph parameter that we need, z_G, is best described in terms of a simple game. The player starts by choosing a nodes of a connected graph G and placing them in a bag. The player is given a collection of εa, $0 \leq \varepsilon < 1$, tokens to play with. The game is played in rounds, each consisting of two steps. In the first step, all of the neighbors of the nodes in the bag are added to the bag. In the second step, the player may exchange tokens for nodes in the bag on a one-for-one basis. Let X_i be the set of nodes in the bag at the end of round i, and let Y_i be the set of nodes removed in the second step of round i. Then X_i is given by the recurrence $X_i = X_{i-1} + \mathcal{N}_1(X_{i-1}) - Y_i$. The game ends when the number of nodes in the bag exceeds it capacity, c, at the end of a step, where $c < N_G$. If k is the number of rounds played, then $|X_i| \leq c$ for $i < k$, $|X_i| > c$ for $i = k$, and $\sum_{i=1}^{k} |Y_i| \leq \varepsilon a$. The goal is to play as many rounds as possible. Let $z_G(a, \varepsilon, c)$ be an upper bound that is non-increasing in a on the length of the longest possible game.

Theorem 2.4 *Suppose that $H = (V_H, E_H)$ is an N_H-node host graph with an $(R, f(R))$-decomposition, and that $G = (V_G, E_G)$ is an N_G-node guest graph. Let*

$$\beta = \max\left\{z_G\left(\frac{N_G}{4}, 0, \frac{3N_G}{4}\right), z_G\left(\frac{3N_G R}{8N_H}, \frac{1}{2}, \frac{N_G}{2}\right)\right\}.$$

Then for any emulation of G by H where $T_G > 3\beta$,

$$I \geq \min\left\{\frac{R}{32\beta f(R)}, \frac{N_H}{96R}\right\}.$$

Proof: The basic strategy is to show that either the host spends a lot of time passing pebbles across the perimeters of the regions in the $(R, f(R))$-decomposition, or the host spends a lot of time creating pebbles. We will break the T_G guest time steps into blocks of 3β consecutive steps and classify every block as either an *importer* or a *creator*. If a block is an importer, then many pebbles for the block cross region perimeters. If a block is a creator, then some region creates many pebbles for the block. If the majority of the blocks are importers, then the time required by the host to pass pebbles across the perimeters of the regions large. Otherwise, the time required to create the pebbles is large.

Before we can get started we need one more piece of notation. For each node v in G there is at least one pebble created by H for each guest time step t between 1 and T_G. The first pebble created for v for time t is called the *t-primary pebble* for v. For each value of t there are exactly N_G t-primary pebbles. The t-primary pebbles are ordered according to the order in which they are created by H, with ties broken arbitrarily. We call the first $3N_G/4$ t-primary pebbles the *t-early* pebbles and the last $3N_G/4$ the *t-late* pebbles.

We begin with the definition an importer block. Consider a block from step t to $t - 3\beta + 1$. The average number of t-early pebbles created by each of the N_H/R regions in the decomposition of H is at least $p = 3N_G R/4N_H$. We say that a region is *t-busy* if it creates at least $p/2$ t-early pebbles. We say that a t-early pebble is t-busy if it is created by a t-busy region. At least half of the t-early pebbles are t-busy. Thus, there are at least $3N_G/8$ t-busy pebbles. Suppose that a t-busy region creates $s \geq p/2$ t-busy pebbles. We say that the region is an *importer* if it imports at least $s/2$ pebbles for time steps between $t - 1$ and $t - 2\beta$. We say that a block is an importer if every t-busy region is an importer, or if some region imports at least $3N_G/16$ pebbles for time steps between $t - 1$ and $t - 2\beta$. In a importer block, a total of at least $3N_G/16$ pebbles for time steps between $t - 1$ and $t - 2\beta$ are imported by all of the regions.

If at least half of the $T_G/3\beta$ blocks are importers, then we can find a lower bound on inefficiency by computing the time required to import pebbles. In this case, the total number of pebbles imported by all of the importer blocks is at least $T_G N_G/32\beta$. The host time required to import these pebbles is at least $T_H \geq T_G N_G R/32\beta N_H f(R)$, because at each host time step, each of the

N_H/R regions can import at most $f(R)$ pebbles. In this case,

$$I \geq R/32\beta f(R).$$

As we shall see, if a block is not an importer then some region must create many pebbles for the block. Hence the name creator. In a creator block there must be some t-busy region $\mathcal{R}$ that creates $s \geq p/2$ t-busy pebbles but imports fewer than $s/2$ pebbles for time steps between $t-1$ and $t-2\beta$. The t-busy pebbles created by $\mathcal{R}$ cannot be created until pebbles for all of their predecessors in the pebble DAG are created. Since $z_G(s, 1/2, N_G/2) \leq z_G(p/2, 1/2, N_G/2) \leq \beta$, $\mathcal{R}$ imports at most $s/2$ pebbles for time steps between $t-1$ and $t - z_G(s, 1/2, N_G/2)$. Thus $\mathcal{R}$ must create at least $N_G/2$ pebbles for time step $t - z_G(s, 1/2, N_G/2)$. Furthermore, since $\mathcal{R}$ imports at most $3N_G/16$ pebbles for time steps between $t-1$ and $t-2\beta$, it must create at least $5N_G/16$ pebbles for every time step between $t - z_G(s, 1/2, N_G/2)$ and $t-2\beta$. For each of these time steps, at least $N_G/16$ of the pebbles are created for nodes whose $(t-2\beta)$-primary pebbles are $(t-2\beta)$-late pebbles. We call these pebbles the *descendant* pebbles.

We have chosen the descendant pebbles so that none are created by H until all of the descendant pebbles for previous blocks have been created. The early pebbles for all time steps at or before $t - 2\beta - z_G(N_G/4, 0, 3N_G/4)$ must be created before the $(t-2\beta)$-late pebbles because $3N_G/4$ nodes in G lie within a distance $z_G(N_G/4, 0, 3N_G/4)$ of the nodes corresponding to the first $N_G/4$ $(t-2\beta)$-primary pebbles. Since $z_G(N_G/4, 0, 3N_G/4) \leq \beta$, the early pebbles for previous blocks must be created before the $(t-2\beta)$-late pebbles. Furthermore, the $(t-2\beta)$-late pebbles must be created before the descendant pebbles, which in turn must be created before the t-busy pebbles for $\mathcal{R}$.

If at least half of the blocks are creators, then we can derive a lower bound on inefficiency by summing the time to create the descendant pebbles for each of the creator blocks. For each of $T_G/6\beta$ creator blocks, at least $\beta N_G/16$ descendant pebbles are created by a single region. The host time for each block is at least $\beta N_G/16R$. The host time for all of the creator blocks is at least $T_G N_G/96R$ and the inefficiency is at least

$$I \geq N_H/96R.$$

Combining the two cases proves the theorem. □

Corollary 2.5 *A k-dimensional mesh H cannot perform a work-preserving emulation of an expander graph G.*

Proof: Apply Theorem 2.4 with $R = \Theta((N_H \log N_H)^{k/(k+1)})$, $f(R) = O(R^{(k-1)/k})$, and $\beta = O(\log(N_H/R))$. The inefficiency is at least $I \geq \Omega((N_H/\log^k N_H)^{1/(k+1)})$. □

Corollary 2.6 *A butterfly network H cannot perform a work-preserving emulation of an expander graph G.*

Proof: Apply Theorem 2.4 with $R = \Theta(N_H \log\log N_H/\log N_H)$, $f(R) = O(R/\log R)$, and $\beta = O(\log(N_H/R))$. The inefficiency is at least $I \geq \Omega(\log N_H/\log\log N_H)$. □

Corollary 2.7 *Any work-preserving emulation of a butterfly G by a k-dimensional mesh H has slowdown at least* $2^{\Omega(N_H^{1/k})}$.

Proof: Apply Theorem 2.4 with $R = \Theta((N_H \log N_G)^{k/(k+1)})$, $f(R) = O(R^{(k-1)/k})$, and $\beta = O(\log N_G)$. The inefficiency is at least $I \geq \Omega((N_H/\log^k N_G)^{1/(k+1)})$.

The difficult part of this proof lies in showing that $\beta = O(\log N_G)$. Since $\beta = max\{z_G(N_G/4, 0, 3N_G/4), z_G(3N_G R/8N_H, 1/2, N_G/2)\}$, it suffices to show that for any $a > 0$, $z_G(a, 1/2, 3N_G/4) = O(\log N_G)$.

The key idea is to view the butterfly as a collection of overlapping complete binary trees. Let the guest graph, G, be an $n \log n$-node butterfly with each edge directed from level l to level $l + 1 \bmod \log n$. Then each node in the butterfly is the root of a complete binary tree of depth $\log n$. In any particular tree, T, no butterfly node appears more than once, except the root, which also appears as a leaf. We can extend T by attaching to each leaf a linear array of $\log n$ nodes. Every butterfly node appears in exactly one linear array. The a butterfly nodes initially in the bag are all tree roots. Since there are at most $a/2$ tokens to play with, at least $a/2$ of these roots are never removed from the bag. Henceforth, we shall restrict our attention to the set, F, of trees (with their attached linear arrays) rooted at these $a/2$ nodes. We will show that the trees in F grow so quickly that no matter how the $a/2$ tokens are spent, after $O(\log N_G)$ steps, at least $3N_G/4$ nodes are in the bag.

Before proceeding, let us introduce a little notation. For a tree T in F, let ϕ_i^T be the number of linear array nodes in the bag at the end of step

$2\log n + i$, and let Φ_t^T be the total amount of time spent in the bag by linear array nodes between steps $2\log n$ and $2\log n + t$, $\Phi_t^T = \sum_{i=0}^{t} \phi_i^T$. For the entire forest F, let $\Phi_t = \sum_{T \in F} \Phi_t^T$. Note that each butterfly node is counted $|F|$ times in Φ_t.

As the $a/2$ tokens are spent, they slow the growth of the trees in F, and we must account for the effect of each token. Consider a single tree T. If no tokens are spent, then after $2\log n$ steps, all of the nodes in T's linear arrays are in the bag. In this case, after $2\log n + t$ steps, $\Phi_t^T = tn\log n$. When a token is spent, it may delay that time at which some nodes enter the bag. For example, if a node, v, at depth l, $0 \leq l \leq \log n$, is removed from the bag at step t, then its children may be prevented from entering the bag at step $t+1$. (Of course, they may already be in the bag.) More generally, the descendants of v at depth $l+i$ may be prevented from entering the bag at step $t+i$. To account for the delay caused by removing v from the bag at step t, we attribute *damage* to the token spent to remove v. Since v has $n/2^l$ descendant linear arrays, each of which contains $\log n$ nodes, we attribute $n\log n/2^l$ damage to the token. If v is a linear array node, then $\log n + 1 \leq l \leq 2\log n$, and we attribute $2\log n - l + 1$ damage to the token.

The usefulness of our accounting system stems from the fact that whenever a linear array node, v, in T is not in the bag at the end of some time step $2\log n + t$, $t \geq 0$, at least one damage point accounts for it. If v is not in the bag at the end of step $2\log n + t$, then either it was removed from the bag during step $2\log n + t$, or its parent was not in the bag at the end of step $2\log n + t - 1$. By construction, the root of T is never removed from the bag. Thus, by induction, if a node is not in the bag at step t, then for some $l \geq 0$, its ancestor l levels higher in the tree was removed from the bag during step $2\log n + t - l$. But when this ancestor was removed, we allotted a damage point to the corresponding token.

Since the trees in F overlap, spending a single token does damage in many trees. Each node in the butterfly appears at most once as a tree root, twice at depth one, and so on. In general, a node appears at most 2^l times at depth l, up to a maximum of $|F|$ appearances. A node also appears at most $|F|$ times at each position in the linear array. Thus, the damage done to all trees in F by spending a single token is at most $(\log|F|+1)(n\log n)+|F|\log n(\log n+1)/2$. For $a/2$ tokens, the total is $O(\log|F|\, an\log n)$.

To prove that there must be at least $(3n\log n)/4$ many nodes in the bag within $O(\log n)$ steps, we show that for some $t \leq O(\log n)$, the average

number of nodes in the bag between steps $2\log n$ and $2\log n + t$, Φ_t/t, is at least $(3n\log n)/4$. Recall that if no tokens are spent, $\Phi_t = |F|\,tn\log n$. After subtracting for damage, the average number of nodes in the bag is at least $(|F|\,tn\log n - O(\log|F|\,an\log n))/t\,|F|$. Here we have divided by $|F|$ because a linear array node is counted $|F|$ times in Φ_t. For $t = \Theta(\log n)$, this average exceeds $3n\log n/4$. □

Corollary 2.8 *Any work-preserving emulation of a j-dimensional mesh G by a k-dimensional mesh H, $j > k$, has slowdown at least $\Omega(N_H^{(j-k)/k})$.*

Proof: Apply Theorem 2.4 with $R = \Theta((N_G^{1/j}N_H)^{k/(k+1)})$, $f(R) = O(R^{(k-1)/k})$, and $\beta = O(N_G^{1/j})$. The inefficiency is at least $I \geq \Omega((N_H^j/N_G^k)^{1/j(k+1)})$. □

3 Emulations by arrays

Although the arrays cannot perform real-time emulations of graphs with small diameter, we can show that they can perform work-preserving emulations of complete binary trees, other arrays, and butterflies. In each case, we find an embedding of the guest graph into the array with acceptable load, congestion, and dilation. The edges of the guest graph are emulated by routing packets between the nodes of the linear array. Observations 3.2 and 3.3 were proved by Fishburn and Finkel [6]. All of the following results can be shown to be tight by Corollaries 2.3, 2.7, and 2.8.

Observation 3.1 *An N-node k-dimensional mesh can perform a work-preserving emulation of an $N^{(k+1)/k}/\log N$-node complete binary tree.*

Proof: An $N^{(k+1)/k}/\log N$-node complete binary tree can be embedded in an N-node k-dimensional mesh with load $O(N^{1/k}/\log N)$, dilation $O(N^{1/k}/\log N)$, and congestion $O(N^{1/(k+1)})$. □

Observation 3.2 [6] *An N-node k-dimensional mesh can perform a work-preserving emulation of an $N^{j/k}$-node j-dimensional mesh, $j > k$.*

Proof: An $N^{j/k}$-node j-dimensional mesh can be embedded in an N-node k-dimensional mesh with load $N^{(j-k)/k}$, congestion $N^{(j-k)/k}$, and dilation 1. □

Observation 3.3 [6] *For any $M \geq N$, an N-node k-dimensional mesh can perform a work-preserving emulation of an M-node k-dimensional mesh.*

Proof: An M-node k-dimensional mesh can be embedded in an N-node k-dimensional mesh with load $O(M/N)$, dilation 1, and congestion $O((M/N)^{1-1/k})$. □

Observation 3.4 *An $N_H = n^k$-node k-dimensional mesh H can perform a work-preserving emulation of an $N_G = n2^n$-node butterfly graph G.*

Proof: An $n2^n$-node butterfly graph with 2^n rows and n columns can be embedded in a $N_H = n^k$-node k-dimensional mesh with load $O(2^n/n^{k-1})$, congestion $O(2^n/n^{k-1})$, and dilation $O(n)$. □

It is interesting to note that every connected network can perform a real-time emulation of a linear array. Hence, Observations 3.1 through 3.4 can be modified to hold for all connected networks.

4 Emulations by complete binary trees

4.1 Work-preserving emulations of larger complete binary trees

The following theorem extends a result of Fishburn and Finkel [6] by showing that a complete binary tree can emulate any larger complete binary tree in a work-preserving fashion.

Theorem 4.1 *For any $M \geq N$, an N-node complete binary tree can perform a work-preserving emulation of an M-node complete binary tree.*

Proof: An M-node complete binary tree can be embedded in an N-node complete binary tree with load $O(M/N)$, dilation 1 and congestion 1. □

4.2 Work-preserving emulations of bounded-degree trees

In this section, we show that any $N \log\log N$-node forest with maximum degree Δ can be embedded in an N-node complete binary tree with load $O(\Delta \log\log N)$, congestion $O(\Delta^2 \log\log N)$, and dilation $O(\log \Delta)$. As a corollary, there is a work-preserving emulation with slowdown $O(\log\log N)$ of the class of bounded-degree forests by the class of complete-binary trees.

In constructing the embedding, we use the following well-known weighted-separator lemma and its corollaries.

Lemma 4.2 *Suppose that $F = (V, E)$ is a forest where each vertex has been assigned some non-negative weight. Then it is possible to remove a single vertex from V so that the remaining vertices can be partitioned into two subforests F_1 and F_2 such that no edge connects a vertex in F_1 with a vertex in F_2, and F_1 and F_2 each contain at most $2/3$ of the total weight.*

Proof: Omitted.

Corollary 4.3 *By removing a single vertex, it is possible to partition a forest $F = (V, E)$ into two subforests each containing at most $2|V|/3$ vertices.*

Proof: Assign each vertex weight 1 and apply Lemma 4.2. □

Corollary 4.4 *By removing a set S of k vertices, it is possible to partition a forest $F = (V, E)$ into two subforests, F_1 and F_2, each containing at most $|V|(1 + (2/3)^k)/2$ vertices.*

Proof: Initially F_1 and F_2 are empty and a third set R contains all of the vertices. Iterate the following step k times. Apply Corollary 4.3 to split R into two subforests, then remove the smaller subforest from R and add it to the smaller of F_1 and F_2. At the end of each step, F_1 and F_2 differ in size by at most $|R|$. After k iterations, R contains at most $|V|\,(2/3)^k$ vertices. Add R to the smaller of the two sets. □

Corollary 4.5 *Suppose that $F = (V, E)$ is a forest where each vertex has been assigned some non-negative weight. Then it is possible remove a set S of k vertices such from V such that the remaining vertices can be partitioned*

into two subforests F_1 and F_2 such that no edge connects a vertex in F_1 with a vertex in F_2, and each contains at most $|V|(1+(2/3)^{(k-1)/2})/2$ vertices and at most $5/6$ of the total weight.

Proof: First apply Lemma 4.2 to partition the forest into two subforests L and R, each containing at most $2/3$ of the weight. Next, apply Corollary 4.4 to split L into L_1 and L_2, and R into R_1 and R_2. Let L_1 and R_1 have more weight than L_2 and R_2 respectively. Then both L_1 and R_1 have at most $2/3$ of the weight, and L_2 and R_2 have at most $1/6$. Let $F_1 = L_1 \cup R_2$ and $F_2 = L_2 \cup R_1$. □

With these tools in hand, we present the embedding.

Theorem 4.6 *An $N \log\log N$-node forest with maximum degree Δ can be embedded in an N-node complete binary tree with load $l = O(\Delta \log\log N)$, congestion $c = O(\Delta^2 \log\log N)$, and dilation $d = O(\log \Delta)$.*

Proof: The embedding begins by using Corollary 4.5 to find a set S of $k \in O(\log\log N)$ nodes that partitions the forest $F = (V, E)$ into two subforests, each containing at most $|V|(1 + 1/\log N)/2$ vertices. We embed S at the root of the binary tree and then recursively embed one of the subforests in the left subtree of the root, and the other in the right.

At levels below the root, we use Corollary 4.5 to simultaneously partition the vertices of the forest and the edges connecting the forest to vertices that are embedded higher in the binary tree. Let $F_i = (V_i, E_i)$ be a forest to be embedded in a subtree rooted at a level i node v_i in the binary tree. Let N_i be the number of edges connecting F_i to vertices embedded higher in the binary tree; N_i is the congestion of the binary tree edge connecting v_i to its parent. We assign each vertex of F_i a weight equal to the number of neighbors it has that are embedded higher in the binary tree. Using Corollary 4.5, we find a set S_i of k vertices that partitions F_i into two subforests, each of size at most $|V_i|(1+1/\log N)/2$, and each having at most $(5/6)N_i$ edges to vertices that are embedded higher in the tree. We embed the vertices of S_i at v_i and recursively embed one of the subforests in the left subtree of v_i, and the other in the right subtree.

To limit the dilation to some integer d, whenever i is a multiple of d we embed at v_i not only S_i but also all of the vertices in F_i that have at least one neighbor embedded somewhere higher in the binary tree.

We must now show how to choose d so that both the congestion and the load of the embedding are small. Consider any simple path from a level i node v_i in the binary tree to a level $i+d$ node, v_{i+d}, where i is a multiple of d. At level i, we embed a separator of size k and at most N_i other vertices that have at least one neighbor embedded higher in the tree. Since each of these vertices has at most Δ neighbors, $N_{i+1} \leq \Delta k + \Delta N_i$. At level $i+1$, we embed a separator of size k that partitions F_{i+1} into two subforests, each having at most $(5/6)N_{i+1}$ edges to vertices embedded higher in the binary tree. Thus, at level $i+2$, we have $N_{i+2} \leq (5/6)N_{i+1} + \Delta k$. In general, N_{i+j} is given by the recurrence

$$N_{i+j} \leq \begin{cases} \Delta k + \Delta N_i & j = 1 \\ (5/6)N_{i+j-1} + \Delta k & 1 < j \leq d \end{cases}$$

Solving the recurrence yields

$$N_{i+j} \leq 6\Delta k + (5/6)^{j-1}\Delta N_i.$$

We are now in a position to calculate the load and the congestion. The preceding argument shows that for $d \in O(\log \Delta)$ and $N_i \in O(\Delta k)$, we have $N_{i+d} \leq N_i$. Thus, in every simple path between a node at level i and a node at level $i+d$, where i is a multiple of Δ, the congestion starts at $O(\Delta k)$ at level i, rises to at most $O(\Delta^2 k)$ at level $i+1$ and proceeds to drop back down to at most $O(\Delta k)$ at level $i+d$. Thus, the congestion of the embedding is at most $O(\Delta^2 \log\log N)$. How large can the load be? At each node of the binary tree we embed a separator of size k. For every i that is a multiple of d, we also embed a set nodes of size $N_i = O(\Delta k)$. Finally, at the leaves we embed forests of size

$$N \log\log N((1 + 1/\log N)/2)^{\log N},$$

which is at most $O(\log\log N)$. Thus the load is at most $O(\Delta \log\log N)$. □

Corollary 4.7 *There is a work-preserving emulation of the class of bounded-degree forests by the class of complete-binary trees with slowdown* $O(\log\log N)$.

4.3 Congestion lower bound for complete ternary trees

In this section we show that any embedding of an N-leaf complete ternary tree T_3 in an M-leaf complete binary tree T_2, $N < M < 3N$, in which the leaves of T_3 are mapped to the leaves of T_2 with load at most $2^{\log^\alpha N}$, fixed $\alpha < 1$, has congestion at least $\Omega(\sqrt{\log \log N})$. This lower bound suggests, but does not prove, that real-time emulation of a complete ternary tree by a complete binary tree is impossible.

Theorem 4.8 *Any embedding of an N-leaf complete ternary tree T_3 in an M-leaf complete binary tree T_2, $N < M < 3N$, in which the leaves of T_3 are mapped to the leaves of T_2 with load $l = 2^{\log^\alpha N}$, fixed $\alpha < 1$, has congestion at least $\Omega(\sqrt{\log \log N})$.*

Proof: The proof has the following outline. Let L denote the number of leaves of T_3 in a subset S of the nodes of T_3, and let w be a base-3 string representing L. First we show that for any S, the number of 1's in w is at most one plus the number of edges between S and $\overline{S}$. As a consequence, if S is the set of nodes mapped to a subtree rooted at a node v in T_2, then the congestion on the edge from the v to its parent is at least as large as the number of 1's in w. Next, we construct a path $v_0, v_1, \ldots, v_{\log M}$ in T_2 from the root v_0 to a leaf $v_{\log M}$ such that there is a long sequence of nodes on the path, $v_j, v_{j+1}, \ldots, v_{j+s-1}$ such that for each v_i, where $j \leq i \leq j + s - 1$, the number of leaves of T_3 mapped to the left and right subtrees of v_i are nearly equal. Let S_i be the set of nodes of T_3 mapped to the subtree rooted at v_i, let L_i be the number of leaves of T_3 in S_i, and let w_i be the base-3 string representing L_i. To complete the proof we show that for some i, where $j \leq i \leq j + s - 1$, there are many 1's in w_i.

First we show that for any subset S of the nodes of T_3, the number of 1's in w is at most $|E_S| + 1$, where E_S is the set of edges in T_3 connecting a node in S to a node in $\overline{S}$. The key idea is that L can be expressed as a series of $|E_S| + 1$ terms, both positive and negative, where each term is a perfect power of 3. If the root of T_3 belongs to S, then the series begins with the term N; otherwise it begins with 0. Thereafter, each edge in E_S contributes a term to the series. An edge between a node u on level l and its parent on level $l - 1$ contributes $N/3^l$ if u is in S, and $-N/3^l$ otherwise. Because adding or subtracting a power of 3 can produce at most one 1 digit in a base-3 number, the number of 1's w is at most $|E(S)| + 1$.

Starting at the root, v_0, we construct the path in T_2 according to the following rule. Suppose that v_i is a node on the path. Then the next node on the path, v_{i+1} is the root of the left or right subtree of v_i containing more leaves of T_3. Let L_i be the number of leaves of T_3 mapped to the subtree rooted at v_i. Then v_{i+1} contains at least $L_i/2$ leaves of T_3. We call the split at v_i *fair* if both of its subtrees contain at most $L_i(1/2+\varepsilon)$ leaves of T_3, where ε will be specified later.

Next we put a lower bound on the length of the longest sequence of consecutive fair splits. Let b be the number of unfair splits on the path. The number of leaves of T_3 mapped to the leaf at the end of the path is at least

$$N\left(\frac{1}{2}\right)^{\log M-b}\left(\frac{1}{2}+\varepsilon\right)^{b}.$$

Since the load is at most l, and $1+x \le e^{x/2}$ for $0 \le x \le 1$, we have $l \ge \frac{1}{3}e^{\varepsilon b}$. Let s be the length of the longest sequence of consecutive fair splits. Then $s \ge \log M/b \ge \varepsilon \log M/\ln 3l$.

We now show that in the longest sequence of consecutive fair splits $v_j, v_{j+1}, \ldots, v_{j+s-1}$, there must be a node v_i, where $1 \le i \le j+s-1$ such that there are many 1's in w_i. For the moment, let us assume that at each node v_i on the sequence, the number of leaves of T_3 mapped to each subtree of v_i is exactly $L_i/2$. Then we can prove that at some node v_i on the sequence, the number of 1's in the t most significant digits of w_i is at least $\sqrt{t}$, where $t = (\log_3 2)s$. Suppose that the the number of 1's in w_j is smaller than $\sqrt{t}$ (otherwise we're done). The 1's in w_j partition it into substrings consisting of 0's and 2's only. In each substring, division by 2 either converts all of the 0's to 1's (leaving the 2's unchanged), or converts all of the 2's to 1's (leaving the 0's unchanged). Thus, after division by 2, 0's and 2's are adjacent in at most $\sqrt{t}$ places in w_{j+1}. Thus, there must be a substring of either $\sqrt{t}$ 0's or $\sqrt{t}$ 2's in w_{j+1}. In either case, after at most s divisions by 2 the substring is converted to all 1's.

Unfortunately, a fair split at a node v_i does not divide L_i exactly by 2; it also adds as much as εL_i. For $\varepsilon \le 1/3^t$, adding εL_i does not change the t most significant bits unless a carry propagates in. We need to show that our substring of $\sqrt{t}$ 0's or 2's is not adversely affected by carries. Since a carry into a substring of 2's turns them all into 0's, we need only consider the effect of a carry into a substring of 0's. A carry into a substring of 0's converts the least significant 0 in the substring into a 1, which is bad, because it reduces

the length of the string. However, $3^{\sqrt{t}/2}$ carries are required to modify the $\sqrt{t}/2$ least significant 0's in the substring. Since at most one carry occurs at each of the s splits, and $s \ll 3^{\sqrt{t}/2}$, the length of the longest string of 0's never drops below $\sqrt{t}/2$.

To finish, we choose values for ϵ, l, and t. To make the lower bound strong, we want to make t large without making l too small. For any fixed $\alpha < 1$, we can choose $l = 2^{\log^{\alpha} N}$, $t = \Theta(\log\log N)$, and $\epsilon = 1/3^t$. The congestion is at least $\sqrt{t}/2 = \Omega(\sqrt{\log\log N})$. □

5 Emulations by butterfly networks

5.1 Work-preserving emulations of larger butterflies

Theorem 5.1 [6] *For any $M \geq N$, an N-node butterfly can perform a work-preserving emulation of an M-node butterfly.*

Proof: An M-node butterfly can be embedded in an N-node butterfly with load $O(M/N)$, dilation 1, and congestion $O(M/N)$. □

5.2 Work-preserving emulations of binary trees

When the Bhatt, Chung, Hong, Leighton, Rosenberg result [3] that a butterfly can emulate a complete binary tree in real time is combined with the material in Section 4.2, we find that there is an $O(\log\log N)$-time work-preserving simulation of the class of bounded-degree trees on the butterfly. Whether or not this emulation can be performed in real time remains an open question.

5.3 Real time emulation of meshes

In this section we prove the following theorem:

Theorem 5.2 *For constant q, any T-step computation on a q-dimensional mesh with N nodes can be emulated in $O(T)$ steps on an $O(N)$-node butterfly.*

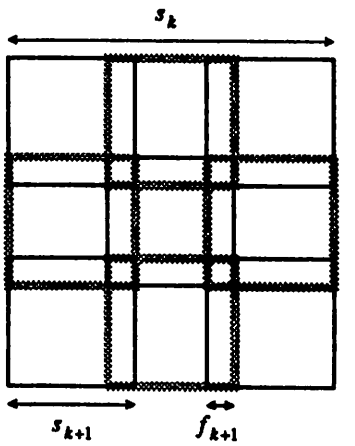

Figure 2: The division of the $s_k \times s_k$ mesh into submeshes. Each $s_{k+1} \times s_{k+1}$ submesh overlaps its neighbors in either f_{k+1} rows or f_{k+1} columns.

Proof: We prove the theorem for $q = 2$ and $T \geq \log N$; for other values of q and smaller T the proof is similar.

We will perform the emulation recursively, by dividing the mesh into overlapping submeshes and recursively emulating them on distinct subbutterflies of the emulating butterfly. In order to complete their computations, the submeshes will at times need pebbles which are computed in other submeshes; for this purpose we must also define connections between the emulating subbutterflies.

At the kth level of the recursion, we will be emulating f_k steps of computation of an $s_k \times s_k$ mesh on a subbutterfly with $N_k = n_k 2^{n_k}$ nodes, where $s_k^2 = N_k/m_k$ and we assume that n_k is a power of two. This will be accomplished by dividing the $s_k \times s_k$ node mesh into submeshes of $s_{k+1} \times s_{k+1}$ nodes which overlap by f_{k+1} rows or columns (see Figure 2). At this level of the recursion, we will assume that each of these submeshes can be emulated by a subbutterfly with $N_{k+1} = n_{k+1} 2^{n_{k+1}}$ nodes, $s_{k+1}^2 = N_{k+1}/m_{k+1}$, and show that the desired emulation for level k can be performed. We will show later that the subscripted quantities can be chosen such that the top-level emulation is performed in real time on a butterfly with $O(N)$ nodes.

The emulation of f_k steps on the $s_k \times s_k$ mesh will be perfomed in f_k/f_{k+1}

phases. In each phase we first run f_{k+1} steps of the emulation of each submesh in a distinct subbutterfly. If nothing else were done, any node of a submesh at distance less than f_{k+1} from the border of its submesh would be unable to successfully perform this emulation, because it would need pebbles which are only computed in some other submesh. However, for every node v on the border of a submesh there is a node v' in another submesh which emulates the same node of the mesh and is at distance f_{k+1} from the border of its submesh; this node will be able to successfully emulate f_{k+1} steps of its computation and therefore calculate the pebbles $(e,1),\ldots,(e,f_{k+1})$ for the edges incident to v. By defining a path in the butterfly between the nodes emulating v and v', we allow v' to send these pebbles to v. As v receives these pebbles (and the other nodes along the border of the submesh receive theirs), the emulation can resume, and the nodes within distance f_{k+1} of the border of the submesh can complete their calculations. Since we must enable all such pairs of nodes to send pebbles simultaneously without slowing down the emulation of the submeshes, we must choose the paths so that a most a constant number of paths share any edge; this must be the case over all levels of the recursion simultaneously. In order to provide these paths connecting nodes in the butterfly, we will reserve some subbutterflies for the sole purpose of routing paths between the subbutterflies which are emulating submeshes.

At level k of the recursion, we will assume that we have an embedding of the nodes of a $s_{k+1} \times s_{k+1}$ mesh in a N_{k+1}-node butterfly, and that we are given the paths connecting corresponding nodes within that subbutterfly for deeper levels of the recursion. In addition, we will assume that for each node v on the border or at distance f_{k+1} from the border of the $s_{k+1} \times s_{k+1}$ mesh (call this set of nodes F_{k+1}) there is a butterfly node $\langle i, Y_v \rangle$ which emulates v and a node $\langle 0, X_v \rangle$ in level 0 of the butterfly such that there is a path between these two nodes along which pebbles can be sent without slowing down the emulation of the $s_{k+1} \times s_{k+1}$ mesh. Furthermore, letting $X_v = c_{n_{k+1}-1} \cdots c_0$, we assume that $c_{\epsilon_{k+1}-1} \cdots c_0 = 10 \cdots 0$ (ϵ_{k+1} is another quantity that will be specified later), and for all v in F_{k+1}, the X_v's will share a common value of $c_{2\epsilon_{k+1}-1} \cdots c_{\epsilon_{k+1}}$ that can be chosen arbitrarily.

We will divide the $s_k \times s_k$ mesh into submeshes as illustrated in Figure 2, with the additional requirement that all nodes which are in F_k of this mesh will lie in F_{k+1} of their respective submeshes. In order to do this, we may have to shrink the sizes of submeshes in up to two rows and two columns of

submeshes. When we recursively emulate f_{k+1} steps of one of these reduced submeshes, we will consider it to be part of a full-size mesh that has dummy nodes.

Now we describe the partition of the N_k-node butterfly into distinct subbutterflies with N_{k+1} nodes each. For each node in the butterfly we will denote the column of the node by $c_{n_k-1} \cdots c_0$.

Each subbutterfly will consist of the set of butterfly nodes with following property: Let α be a multiple of n_{k+1} (possibly zero); then all nodes of a subbutterfly share common values of $c_{\alpha-1} \cdots c_0$ and $c_{n_k} \cdots c_{\alpha+n_{k+1}}$, and are in levels between α and $\alpha + n_{k+1} - 1$, inclusive. Intuitively, this divides the butterfly into subbutterflies by removing all edges between each level which is a multiple of n_{k+1} and the next highest level, and considering the remaining connected components.

As was mentioned before, not all subbutterflies will be used to emulate submeshes; some will be used only to create connections between the subbutterflies which are performing emulations. A subbutterfly will be used exclusively for creating such connections if there exists γ a multiple of n_{k+1}, $\gamma > \alpha$ (where α is the value used to define the subbutterfly) and $c_{\gamma+\epsilon_{k+1}-1} \cdots c_\gamma = 10 \cdots 0$ for all nodes in the subbutterfly, or if $\alpha > 0$ and $c_{\epsilon_{k+1}-1} \cdots c_0$ equals either $10 \cdots 0$ or $0 \cdots 0$.

First we must insure that the number of subbutterflies to be used for emulating submeshes is at least as big as the number of submeshes to be emulated. The number of $s_{k+1} \times s_{k+1}$ submeshes is at most

$$\left(\frac{s_k}{s_{k+1} - f_{k+1}} + 2 \right)^2$$

(the additive two is due to the shrinkage of some submeshes to insure that F_k is included in F_{k+1}).

The total number of subbutterflies in the partition of the butterfly is N_k/N_{k+1}. The number of subbutterflies that will be used exclusively for routing paths and not for the emulation of submeshes is at most

$$\left(\frac{n_k}{n_{k+1}} \right)^2 2^{n_k - n_{k+1} - \epsilon_{k+1}}.$$

Thus in order for there to be enough subbutterflies to perform the emulation, we must have

$$\left(\frac{s_k}{s_{k+1}-f_{k+1}}+2\right)^2 \leq \frac{N_k}{N_{k+1}} - \left(\frac{n_k}{n_{k+1}}\right)^2 2^{n_k-n_{k+1}-\epsilon_{k+1}}. \qquad (1)$$

Now, in order to satisfy the recursive hypothesis at level k, we must do two things:

1. Choose paths in the N_k-node butterfly between the nodes emulating each v in F_{k+1}, in such a way that the emulation will not be slowed down by pebbles moving along the paths.

2. Choose for each node u in F_k a path from some node emulating u to some node in level 0 of the butterfly with the ϵ_k lowest order bits of its column equal to $10\cdots0$, and such that all the chosen nodes in level 0 share a common value for the next ϵ_k lowest order bits.

First we describe how to choose paths in the butterfly between the two nodes emulating each node v in F_{k+1}. Let u be a butterfly node emulating v, and let α be the parameter describing the subbutterfly in which u is located. Since v is in the set F_{k+1} for its submesh, we know that there exists a node u_1 in level α of the butterfly (that is, in level 0 of the subbutterfly) such that if the column of u_1 in the butterfly is $c_{n_{k+1}-1}\cdots c_0$, then we have $c_{\alpha+\epsilon_{k+1}-1}\cdots c_\alpha = 10\cdots0$, and there is a path from u to u_1 allowing pebbles to be sent without any slowdown to the emulation. In addition, for all v in the set F_{k+1} for that submesh, the columns of their corresponding u_1's share a common value of $c_{\alpha+2\epsilon_{k+1}-1}\cdots c_{\alpha+\epsilon_{k+1}}$ that can be chosen arbitrarily. We choose the bits $c_{\alpha+2\epsilon_{k+1}-1}\cdots c_{\alpha+\epsilon_{k+1}}$ to be equal to $c_{\epsilon_{k+1}-1}\cdots c_0$, which all nodes in the same subbutterfly as u have in common (except for the case of $\alpha=0$, but in that case the paths to level 0 defined in the recursive hypothesis are all we need). Let u_2 be the node in level zero of the butterfly whose column is obtained by changing the ϵ_{k+1} least significant bits of the column of u_1 to $10\cdots0$. Now our choice of subbutterflies to be used for emulating submeshes and our choice of $c_{\alpha+2\epsilon_{k+1}-1}\cdots c_{\alpha+\epsilon_{k+1}}$ for u_1's column are sufficient to show that the paths from u_1 to u_2 for different choices of v are all disjoint.

To illustrate this, consider two nodes u and u' (which emulate nodes in F_{k+1}) in subbutterflies with parameters α and α'. We must show that the paths described in the previous paragraph are disjoint. Suppose $\alpha \neq$

α' (without loss of generality, $\alpha < \alpha'$) and that the paths from u and u' to their respective nodes in level 0 intersect. Since we are using greedy routing to level 0, the bits $c_{\alpha'+\epsilon_{k+1}-1} \cdots c_{\alpha'}$ of u must be equal to $10 \cdots 0$, the value of those bits in u'. Thus setting $\gamma = \alpha'$, we see that the subbutterfly containing u' would not have been used for emulating a submesh, yielding a contradiction. Therefore it must be the case that $\alpha = \alpha'$. Next we observe that since we are only changing the lowest order ϵ_{k+1} bits, paths from the same subbutterfly will never intersect. Thus we need only consider paths from distinct subbutterflies with the same parameter α. Since we are using greedy routing to level 0, intersections can only occur if the nodes u_1 and u_1' differ only in the ϵ_{k+1} lowest order bits. In this case, however, the paths will not collide since we have set the bits $c_{\alpha+\epsilon_{k+1}-1} \cdots c_{\alpha}$ equal to those differing low order bits. Thus all the paths from u to u_1 to u_2 and from u' to u_1' to u_2' are disjoint; similar arguments can be used to show that these paths do not conflict with paths from deeper levels of the recursion.

Given butterfly nodes u and u' emulating node v in F_{k+1}, we now have paths from u to u_1 to u_2 and from u' to u_1' to u_2' along which pebbles can be sent with no additional slowdown to the emulation. All that remains us to connect each u_2 to the corresponding u_2'. This is done by routing a permutation on the subbutterfly consisting of those nodes for which the ϵ_{k+1} lowest order bits of their columns are $10 \cdots 0$, using one pass up and one pass down through the butterfly [1]. None of these paths will conflict with any paths chosen at a different level of the recursion, since the ϵ_{k+1}'s will be distinct.

It remains to show that for each node v in F_k of the $s_k \times s_k$ mesh there is a node u in level zero of the butterfly and a node w emulating v such that u and w can be connected by a path of length $O(n_k)$ along which pebbles can be sent between u and w without slowing down the simulation. In addition, the ϵ_k lowest order bits of the column of u must be $10 \cdots 0$, and $c_{2\epsilon_k-1} \cdots c_{\epsilon_k}$ is common for all u and can be chosen arbitrarily.

First we assign the nodes in F_k of the mesh to nodes in the butterfly with the required characteristics, so that at most one node of the mesh is assigned to any node in the butterfly. In order for there to be enough nodes in the butterfly with the required properties, we must have

$$\log 8s_k < n_k - 2\epsilon_k. \tag{2}$$

Since each node in F_k is in F_{k+1} in its submesh, we know that for each v in F_k there will be some node u' in level zero of the butterfly, the ϵ_{k+1} least significant bits of whose column are $10\cdots0$, which is connected by a path of length $O(n_k)$ to some w which emulates v. Choosing a set of appropriate u's, we can complete the desired paths from each w to each u' by combining the previously defined paths from w to u with the result of routing one permutation on the columns to connect u and u' for each v. Again, the distinctness of the ϵ_{k+1}'s guarantees that these paths will not conflict over the different levels of the recursion.

We have shown how to preserve the recursive hypothesis at level k, given the hypothesis at level $k+1$. Therefore at the top level, we will be emulating an $s_0 \times s_0$ mesh on an N_0-node butterfly. What remains is to choose the values s_k, f_k, N_k, ϵ_k and m_k so that (1) and (2) are satisfied, and in such a way that the resulting emulation will be performed in real time. Let $\omega(N)$ be the smallest value of k such that $N^{20^{-k}} < 2$, and let $\epsilon_k = \frac{1}{30}\log n_k$. We let $s_0 = \sqrt{N}$, and for $0 < k \leq \omega(N)$ choose s_k and n_k so that $N^{10^{-k}} \leq s_k \leq (N^{10^{-k}})^2$, n_k is a power of two ($N_k = n_k 2^{n_k}$) and

$$N_k = s_k^2 \prod_{j=k+1}^{\omega(N)} m_j',$$

where

$$m_k' = \left(\frac{s_k}{s_k - f_k} + 2\frac{s_k}{s_{k-1}}\right)^2 \left(\frac{1}{1 - \frac{n_{k-1}}{n_k}2^{-\epsilon_k}}\right).$$

Then we choose $f_0 = T$, $f_1 = \min\{T, \sqrt{s_1}\}$, and for $k \geq 2$, $f_k = \sqrt{s_k}$. Since the product

$$m_k = \prod_{j=k+1}^{\omega(N)} m_j'$$

is bounded for all possible s_k in the specified range (to see that the terms shrink fast enough to insure this, observe that for all k, $s_{k+1} \leq (s_k)^{1/5}$), we know that we can choose such an s_k for each k. It also follows that the size of the emulating butterfly exceeds the size of the mesh by only a constant factor (since m_0 is bounded). Now by simple substitution we can verify that

(1) and (2) hold, so that the emulation we have described will work. We need only verify that the emulation takes place in real time.

Let T_k be the time to emulate f_k steps of a $s_k \times s_k$ mesh on a N_k-node butterfly. This emulation is divided into f_k/f_{k+1} phases; in each phase, we attempt to recursively emulate f_{k+1} steps of the $s_{k+1} \times s_{k+1}$ submeshes on the N_{k+1}-node subbutterflies described previously. Each phase requires time $T_{k+1} + O(n_k)$ for the recursive emulation and the delivery of pebbles along the paths of length $O(n_k) = O(\log s_k)$. Therefore

$$T_k = \frac{f_k}{f_{k+1}}(T_{k+1} + O(\log s_k)),$$

so that the total time required for the emulation is

$$\begin{aligned}
&\sum_{k=0}^{\omega(N)} \left(\prod_{j=0}^{k} \frac{f_k}{f_{k+1}} \right) \log s_k \\
&= \sum_{k=0}^{\omega(N)} \frac{f_0 \log s_k}{f_{k+1}} \\
&= T \sum_{k=0}^{\omega(N)} \frac{\log s_k}{f_{k+1}} \\
&= O(T).
\end{aligned}$$

□

5.4 Embedding the shuffle-exchange graph in the butterfly

In this section, we show how to embed an N-node shuffle-exchange graph in an $O(N)$-node butterfly with constant load, constant congestion, and $O(\log N)$ dilation.

A constant congestion embedding requires that very few edges of the shuffle-exchange graph be mapped to long (more than constant length) paths in the butterfly. In addition, these paths must not overlap each other very often. To ensure this, we use the fact that the inputs and outputs of a Benes network can be connected in any permutation by a set of disjoint paths [2].

That is, if the set of long paths can be decomposed into a constant number of (partial) permutations of the inputs of the butterfly, the long paths can be embedded with constant congestion. It is easy to see that we can embed the long paths in this manner when there are at most a constant number of endpoints of long paths in any single butterfly column. (We first route a path from each endpoint to the input of its column, which leaves us with a constant number of permutations to route on the Benes network.)

We map the nodes of a shuffle-exchange graph to the nodes of a butterfly so that

1. at most a constant number of shuffle-exchange nodes are mapped to any one butterfly node, and

2. each butterfly column contains at most a constant number of shuffle-exchange nodes which have any neighbor mapped to a distant node in the butterfly.

Short paths only contribute constant congestion since they have constant length. Long paths only contribute constant congestion since we can route any permutation with congestion 2, and we only need to route a constant number of (partial) permutations. Also, the length of the short paths is constant and the long paths is $O(\log n)$.

In particular, we map the nodes of a $N = 2^n$-node shuffle-exchange graph to the nodes of a $(n + 2 - \log n)2^{n+2-\log n} \approx 4N$-node butterfly. Each node in this N-node shuffle-exchange graph has n bits in its label. A node in the butterfly can be specified by a column represented by $n+2-\log n$ bits, and a level in that column. The level in the column corresponds to a bit that can be flipped to enter another column. Thus, we first associate a shuffle-exchange node with a particular column of the butterfly by removing $\log n - 1$ adjacent bits of its label none of which are the least significant bit, then we pick the level in the column which corresponds to where the least significant bit of the shuffle-exchange node appears in the column string.

We map a shuffle-exchange node w to a node in the butterfly as follows,

1. Consider the longest string of zeros in w ignoring the least significant bit; break ties by choosing the first one from the left.

2. Pick out $\log n - 1$ bits as follows;

(a) If possible choose the $\log n - 1$ bits after the zeros and before the lsb,

(b) otherwise if possible choose the $\log n - 1$ bits preceding the longest string of zeros,

(c) otherwise choose the last $\log n - 1$ bits of the string of zeros (note that in this case more than $n - 2\log n$ bits are zeros).

3. Treat these bits as a number (it will be in the range $0...\frac{n}{2}$), call this number s, and the sequence of bits a_s.

4. Remove the bits of s from w, extend the chosen string of zeros on the right (left) by a 01 (10) if the bits were removed from the right (left) of the block of zeros, and cyclic shift the resulting string so that s bits appear after the longest string of zeros, this specifies the column.

Symbolically, we map $w = z0^k a_s yb$ to column $u0^{k+1}1v$, or we map $w = za_s0^k yb$ to column $u10^{k+1}v$, with $ybz = vu$ and $|v| = s$. (Note that we map to a column with a unique longest string of zeros not straddling the bit which is at the level of the butterfly node.) It is easy to see that the least significant bit of w, b, is somewhere in the column string. We choose the level to correspond to the position of b in the column string C.

We must argue that the mapping achieves condition 1 and 2 above.

First, we introduce some more notation. We define a *necklace* to be a set of shuffle-exchange nodes which are connected only by shuffle edges. Alternatively, a necklace is a set of nodes having labels which are cyclic shifts of each other. A necklace's *label* is the lexicographically minimum label of its nodes. We can specify a shuffle-exchange node by the label of its necklace and the position of the least significant bit of the node's label in the necklace's label.

We define the *domain* of a butterfly node to be the set of shuffle-exchange nodes that are mapped to it by our mapping.

Now we show that the mapping is at most two to one. That is, given a butterfly node $\langle l, C\rangle$ we can describe at most two shuffle-exchange nodes that could possibly be mapped to $\langle l, C\rangle$ as follows. Recall that a butterfly node $\langle l, C\rangle$ has all the bits of w in the string C except for a_s. And these, we recover by finding the length of the string after the longest group of zeros in C not straddling the lth bit. We know that we have to reinsert them either directly

before or directly after that group of zeros. This gives us all the bits of the domain nodes except for a cyclic shift uncertainty. Thus, the domain of $\langle l, C \rangle$ can only be nodes from two necklaces. Furthermore, the least significant bit of the nodes' labels is uniquely specified by the place where the lth bit of C occurs in the necklaces' labels. Thus only two shuffle-exchange nodes can be mapped to any node in the butterfly.

Finally, we argue that we map at most a constant number of shuffle exchange nodes with distant neighbors to any butterfly column.

Notice that we always ignore the value of the least significant bit in the mapping of shuffle-exchange nodes to butterfly nodes. Thus the mapping maps two shuffle-exchange nodes to two nodes that only differ in the bit that can currently be changed by a butterfly edge. Thus, any exchange edge needs only flip the bit at the node's level, which only requires a path of length 2. Thus all exchange edges are embedded in short paths.

Now consider the shuffle edges. We show that at most a constant number of shuffle edges leave any column of the butterfly. (It is easy to see that all the shuffle edges in a column are mapped to single edges in the butterfly graph.) Again, consider the inverse mapping of a butterfly node, $\langle l, C \rangle$, to two shuffle-exchange nodes. The necklaces of the domain nodes of column C's nodes are the same for most of the column. They change only at certain transition levels in the column; levels, l, in the column where the position of the longest string of zeros not straddling l changes, or levels in the column where we become unsure or sure of which side of the zeros to replace the removed bits, a_s.

The position of the longest string of zeros not straddling l only changes at two points; inside the column's unique longest string of zeros. When the column level is within $\log n$ bit positions to the right of the longest string of zeros, we know that pieces of two shuffle-exchange necklaces could have been mapped to the column. Outside this range we know that only one necklace is mapped to the column: Inside the group of zeros the bits were definitely taken out before the group of zeros, and further to the right they were definitely taken out after the group of zeros. Thus entering this stretch and leaving this stretch gives us two more bad levels. Thus we have four transition levels in all, and for each of these at most four necklaces could enter or leave the column at any of these levels. Thus at most 16 long shuffle edges can have endpoints in this column. (Careful counting can reduce this number to 6.)

Thus at most 16 long edges are adjacent to any column of the butterfly. This satisfies condition 2, above.

Thus, the shuffle-exchange graph can be embedded in the butterfly with constant congestion.

5.5 Layouts for the shuffle-exchange graph with optimal area and volume

The N-node butterfly can be laid out in $O(N^2/\log^2 N)$ area (trivially) and in $O(N^{3/2}/\log^{3/2} N)$ volume [22]. Since the N-node shuffle-exchange graph can be embedded in the N-node butterfly with constant congestion, we can simply blowup these layouts by a constant factor to obtain layouts for the shuffle-exchange graph with equivalent area and volume.

5.6 A real time emulation of the shuffle-exchange graph

In this section, we prove the following theorem:

Theorem 5.3 *T steps of an N-node shuffle-exchange graph computation can be emulated in $O(T)$ steps on an N-node butterfly, for any T.*

Proof: We will show how to map the initial states of the shuffle-exchange graph nodes to the nodes of the butterfly so that after $O(n)$ steps, each node which originally had the state of shuffle-exchange node X at time 0 will contain the state of shuffle-exchange node X at time $n/8$. By iterating this procedure as many times as is necessary, we can simulate any T-step computation in $O(T)$ steps.

Let S be the set of strings of length $n/8$ such that one of their longest runs of zeroes is either at the extreme left or the extreme right of the string. Since the mapping $Y \mapsto 0Y1$ maps strings with one of their longest runs of zeroes at the extreme left to strings of length $n/8+2$ which have a unique longest run of zeroes and are lexicographically least among their cyclic shifts, the number of strings in S is $O(2^{n/8}/n) = O(N^{1/8}/\log N)$.

Next we observe that $O(N^{1/8}/\log N)$ butterflies of $\frac{1}{2}N^{7/8}$ by $\frac{7}{8}\log N$ nodes, without wraparound, can be embedded in an $O(N)$-node butterfly with constant load, dilation and congestion in such a way that the nodes in level i of each small butterfly are all mapped to level i of the larger butterfly.

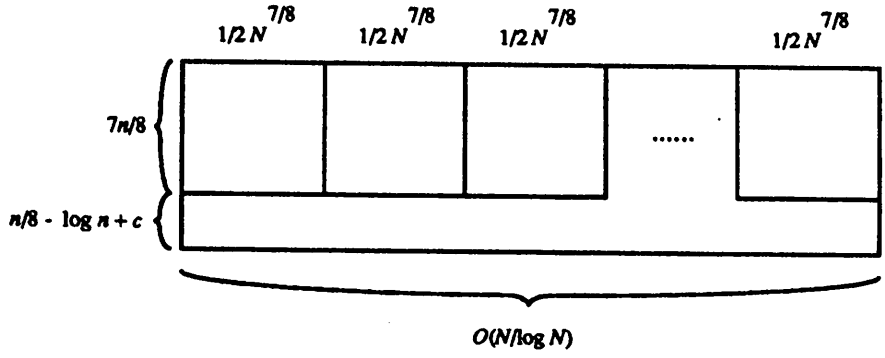

Figure 3: Embedding the set of butterflies indexed by $\mathcal{S}$ in one large butterfly ($n = \log N$).

This is done by lining up these butterflies as the top $\frac{7}{8}\log N$ levels of a butterfly with $O(N/\log N)$ columns and hooking them up with $\frac{1}{8}\log N - \log\log N$ (plus some constant) additional levels of nodes, as illustrated in Figure 3.

We will map the nodes of the shuffle-exchange graph to the nodes of such a collection of $O(N^{1/8}/\log N)$ butterflies, indexed by the strings in $\mathcal{S}$. Node $X = x_n \cdots x_1$ of the shuffle-exchange graph is mapped to several locations in this collection of butterflies, depending on which of its substrings are in $\mathcal{S}$. Suppose that for $j \in \{n/4, \ldots, 7n/8\}$, the substring $x_j \cdots x_{j+1-n/8}$ is in $\mathcal{S}$; then X is mapped to node $\langle n-j, x_{j-n/8} \cdots x_1 x_n \cdots x_{j+1}\rangle$ in butterfly $x_j \cdots x_{j+1-n/8}$. Thus X is mapped to as many locations as there are j's which satisfy this condition. Let $G(X)$ be the set of locations to which X is mapped; we refer to this as the set of *images of* X.

To show that at most one shuffle-exchange node X is mapped to each node in the collection of butterflies, let $Y \in \mathcal{S}$, $l \in \{0, \ldots, 7n/8 - 1\}$ and $C \in \{0,1\}^{7n/8}$ be given. It is clear from the mapping defined in the previous paragraph that for any X mapped to location $\langle l, C\rangle$ in butterfly Y, $x_{n-l} \cdots x_{7n/8+1-l} = Y$. Furthermore, we must have $x_{7n/8-l} \cdots x_1 x_n \cdots x_{n-l+1} = C$, which determines the remaining bits of X. Thus for each node in the collection of butterflies, there is only one node X which could possibly be mapped to that position.

The following lemma shows that any image of a node which is not on the boundary of the region containing the images (levels $n/8, \ldots, 3n/4$ of each butterfly) has images of all that node's neighbors nearby.

Lemma 5.4 *Let $g(X)$ be an image of shuffle-exchange node X in butterfly Y which is in a level other than $n/8$ or $3n/4$ of Y. Then all the neighbors of X in the shuffle-exchange graph have images in butterfly Y within one level and distance two of $g(X)$.*

Proof: Let X be such a node. Then for some $j \in \{n/4+1, \ldots, 7n/8-1\}$, $Y = x_j \cdots x_{j+1-n/8}$; in butterfly Y, this node is mapped to level $n-j$ and column $C = x_{j-n/8} \cdots x_1 x_n \cdots x_{j+1}$. It is easy to verify that $\sigma_1(X)$ and $\sigma_{n-1}(X)$ are mapped to $\langle n-j+1, C\rangle$ and $\langle n-j-1, C\rangle$ in butterfly Y and that $x_n \cdots x_2 \overline{x_1}$ is mapped to $\langle l, C'\rangle$ in butterfly Y, where C' differs from C in only the $(n-j+1)$st bit. This proves the lemma. □

For each X, assign the initial state of node X to all the nodes in $G(X)$. From Lemma 5.4 it follows that all images in levels $3n/8 \ldots n/2$ of the butterfly can simulate $n/8$ steps of their nodes' computations correctly in $O(n)$ steps, since they are more than $n/8$ levels away from either boundary. Let the images which are in these $n/8+1$ levels be called *good* images of the nodes they are simulating.

Claim: Every node X has a good image.

Proof Consider the middle $n/4$ bits of X, $x_{5n/8} \cdots x_{3n/8+1}$, and look at one of its longest runs of zeroes. We can always choose a substring of length $n/8$ with this run of zeroes at its extreme left or right; letting this be Y yields an image of X in one of the levels $3n/8 \ldots n/2$ of butterfly Y, proving the claim.

We can simulate $n/8$ steps of computation on each of the small butterflies in $O(n)$ steps on an $O(N)$-node butterfly. After this, each good image of a node X will have calculated the state of X at time $n/8$; it follows that for $T \leq n/8$, simulating T steps of computation on each of the small butterflies is all that is necessary to effect the desired simulation. In order for us to be able to continue the simulation beyond $n/8$ steps, we must show how to update the other images, so that every location which began with the initial state of X will contain the state of X at time $n/8$.

Each good image of a node can calculate the pebbles $(e,1),\ldots,(e,n/8)$ for all edges incident to that node. In order to update the images in levels $n/8\ldots n/4$ and $5n/8\ldots 3n/4$ of the butterfly, the good image of each node with an image on the boundary can send these pebbles in turn to the boundary image. The images in levels $n/8\ldots n/4$ and $5n/8\ldots 3n/4$ can save their input pebbles and perform their computations as the needed pebbles arrive at the boundary. As long as the paths connecting the good images and the boundary images are of length at most $O(n)$ and have constant total congestion, this update will take $O(n)$ steps.

If we could find a good image for each node with an image on a boundary such that only a constant number of these good images were in any column, then this could be accomplished by routing a constant number of permutations on the columns of this butterfly. Unfortunately, we cannot guarantee that such good images can be found. Instead we will adjust the boundary so that such good images exist for all shuffle-exchange nodes mapped to a boundary level.

Consider a shuffle-exchange node X whose image is in a boundary level of its butterfly (without loss of generality, say level $3n/4$), and let $\mathcal{R}$ be the set of nodes which differ from X only in the $n/8$ least significant bits. Then the images of the nodes of $\mathcal{R}$ on the boundary form the inputs of a sub-butterfly of depth $n/8$. Let c be such that X has a good image in level $3n/8+c$ of some butterfly; it follows that each node in $\mathcal{R}$ has a good image in that level of the same butterfly. On the columns containing the boundary images of the nodes of $\mathcal{R}$, push the boundary up to level $3n/4-c$; this has the effect of 'unembedding' some images of nodes, but leaves the good images undisturbed (this operation is illustrated in Figure 4). Each of the nodes with images on this new boundary will have a good image in level $3n/8$. Repeat this process for each of the $O(N^{7/8}/\log N)$ possible sets $\mathcal{R}$, and perform the corresponding operation on boundary level $n/8$.

All the nodes with images on this modified boundary will have good images in either level $3n/8$ or $n/2$; furthermore, all these good images will still be at least $n/8$ levels from either boundary, allowing them to calculate their nodes' states and the pebbles $(e,1),\ldots,(e,n/8)$ for all edges e incident to X correctly in $O(n)$ steps. Since there are a constant number of boundary images and good images corresponding to those boundary images in any column of the $O(N)$-node butterfly, we can choose a set of one-to-many routing paths connecting each good image to its corresponding boundary

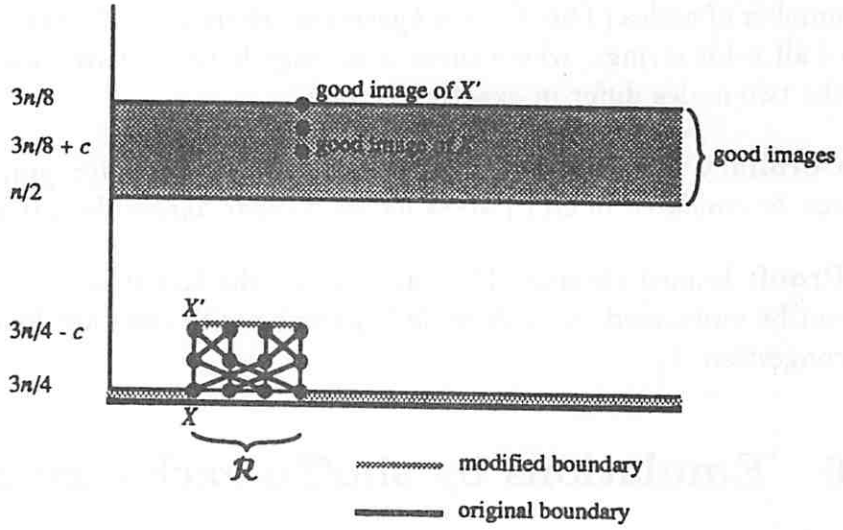

Figure 4: Adjusting the boundary for X and those nodes which differ from X in the $n/8$ lowest order bits. Note that all good images remain at distance at least $n/8$ from the boundary.

images. These paths will have length $O(n)$ and constant total congestion. Thus we can in $O(n)$ steps update each location which began with the initial state of shuffle-exchange node X to contain the state of X at time $n/8$.

Since the above procedure takes $O(n)$ steps, iterating it will allow us to simulate any T-step shuffle-exchange calculation in $O(T)$ steps an $O(N)$-node butterfly. This suffices to prove the theorem. □

The following corollary to this theorem shows that an N-node shuffle-exchange graph can be emulated in real time by a hypercube with the same number of nodes (The N-node *hypercube*, where $N = 2^n$, has nodes consisting of all n-bit strings, where there is an edge between two nodes if and only if the two nodes differ in exactly one bit).

Corollary 5.5 *T steps of an N-node shuffle-exchange graph computation can be emulated in $O(T)$ steps on an N-node hypercube, for any T.*

Proof: Immediate from Theorem 5.6 and the fact that an N-node butterfly can be embedded in an N-node hypercube with constant load, dilation and congestion [7]. □

6 Emulations by shuffle-exchange graphs

6.1 Work-preserving emulations of larger shuffle-exchange graphs

Theorem 6.1 [6] *For any $M \geq N$, an N-node shuffle-exchange graph can perform a work-preserving emulation of an M-node shuffle-exchange graph.*

Proof: An M-node shuffle-exchange graph can be embedded in an N-node shuffle-exchange graph with load $O(M/N)$, dilation 2, and congestion $O(M/N)$. □

6.2 Work preserving emulations of arbitrary binary trees

It is well known that the shuffle-exchange graph can emulate a complete binary tree in real time. Thus by the results of Section 4, we know that there

is an $O(\log \log N)$-time work-preserving emulation of the class of bounded-degree trees on the shuffle-exchange graph. Whether or not this emulation can be made real-time remains an open question.

6.3 Embedding small butterflies in the shuffle-exchange graph

In this section we show how to embed $M/\log M$ distinct $M \log M$-node butterfly graphs in an $N = M^2$ shuffle-exchange graph with load $l = 2$, congestion $c = O(1)$, and dilation $d = 3$. A similar result was proved by Raghunathan and Saran [18]. We assume that $M = 2^k$, so that each column of the 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 easily proved lemma.

Lemma 6.2 *The set of* $\log M$*-bit strings has at least* $M/2 \log M$ *disjoint subsets each containing* $\log M$ *distinct strings which are cyclic shifts of each other.*

For each of these subsets we pick the lexicographically minimum string to represent the subset. We associate the $M/\log M$ butterflies two to one with the $M/2 \log M$ representative strings. Say butterfly i is associated with string W^i. We map a node $\langle p, C \rangle$ in butterfly i to a shuffle-exchange node by shuffling the bits of W_i with the bits of C's representation $c_{\log M} \cdots c_1$, and cyclically shifting the string so that the image of c_{p+1} is at the rightmost bit position. In other words, for $W^i = w^i_{\log M} \cdots w^i_1$, node $\langle p, C \rangle$ of butterfly i is mapped to shuffle-exchange node $w^i_p c_p \cdots w^i_1 c_1 w^i_{\log M} c_{\log M} \cdots w^i_{p+1} c_{p+1}$.

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 column by picking out the other bits and shifting by the same amount. The position in the column is clearly the number of shifts we used to get to W_i and the column 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 the image of $\langle p+1, C \rangle$, or we complement the rightmost bit and shift twice to reach the image of $\langle p+1, c_n \ldots \overline{c_p} \ldots c_1 \rangle$.

Thus we can embed $2\sqrt{N}/\log N$ butterflies with $\frac{1}{2}\sqrt{N}\log N$ nodes each in an N-node shuffle-exchange graph with load 2, congestion $O(1)$, and dilation 3. This technique can be extended to prove that for any constant $0 < \epsilon < 1$, N^ϵ distinct $N^{1-\epsilon}$-node butterflies can be embedded in an N-node shuffle-exchange with constant dilation, and load and congestion $O(1/\epsilon)$.

6.4 A real time emulation of the butterfly

In this section, we prove the following theorem:

Theorem 6.3 *T steps of an N-node butterfly computation can be emulated in $O(T)$ steps on an N-node shuffle-exchange graph, for any T.*

Proof: We will show how to map the initial states of the butterfly nodes to the nodes of the shuffle-exchange graph so that after $O(r)$ steps, each node which originally had the state of butterfly node v at time 0 will contain the state of butterfly node v at time $r/8$. By iterating this procedure as many times as is necessary, we can simulate any T-step computation in $O(T)$ steps.

Without loss of generality, assume r is even. Consider the network which results from restricting the N-node butterfly to its first $r/2$ levels; we call this a *half-butterfly*. The following lemma will allow us to embed multiple half-butterflies in an $O(N)$-node shuffle-exchange graph for the purpose of our simulation.

Lemma 6.4 *An M-node butterfly ($M = s2^s$, s even) can be embedded in an $O(M)$-node shuffle-exchange graph with constant congestion and load, and constant dilation except for the edges from levels $s/2 - 1$ to $s/2$ and from levels $s - 1$ to 0 (which will have dilation $O(\log M)$).*

Proof: In the previous section we showed that $2\sqrt{N}/\log N$ distinct butterflies with $\frac{1}{2}\sqrt{N}\log N$ nodes each can be embedded in an N-node shuffle-exchange graph with constant load, dilation and congestion.

Call the two butterflies embedded using the string W^i *upper* and *lower butterfly i*. We will form a single butterfly in which the upper butterflies serve as the first $\frac{1}{2}\log N$ levels and the lower butterflies supply the remaining levels (while also duplicating some levels already present in the upper butterflies). Thus for $k = 1, \ldots, \sqrt{N}$ we must connect the set consisting of the kth output

of each upper butterfly to some set of $\sqrt{N}/\log N$ consecutive inputs of one of the lower butterflies; each lower butterfly will receive $\log N$ such sets. After this is done for each k, the jth set of $\log N$ outputs from upper butterfly i will be mapped to the ith set of $\log N$ inputs of lower butterfly j, so that the first $\log N - \log\log N$ levels of the lower butterflies will form the needed connections (the remaining levels will duplicate the first $\log\log N$ levels of the upper butterflies).

Since we can permute the inputs of each butterfly with $O(\log N)$ dilation and constant congestion, it suffices to show that we can choose $\log N$ paths from each upper butterfly to each lower butterfly such that over all the paths, the total number of endpoints in any column of an upper or lower butterfly is constant and the total congestion is constant. Routing the $\sqrt{N}$ outputs (inputs) of each upper (lower) butterfly to the endpoints of these paths as necessary will result in a network of $N/\log N$ by $\log N$ nodes which is a butterfly with $\log\log N$ duplicated levels and no wraparound edges. This network will be embedded with constant load and congestion, and constant dilation in all levels but one, which will have dilation $O(\log N)$. Removing the duplicated levels equally from the upper and the lower butterflies and routing a permutation on the columns for the wraparound edges will yield a butterfly with $M = \Omega(N)$ nodes which is embedded in a shuffle-exchange graph with $O(M) = N$ nodes, thus satisfying the conditions of the lemma.

All that remains is to show that we can choose such paths between the upper and lower butterflies. For each $i, j \in \{1, \ldots, m\}$ and $h \in \{0, \ldots, \frac{1}{2}\log M - 1\}$, let $p_{i,j,h}$ be the result of shuffling the bits of W^i with those of $\sigma_h(W^j)$, and let $q_{i,j,h}$ be the result of shuffling the bits of $\sigma_h(W^j)$ with those of $\sigma_1(W^i)$. Then $p_{i,j,h}$ is in upper butterfly i and $q_{i,j,h}$ is in lower butterfly j, and these two nodes are adjacent in the shuffle-exchange graph. Since all the W^i's and W^j's are from distinct nondegenerate necklaces, at most one $p_{i,j,h}$ exists in each column of upper butterfly i and at most one $q_{i,j,h}$ exists in each column of lower butterfly j. Using each path from $p_{i,j,h}$ to $q_{i,j,h}$ twice yields the desired set of paths, and the lemma follows. □

Now we note that a butterfly with $(r+1)2^{r+1} = O(N)$ nodes contains as subgraphs four disjoint half-butterflies, two in its upper half and two in its lower half. It follows that four half-butterflies can be embedded in an $O(N)$-node shuffle-exchange graph with constant load, dilation and congestion.

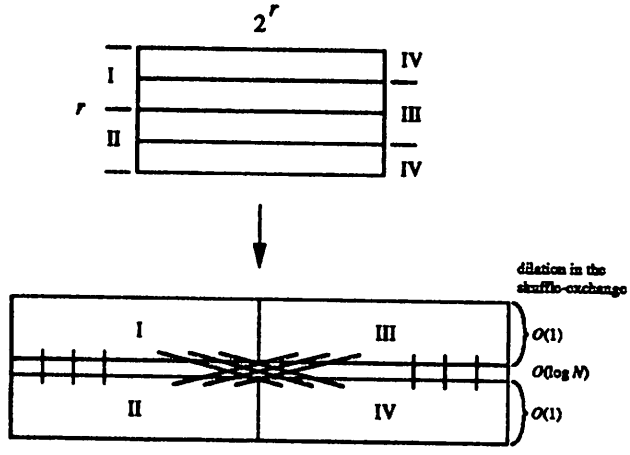

Figure 5: Mapping the butterfly nodes to the four half-butterflies.

We begin the simulation of the N-node butterfly by assigning the butterfly nodes to the four half-butterflies, as illustrated in Figure 5. One half-butterfly receives levels $0 \ldots r/2-1$ of the butterfly; the others receive levels $r/4 \ldots 3r/4-1$, $r/2 \ldots r-1$ and $3r/4 \ldots r/4-1$ respectively. Note that each node of the butterfly is assigned to two different half-butterflies, and therefore to two different locations in the shuffle-exchange graph. For each butterfly node v, we begin with the initial state of v at the two locations in the shuffle-exchange graph to which v is mapped.

It is clear that in $O(r)$ steps, we can simulate $r/8$ steps of computation on each half-butterfly. Once this is done, the middle $r/4$ levels of each embedded half-butterfly will have calculated the states of their butterfly nodes for time $t = r/8$. Thus for each butterfly node v, its state at time $r/8$ will be calculated at one of its two embedded locations; it follows that in the case where $T \leq r/8$, simulating T steps of computation on each half-butterfly is all that is necessary to effect the desired simulation. However, the first and last $r/8$ levels of each half-butterfly will not have calculated the correct states for their embedded butterfly nodes. In order to continue the simulation beyond $r/8$ steps, we must insure that for each butterfly node v, both embedded locations of v in the shuffle-exchange graph contain the state of v

at time $r/8$.

Consider the nodes $v = \langle l, C \rangle$ of the butterfly which are embedded to a node on the boundary of one of the half-butterflies; for all such v, $l \bmod r/4$ is either 0 or $r/4 - 1$. Thus every such v is also embedded to a node in the middle $r/4$ levels of some other half-butterfly, which successfully calculates the state of node v at $t = 1, \ldots, r/8$. Suppose that as these states were calculated, the pebbles $(e, 1), \ldots, (e, r/8)$ for the edges e incident to v were created and sent to the boundary location simulating v. If this was done for each such v, then the remaining levels of the half-butterflies could calculate their states for $t = 1, \ldots, r/8$ by saving their initial states and performing their calculations as the needed pebbles arrived at the boundary.

In order for this updating of the outer $r/4$ levels of each half-butterfly to take only $O(r)$ steps, we must choose paths between the two embedded locations of v for each $v = \langle l, C \rangle$ such that $l \bmod r/4$ is either 0 or $r/4 - 1$. Furthermore, each path must be of length $O(r)$ and no edge can be used more than a constant number of times. This will guarantee that all the pebbles (e, t) for the edges incident to such a v and $t = 1, \ldots, r/8$ can be delivered along these paths in $O(r)$ steps .

Since all the endpoints of these paths are contained in only eight levels of the embedded butterfly, choosing these paths reduces to the problem of routing a constant number of permutations on the columns of the butterfly, which can be accomplished with constant congestion. Since these routing paths use each level of the butterfly a constant number of times, they will have dilation $O(\log N) = O(r)$ in the shuffle-exchange graph (by Lemma 6.4).

Thus the complete simulation for $r/8$ steps proceeds as follows: Assign the initial states as described previously. Simulate $r/8$ steps of computation on each half-butterfly, calculating the states for $t = 1, \ldots, r/8$ for those nodes in the middle $r/4$ levels of each half-butterfly. As these states are calculated, those locations simulating nodes which also have images on the boundaries of the half-butterflies create the pebbles $(e, 1), \ldots, (e, r/8)$ for the edges incident to v and send them to the boundaries along the paths described above. As they arrive at the boundaries, perform the computations for the first and last $r/8$ levels of each half-butterfly. When this is complete, all locations which began with the initial state of node v will now contain the state of v at time $r/8$.

Since the above procedure takes $O(r)$ steps, iterating it will allow us to simulate any T-step butterfly computation in $O(T)$ steps on an $O(N)$-node shuffle-exchange graph. This suffices to prove the theorem. □

6.5 Application to sorting on a shuffle-exchange graph

It is known that an N-node butterfly can sort N packets with high probability in $O(\log N)$ steps [12, 17, 19]. The result 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, the emulation result of the previous section allows us to emulate this sorting algorithm on the shuffle-exchange graph and thus yields an algorithm for sorting N packets on an N-node shuffle-exchange graph in $O(\log N)$ steps with high probability.

6.6 Real time emulations of arrays

By combining the emulation result of Section 6.4 with the real-time emulation of a mesh on a butterfly in Section 5.3, we obtain an algorithm for emulating an array in real time on a shuffle-exchange graph. This is despite the fact that any $O(1)$-to-1 embedding of an N-node array (with dimension 2 or more) in a shuffle-exchange graph has dilation $\Omega(\log \log N)$ [3].

Acknowledgements

We are deeply indebted to Marc Snir for his helpful comments and for motivating this research. Thanks also to Tom Cormen for producing Figure 2, and to James Park and Joel Wein for helpful discussions.

References

[1] V. E. Benes. Permutation groups, complexes, and rearrangeable multistage connecting networks. *Bell System Technical Journal*, 43:1619–1640, July 1964.

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

[3] S. N. Bhatt, F. R. K. Chung, J.-W. Hong, F. T. Leighton, and A. L. Rosenberg. Optimal simulations by butterfly networks. In *Proceedings of the 20th Annual ACM Symposium on Theory of Computing*, pages 192–204, May 1988.

[4] S. N. Bhatt, F. R. K. Chung, F. T. Leighton, and A. L. Rosenberg. Optimal simulations of tree machines. In *Proceedings of the 27th Annual Symposium on Foundations of Computer Science*, pages 274–282. IEEE, October 1986.

[5] S. N. Bhatt and I. Ipsen. Embedding trees in the hypercube. Technical Report RR–443, Yale University, New Haven, CT, 1988.

[6] J. P. Fishburn and R. A. Finkel. Quotient networks. *IEEE Transactions on Computers*, C–31(4):288–295, April 1982.

[7] D. S. Greenberg, L. S. Heath, and A. L. Rosenberg. Optimal embeddings of butterfly-like graphs in the hypercube. *Mathematical Systems Theory*, 1990. To appear.

[8] D. Hoey and C. E. Leiserson. A layout for the shuffle-exchange network. In *Proceedings of the 1980 International Conference on Parallel Processing*, pages 329–336. IEEE, August 1980.

[9] D. J. Kleitman, F. T. Leighton, M. Lepley, and G. L. Miller. New layouts for the shuffle-exchange graph. In *Proceedings of the 13th Annual ACM Symposium on Theory of Computing*, pages 278–292, May 1981.

[10] C. P. Kruskal, L. Rudolph, and M. Snir. A complexity theory of efficient parallel algorithms. Unpublished manuscript.

[11] F. T. Leighton, M. Lepley, and G. L. Miller. Layouts for the shuffle-exchange graph based on the complex plane diagram. *SIAM Journal of Algebraic and Discrete Methods*, 5(2):202–215, June 1984.

[12] F. T. Leighton, B. M. Maggs, A. G. Ranade, and S. B. Rao. Randomized routing and sorting on fixed-connection networks. Unpublished manuscript.

[13] F. T. Leighton and G. L. Miller. Optimal layouts for small shuffle-exchange graphs. In J. Gray, editor, *VLSI 81-Very Large Scale Integration*, pages 289–299. Academic Press, 1981.

[14] 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. IEEE, October 1988.

[15] F. Meyer auf der Heide. Efficient simulations among several models of parallel computers. *SIAM Journal on Computing*, 15(1):106–119, February 1986.

[16] C. H. Papadimitriou and M. Yannakakis. Towards an architecture-independent analysis of parallel algorithms. In *Proceedings of the 20th Annual ACM Symposium on Theory of Computing*, pages 510–513, May 1988.

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

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

[19] 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.

[20] M. Sekanina. On an ordering of the set of vertices of a connected graph. *Publications of the Faculty of Science, University of Brno*, 412:137–142, 1960.

[21] D. Steinberg and M. Rodeh. A layout for the shuffle-exchange network with $\Theta(N^2/\log^{3/2} N)$ area. *IEEE Transactions on Computers*, C–30(12):977–982, December 1981.

[22] D. S. Wise. Compact layouts of Banyan/FFT networks. In H. T. Kung, B. Sproull, and G. Steele, editors, *CMU Conference on VLSI Systems and Computations*, pages 186–195, Rockville, MD, October 1981. Computer Science Press.