Work-Preserving Emulations of Fixed-Connection Networks

Richard R. Koch[1] F. T. Leighton[2,3] Bruce M. Maggs[4]
Satish B. Rao[5] Arnold L. Rosenberg[6] Eric J. Schwabe[7]

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

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

[4]School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

[5]NEC Research Institute
4 Independence Way
Princeton, NJ 08540

[6]Dept. of Computer Science
University of Massachusetts
Amherst, Massachusetts 01003

[7] Dept. of Electrical and Computer Engineering
Northwestern University
Evanston, IL 60208

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. In addition to describing several work-preserving and real-time emulations, we also provide a general model in which lower bounds can be proved. 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 proof that a butterfly can emulate a shuffle-exchange network in a real-time work-preserving fashion, and vice versa,
3. a proof that a butterfly can emulate a mesh (or an array of higher, but fixed, dimension) in a real-time work-preserving fashion, even though any $O(1)$-to-1 embedding of an N-node mesh in an N-node 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 network.

Portions of this research were conducted while Koch, Leighton, Maggs, Rao, and Schwabe were at MIT. At that time, it was supported by Defense Advanced Research Projects Agency Contract N00014–87–K–825, Office of Naval Research Contract N00014–86–K–0593, Air Force Contract OSR–86–0076, and Army Contract DAAL–03–86–K–0171. In addition, Tom Leighton was supported by an NSF Presidential Young Investigator Award with matching funds provided by IBM and Bruce Maggs and Eric Schwabe were supported by NSF Graduate Fellowships, and Arnold Rosenberg was supported by NSF Grant CCR–88–12567.

Tom Leighton is now supported by Army Contract DAAH04–95–0607 and by ARPA Contract N00014–95–1–1246. Bruce Maggs is supported by an NSF National Young Investigator Award, No. CCR-94-57766, with matching funds provided by NEC Research Institute, and by ARPA Contract F33615-93-1-1330. Eric Schwabe is supported by NSF Grant CCR–93–09111.

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.

Formally, 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.

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.

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 network. 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 network (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.

As we shall see, in order for a work-preserving emulation to be possible, it is sometimes necessary for the guest network to be substantially larger than the host. For example, we will show that a small linear array (which has a very simple structure) can perform a work-preserving emulation of a butterfly (which has a more complicated structure), but only if the butterfly is exponentially larger. In practice, however, 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.

1.1 Embeddings

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 network G into a network H, we mean a mapping $\phi : G \to H$ 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 [16] 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 network, the schedule can be computed on-line using an algorithm that works for all leveled networks [14, 16].

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 [27] that an N-node linear array can be embedded in any connected bounded-degree N-node network with load 1, dilation $O(1)$, and congestion$O(1)$. 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 N-node 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 with wraparound* 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

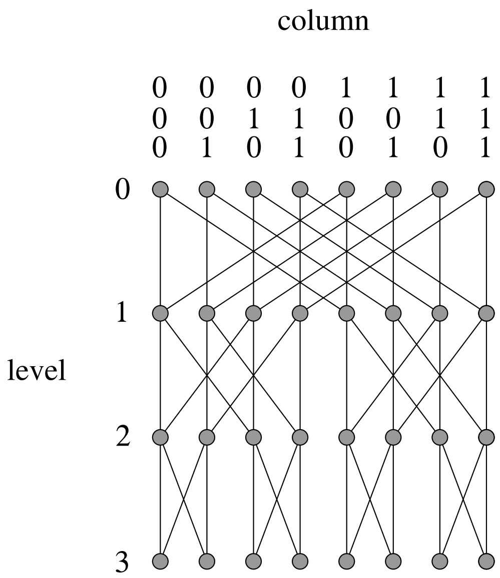

Figure 1: An 8-input butterfly without wraparound.

column C is an r-bit string. Hence $M = r2^r$. Node $\langle l, c_{r-1} \cdots c_{r-1-l} \cdots c_0 \rangle$ is connected to node $\langle l+1 \bmod r, c_{r-1} \cdots c_{r-1-l} \cdots c_0 \rangle$ by a *straight* edge, and to node $\langle l+1 \bmod r, c_{r-1} \cdots \overline{c_{r-1-l}} \cdots c_0 \rangle$ by a *cross* edge, where $\overline{c_{r-1-l}}$ denotes the complement of bit c_{r-1-l}. 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*. In the butterfly with wraparound, the inputs and the outputs are identified into a single level. 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-1} 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 an arbitrary 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.

In this paper 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.

1.2 A closer look at the computational model

If we can find an embedding of a network G into a network 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, Obrenić, Rosenberg and Schwabe [3] proved that any embedding of an N-node mesh into an N-node butterfly with constant load requires dilation $\Omega(\log N)$ (the diameter of the butterfly). At first glance, it might seem that this result implies that any emulation of an N-node mesh by an 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 the previous level of 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 network G can be computed in $O(1)$ steps on any N-node network 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 network H to emulate a network 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).

An array of dimension d and side length n has $N = n^d$ nodes. Each node in the array has a distinct label $(x_1, x_2, \ldots, x_d)$, where $0 \leq x_i \leq n-1$, for $1 \leq i \leq d$. If the array has wraparound, then for each dimension i, node $(x_1, \ldots, x_i, \ldots, x_d)$ is connected to node $(x_1, \ldots, x_i - 1 \bmod n, \ldots, x_d)$ and to node $(x_1, \ldots, x_i + 1 \bmod n, \ldots, x_d)$. If the array does not have wraparound, then for each dimension i, node $(x_1, \ldots, x_i, \ldots, x_d)$ is connected to node $(x_1, \ldots, x_i - 1, \ldots, x_d)$, unless $x_i = 0$, and to node $(x_1, \ldots, x_i + 1 \bmod n, \ldots, x_d)$, unless $x_i = n-1$. A 1-dimensional array without wraparound is called a *linear array*; with wraparound it is called a *ring*. A 2-dimensional array without wraparound is called a *mesh*; with wraparound it is called a *torus*. All of the results in this paper that are proven for meshes can be extended to arrays of higher, but fixed dimension, with or without wraparound.

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 network*, 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.

A class of networks is called an *expander* if there exists a constant $\alpha > 0$ such that if $G = (V, E)$ is an N-node network in the class, then for any set $S \subseteq V$, $|S \cup N(S)| \geq \min\{(1+\alpha)|S|, 3N/4\}$, where $N(S) = \{v \in V \mid \exists (u,v) \in E, u \in S\}$. In other words, a network is an expander if every set S has at least $\alpha|S|$ neighbors, provided that $(1+\alpha)|S| \leq 3N/4$.

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 of fixed dimension, complete binary trees, butterflies, and shuffle-exchange networks, 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 fixed 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 fixed 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 almost 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 an N-node complete binary tree can perform a work-preserving emulation of any bounded-degree tree with $O(N \log\log N)$ (or more) nodes. 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 in Section 5.1 by observing that a butterfly can perform a work-preserving emulation of any larger butterfly. Next, in Section 5.2, we show that an N-node butterfly can perform a work-preserving emulation of any bounded-degree tree of size $N \log\log N$ or larger. The emulations of butterflies and trees are relatively straightforward; the emulations in the remainder of Section 5 are more sophisticated.

In Section 5.3 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 (of 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.

Next, in Section 5.4 we describe a simple constant-congestion and constant-load embedding of an N-node shuffle-exchange network in an N-node butterfly. This result can be used to provide an elementary proof that the N-node shuffle-exchange network 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 [11], but the proof was much more difficult (as were the proofs for several suboptimal layouts for the shuffle-exchange network [10, 13, 15, 28]). The 3-d layout bound is new and was not obtainable by any of the previous approaches to the 2-d layout problem.

Finally, in Section 5.6 we describe a real-time emulation of the shuffle-exchange network 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 network. Taken together, these results resolve the long open question of whether the butterfly and shuffle-exchange network are computationally equivalent. The real-time emulation of the butterfly by the shuffle-exchange network yields several new efficient algorithms for the shuffle-exchange network. For example, we now know that a shuffle-exchange network can sort N numbers in $O(\log N)$ steps with high probability. Previously, such an algorithm was known for the butterfly [16, 24, 26], but that algorithm made crucial use of the recursive structure of the butterfly, a structure not present in a shuffle-exchange network. The emulation also yields a real-time emulation of arrays of fixed-dimension by the shuffle-exchange network.

1.5 Previous work

The notion of work-preserving emulations was previously studied by Fishburn and Finkel [8]. 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 network embeddings with the intent of showing that one network can or can't emulate another network efficiently [1, 3, 4, 5, 9, 16, 25]. 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 their significance is 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.

Embeddings in which a guest node may be mapped to several host nodes, thus allowing redundant computation, were studied by Fellows [7] and Meyer auf der Heide [17, 18, 19]. In particular, in [18] Meyer auf der Heide showed how to construct a bounded-degree network with $N^{1+\epsilon}$ nodes, for some arbitrary fixed $\epsilon > 0$, that can emulate any bounded-degree N-node network with only constant slowdown. Also, in [17] he showed that it is not possible to construct an N-node bounded-degree network that can emulate all N-node bounded-degree networks with slowdown less than $\Omega(\log N/\log\log N)$.

Work-preserving PRAM algorithms have previously been studied by Kruskal, Rudolph, and Snir [12] and served to motivate this work. Related problems of scheduling computations on fixed-connection networks have also been studied by Papadimitriou and Yannakakis [21].

2 Lower bounds

In this section we present lower bounds on slowdown and inefficiency. Loosely speaking, these lower bounds apply when the guest network expands faster than the host network. 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 network, 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 array by a k-dimensional array 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 $(v_1, t-1)$, $(v_2, t-1)$, $\ldots$, $(v_k, t-1)$, 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 network $G = (V, E)$, 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 network grows faster than the host network, then any emulation of the guest by the host, work-preserving or not, must be slow.

Theorem 2.1 *Let $H = (V_H, E_H)$ be an N_H-node host network and $G = (V_G, E_G)$ be an N_G-node guest network, 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 network 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 network 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, $T_H - k(\tau_H + 1) \geq 0$. Dividing the inequality $T_H \geq k(\tau_H + 1)$ by the inequality $T_G < (k+1)\tau_G$, we have

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

for $T_G \geq \tau_G$. □

Corollary 2.2 *Any emulation of $T_G \geq \tau_G$ steps of G by H has inefficiency*

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

Proof:
$$I = \frac{W_H}{W_G} = \frac{T_H N_H}{T_G N_G} > \frac{(\tau_H + 1) N_H}{2\tau_G N_G}.$$
$\square$

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

Proof: Since the diameter of an N_G-node complete binary tree is $\lceil \log N_G \rceil$, by choosing $\tau_G = 2\lceil \log N_G \rceil$, we can force the sum $\min_{v \in V_G} \sum_{j=1}^{\tau_G} b_G(v, j)$ to be at least $N_G \log N_G$. (For $j \geq \lceil \log N_G \rceil + 1, b_G(v, j) = N_G$.) On the other hand, the sum $\max_{u \in V_H} \sum_{i=1}^{\tau_H} b_H(u, i)$ is less than $N_G \log N_G$ for $\tau_H = \Theta\left((N_G \log N_G)^{1/(k+1)}\right)$. Hence
$$S > \frac{\tau_H + 1}{2\tau_G} = \Omega\left((N_G/\log^k N_G)^{1/(k+1)}\right).$$
$\square$

2.2 Congestion-based lower bound

The second lower bound requires a little more notation. Let $G = (V_G, E_G)$ be an undirected network as before. For a set $U \subseteq V_G$, 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 a network $H = (V_H, E_H)$ to be a partition of V_H into sets of nodes (regions) such that each contains at least R nodes and at most $2R$ nodes, and has a 1-neighborhood of size at most $f(R)$. Note that there are at least $|V_H|/2R$ and at most $|V_H|/R$ regions in an $(R, f(R))$-decomposition.

The last network 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 network 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 spend tokens in order to remove nodes from the bag. For each token spent, one node may be removed. Let X_0 denote the set of nodes in the bag initially, and let X_i denote the set of nodes in the bag at the end of round i. 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 its 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 a upper bound on the length of the longest possible game that is at most $2N_G$ and non-increasing in a. Note that since at least one node is always added to the bag in the first step of each round, and there are εa tokens to spend, the game always ends within $c + \varepsilon a$ steps. Since $c < N_G$ and $\varepsilon a < a \leq N_G$, one (usually weak) choice for $z_G(a, \varepsilon, c)$ is $2N_G$.

The game will be used in the proof of Theorem 2.4 below as follows. The nodes in the bag at the beginning of the game are nodes of G for which some region $\mathcal{R}$ of H must create pebbles for some particular guest time step t. The first step of each round brings the predecessors of the nodes in the bag into the bag. Pebbles for these nodes for the previous guest time step, $t-1$, must either be created by the region or imported from another region before the pebbles for time step t can be created. Spending a token corresponds to importing a pebble from another region. Thus, for each node left in the bag at the end of a round, the region must create a pebble. If there are a nodes in the bag at the beginning, and the player has εa tokens to spend, then the region must create c pebbles for some guest time step between t and $t - z_G(a, \varepsilon, c)$.

Theorem 2.4 *Suppose that $H = (V_H, E_H)$ is an N_H-node host network with an $(R, f(R))$-decomposition, and that $G = (V_G, E_G)$ is an N_G-node guest network. 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 $T_G > 3\beta$,

$$I \geq \min\left\{ \frac{R}{32\beta f(R)}, \frac{N_H}{192R} \right\}.$$

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 is 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. (Note that half of the t-primary pebbles are both early and late).

We begin with the definition of an importer block. Consider a block from step t to $t-3\beta+1$. Since there are at most N_H/R regions in the $(R, f(R))$-decomposition of H, the average number of t-early pebbles created by each region is at least $p = 3N_GR/4N_H$. 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 t-busy pebbles, where $s \geq p/2$. We say that the region is an *importer* if it imports a total of 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 a total of at least $3N_G/16$ pebbles for time steps between $t-1$ and $t-2\beta$. Note that in either case, if a block is an importer then 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 $T_GN_G/32\beta$. The host time required to import these pebbles is at least $T_H \geq T_GN_GR/32\beta N_Hf(R)$, because at each host time step, each of the (at most) 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 t-busy pebbles, where $s \geq p/2$, but imports fewer than $s/2$ pebbles for time steps between $t-1$ and $t-2\beta$. Since $s \geq p/2 = 3N_GR/8$ and $z_G(a, \varepsilon, c)$ is non-increasing in a, $z_G(s, 1/2, N_G/2) \leq z_G(p/2, 1/2, N_G/2) \leq \beta < 2\beta$. Thus, $\mathcal{R}$ imports at most $s/2$ pebbles for time steps between $t-1$ and $t - z_G(s, 1/2, N_G/2)$. The t-busy pebbles created by $\mathcal{R}$ cannot be created until pebbles for all of their predecessors in the pebble DAG are created. Thus $\mathcal{R}$ must create at least $N_G/2$ pebbles for some time step between t and $t - z_G(s, 1/2, N_G/2)$. Furthermore, since $\mathcal{R}$ imports a total of 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 each time step between $t - z_G(s, 1/2, N_G/2)$ and $t - 2\beta$ (and hence for each time step between $t - \beta$ and $t - 2\beta$) before creating its t-busy pebbles. For each of these time steps, at least $N_G/16$ of these $5N_G/16$ pebbles are created for nodes whose $(t-2\beta)$-primary pebbles are $(t-2\beta)$-late pebbles. We call these $N_G/16$ pebbles the *descendant* pebbles. The descendant pebbles may or may not be primary pebbles (if they are, then they are late pebbles), but by construction they are created by $\mathcal{R}$ before the t-busy pebbles for $\mathcal{R}$.

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 proof is as follows. None of the descendant pebbles are created until all of the first $N_G/4$ $(t-2\beta)$-primary pebbles are created. In order to create these $N_G/4$ pebbles, all of 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, 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 $z_G(N_G/4, 0, 3N_G/4) \leq \beta$, it follows that all of the early pebbles (and hence all busy pebbles) for all time steps at or before $t - 3\beta$ (i.e., for all previous blocks) must be created before any descendant pebble is created. But the busy pebbles for a block are not created until after the descendant pebbles for that block.

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. Since a region contains at most $2R$ host nodes, the host time for each block is at least $\beta N_G/32R$. The host time for all of the creator blocks is at least $T_GN_G/192R$ and the inefficiency is at least

$$I \geq N_H/192R.$$

Combining the two cases proves the theorem. $\square$

Corollary 2.5 *For fixed k, a k-dimensional array H cannot perform a work-preserving emulation of an expander network G.*

Proof: The proof concludes by applying 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 $I \geq \Omega((N_H/\log^k N_H)^{1/(k+1)})$. Thus, independent of the ratio N_G/N_H, the inefficiency is non-constant, and hence the emulation is not work-preserving.

Before applying the theorem, we must show that β is $O(\log(N_H/R))$. Recall that the number of nodes in the bag is given by the recurrence $X_i = X_{i-1} + \mathcal{N}_1(X_{i-1}) - Y_i$. For an expander network G, $|\mathcal{N}_1(X_{i-1})| \geq \min\{\alpha|X_{i-1}|, 3N_G/4\}$, for some constant $\alpha > 0$. The most efficient way to slow the growth of this recurrence is to spend all of the tokens during the first round, i.e., $|Y_i| = 0$ for $i > 1$. Hence,

$$\begin{aligned}
\beta &= \max\{z_G(N_G/4, 0, 3N_G/4), z_G(3N_G R/16N_H, 0, N_G/2)\} \\
&= \max\left\{\log_{(1+\alpha)/\alpha} 3, \log_{(1+\alpha)/\alpha} 32N_H/3R\right\} \\
&= O(\log(N_H/R)).
\end{aligned}$$

$\square$

Corollary 2.6 *A butterfly network H cannot perform a work-preserving emulation of an expander network 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 $I \geq \Omega(\log N_H/\log\log N_H)$. Thus, independent of the ratio N_G/N_H, the inefficiency is non-constant. $\square$

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

Proof: The proof of this thereom concludes by applying 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 $I \geq \Omega((N_H/\log^k N_G)^{1/(k+1)})$. Thus, for the inefficiency to be constant, we must have $N_G \geq 2^{\Omega(N_H^{1/k})}$. Since the slowdown S is at least N_G/N_H, we have $S \leq 2^{\Omega(N_H^{1/k})}/N_H = 2^{\Omega(N_H^{1/k})}$.

The difficult part of the proof lies in showing that $\beta = O(\log N_G)$. Since $\beta = max\{z_G(N_G/4, 0, 3N_G/4), z_G(3N_GR/8N_H, 1/2, N_G/2)\}$, 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 network, G, be an $n \log n$-node butterfly (with wraparound) 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 $2\log n + 1$ and $2\log n + t$, i.e., $\Phi_t^T = \sum_{i=1}^{t} \phi_i^T$. For the entire forest F, let $\Phi_t = \sum_{T \in F} \Phi_t^T$. Note that each butterfly node may be 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 the time at which some nodes enter the bag. For example, if a node, v, at depth l, $0 \le l \le \log n$, 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 $\log n + 1 \le l \le 2\log n$, 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, u, in T is not in the bag at the end of some time step $2\log n+t$, $t \geq 1$, at least one damage point accounts for it. The reasoning follows. If u 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 u is not in the bag at step $2\log n+t$, 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 v in the butterfly appears at most once as a tree root, twice at depth one, and so on. In general, a node v appears in trees at most 2^l times at depth l, up to a maximum of $|F|$ total appearances. A node also appears at most $|F|$ times in linear arrays. Let k_l denote the number of times a node v appears in a tree at depth l. Then, $\sum_{l=0}^{k_l} \leq |F|$, and for $0 \leq l \leq \log n$, $k_l \leq 2^l$. The total damage allotted when a token is spent to remove v is

$$\sum_{l=0}^{\log n} k_l n \log n / 2^l + \sum_{l=\log n+1}^{2\log n} k_l(2\log n - l + 1).$$

The first term is maximized when all of the appearances of v are as close to the root as possible, i.e., $k_l = 2^l$ for $l \leq \log(|F|+1)-1$ and $k_l = 0$ for $l > \log(|F|+1)+1$. Thus, the value of the first term is $O(n\log n \log|F|)$. The value of the second term is at most $|F|\log n \leq n\log n$. Hence, for $a/2 = |F|$ tokens, the total is $O(n\log n|F|\log|F|)$.

To prove that there must be at least $(3n\log n)/4$ 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 $2\log n+1$ and $2\log n+t$, Φ_t/t, is at least $(3n\log n)/4$. Recall that if no tokens are spent, $\Phi_t = n\log n\,|F|\,t$. After subtracting for damage, the average number of nodes in the bag is at least $(n\log n\,|F|\,t - O(n\log n|F|\log|F|))/t\,|F|$. Here we have divided by $|F|$ because a linear array node is counted $|F|$ times in Φ_t. For $t = c\log n$, where c is a sufficiently large constant, this average exceeds $3n\log n/4$. □

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

Proof: The proof concludes by applying 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 $I \geq \Omega((N_H^j/N_G^k)^{1/j(k+1)})$. Hence, in order for the inefficiency to be constant, we must have $N_G \geq \Omega(N_H^{j/k})$. In this case, the slowdown is at least $N_H^{(j-k)/k}$.

We bound β as follows. The number of nodes in the bag is given by the recurrence $X_i = X_{i-1} + \mathcal{N}_1(X_{i-1}) - Y_i$. For a j-dimensional array G (j fixed), $|\mathcal{N}_1(X_{i-1})| = O(|X_{i-1}|^{(j-1)/j})$. The most efficient way to slow the growth of this occurrence is to spend all of the tokens during the first round, i.e., $|Y_i| = 0$ for $i > 1$. In this case, the number of rounds is bounded by the diameter of G, i.e., $O(N_G^{1/j})$. □

3 Emulations by arrays

Although arrays cannot perform real-time emulations of networks 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 network into the array with the appropriate load, congestion, and dilation. The edges of the guest network are emulated by routing packets between the nodes of the linear array. Observations 3.2 and 3.3 were proved by Fishburn and Finkel [8]. Using Corollaries 2.3 and 2.8, the emulations of complete binary trees and other arrays can be shown to be tight in the sense that, up to constant factors, there are no work-preserving emulations with smaller slowdowns. For butterflies, the lower bound on the slowdown given in Corollary 2.7 and the slowdown of the emulation of Observation 3.4 are both exponential in the size of the array, but the exponents differ by a constant factor.

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

Proof: We show how to embed an $N^{(k+1)/k}/\log N$-leaf complete binary tree into an N-node k-dimensional array with load and dilation $O(N^{1/k}/\log N)$, and constant congestion; the desired emulation follows.

First, we embed an $N^{(k+1)/k}/\log N$-leaf complete binary tree into an N-leaf complete binary tree, by mapping the nodes of each $N^{1/k}/\log N$-leaf subtree rooted at depth $\log N$ to its root. This embedding has load $N^{1/k}/\log N$, and dilation and congestion one.

When $k = 2$, we use a result of Paterson, Ruzzo and Snyder [22] that states that an N-leaf complete binary tree can be laid out in area $O(N)$ with maximum edge length $O(\sqrt{N}/\log N)$. This layout immediately yields an embedding of an N-leaf complete binary tree into an N-node two-dimensional array with load $O(1)$, dilation $O(\sqrt{N}/\log N)$, and congestion $O(1)$. The techniques in their proof generalize in a straightforward way to yield an embedding into any N-node k-dimensional array with load $O(1)$, dilation $O(N^{1/k}/\log N)$, and congestion $O(1)$.

Combining these two embeddings yields an embedding of an $N^{(k+1)/k}/\log N$-leaf complete binary tree into an N-node k-dimensional array with load and dilation $O(N^{1/k}/\log N)$, and constant congestion. □

Observation 3.2 [8] *An N-node k-dimensional array can perform a work-preserving emulation of an $N^{j/k}$-node j-dimensional array, $j > k$.*

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

Observation 3.3 [8] *For any $M \geq N$, an N-node k-dimensional array can perform a work-preserving emulation of an M-node k-dimensional array.*

Proof: An M-node k-dimensional array can be embedded in an N-node k-dimensional array 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 array H can perform a work-preserving emulation of an $N_G = n2^n$-node butterfly network G.*

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

Because an N-node linear array can be embedded in any N-node connected network with constant load, congestion, and dilation [27], 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

In this section we examine the power of complete binary trees to emulate other trees and forests. We begin in Section 4.1 by showing that a complete binary tree can perform a work-preserving emulation of any larger complete binary tree. Next, in Section 4.2 we show that an N-node complete binary tree can perform a work-preserving emulation of any $N \log\log N$-node bounded-degree forest. Finally, in Section 4.3 we prove that any static emulation of an N-leaf complete ternary tree by an M-leaf complete binary tree, where $N < M < 3N$ has slowdown $\Omega(\sqrt{\log\log N})$. This result suggests that there is no real-time emulation of a complete ternary tree by a complete binary tree, and that the emulation of Section 4.2 is optimal.

4.1 Work-preserving emulations of larger complete binary trees

The following theorem extends a result of Fishburn and Finkel [8] 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. $\square$

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 of the class of bounded-degree forests by the class of complete binary trees with slowdown $O(\log\log N)$.

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 to a vertex in F_2, and F_1 and F_2 each contain at most $2/3$ of the total weight.*

Proof: An early proof for the unweighted case appears in [20]. The generalization to the weighted case is straightforward. $\square$

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. $\square$

Corollary 4.4 *For any k, 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. $\square$

Corollary 4.5 *Suppose that $F = (V, E)$ is a forest where each vertex has been assigned some non-negative weight. Then for any k, it is possible to remove a set S of k vertices 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, using Corollary 4.4, remove $(k-1)/2$ nodes from each of L and R 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, and let $F_1 = L_1 \cup R_2$ and $F_2 = L_2 \cup R_1$. The weight of F_1 is maximized when L_1 has $2/3$ of the total weight and R_2 has half of the remaining $1/3$ of the weight. Thus F_1 has at most $5/6$ of the total weight. A similar argument holds for F_2. The size of F_1 (and similarly F_2) is at most $|L|(1+(2/3)^{(k-1)/2})/2+|R|(1+(2/3)^{(k-1)/2})/2$ which, since $|L|+|R| \leq |V|$, is at most $|V|(1+(2/3)^{(k-1)/2})/2$. $\square$

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 4.4 to find a set S of $k = O(\log\log N)$ 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 $k = O(\log\log N)$ 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, $N_{i+1} \leq \Delta N_i + \Delta k$. 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

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

Solving the recurrence yields

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

We are now in a position to calculate the load and the congestion. Suppose by induction that $N_i \leq 12\Delta k$. (For the base case, $N_0 = 0$.) The solution to the recurrence implies that for some d such that $d = O(\log \Delta)$, we have $N_{i+d} \leq 12\Delta k$. Thus, in every simple path between a node at level i and a node at level $i+d$, where i is a multiple of d, the congestion starts at $12\Delta k$ (or less) at level i, rises to at most $O(\Delta^2 k)$ at level $i+1$ and proceeds to drop back down to at most $12\Delta k$ at level $i+d$. Since $k = O(\log\log N)$, 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 $N_i \leq 12\Delta k$. 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 Lower bounds for emulating complete ternary trees

In this section we show that any static emulation of an N-node complete ternary tree by an M-node complete binary tree, where $N < M < 3N$, must have slowdown $O(\log\log N)$. Informally, a static emulation is one in which each host node emulates a fixed set of guest nodes. Unlike a simple embedding-based emulation, however, redundant computation is allowed, i.e., several host nodes may each emulate the same guest node. The lower bound in this section holds only for static emulations, whereas those proved in Section 2 hold for even more general classes of emulations. All of the emulations described in this paper, however, are static, and the lower bound strongly suggests both that a complete binary tree cannot perform a work-preserving emulation of a complete ternary tree, and that the emulation scheme of Section 4.2 is optimal.

4.3.1 Static emulations

In a *static* emulation, a *redundant* guest network $G' = (V', E')$ is embedded in the host H. The redundant network is defined as follows. For every node v in the guest network $G = (V, E)$, there is set of nodes $\pi(v)$ in V'. Each set $\pi(v)$ contains at least one node, and for $u \neq v$, $\pi(v)$ and $\pi(u)$ are disjoint. We call the nodes in $\pi(v)$ the *instances* of v in G'. The network G' is called redundant because it may contain several instances of each guest node. For every node $v' \in \pi(v)$, and every edge (u, v) in E, the redundant network contains a directed edge (u', v'), for some $u' \in \pi(u)$. The embedding maps nodes of G' to nodes in H, and edges of G' to paths in H.

The host emulates T steps of the guest network's computation as follows. The embedding of G' into H maps a set $\psi(a)$ of nodes of G' to each host node a. Node a emulates each node $v' \in \psi(a)$ by creating a node pebble (v', t) for $1 \leq t \leq T$. A node pebble (v', t) represents the state of node v' at time t. Initially, each node a of H contains node pebbles $(v', 0)$ for $v' \in \psi(a)$. Node a can create a node pebble (v', t) only if it has already created a node pebble $(v', t-1)$, and has received all of the edge pebbles of the form $(e, t-1)$, where e is an edge (u', v') into v'. An edge pebble $(e, t-1)$ represents the communication that v' receives from its neighbor u' in step $t-1$. After creating a node pebble (v', t), a node a can create all of the edge pebbles of the form (g, t) for each edge g out of v'. At each host time step a host node a can create a single node pebble (and the corresponding edge pebbles). An edge pebble for an edge (u', v') is sent along the path from u' to v' that is specifed by the embedding. Note that a node u' may send edge pebbles to a neighbor v', but receive edge pebbles from a different instance v'' of guest node v.

The following three lemmas from [6] show that if a static emulation has slowdown s, then the load and congestion of the embedding of G' into H cannot exceed s, and the average dilation of the edges on any cycle in G' cannot exceed s.

Lemma 4.8 *[6] Suppose that there is a value $T_0 > 0$ such that for all $T > T_0$, the host can perform a static emulation of a T-step guest computation in Ts steps. Then the maximum load on any host node is at most s.*

Lemma 4.9 *[6] Suppose that there is a value $T_0 > 0$ such that for all $T > T_0$, the host can perform a static emulation of a T-step guest computation in Ts steps. Then the maximum congestion on any host edge is at most s.*

Lemma 4.10 *[6] Suppose that there is a value $T_0 > 0$ such that for all $T > T_0$, the host can perform a static emulation of a T-step guest computation in at most Ts steps. Then the average dilation of the edges on any cycle in G' is at most s.*

4.3.2 A lower bound on embeddings

The lower bound on slowdown is based on the following lower bound on the load and congestion of any embedding of an N-leaf complete ternary tree in an M-leaf complete binary tree, where $N < M < 3N$.

Lemma 4.11 *Any embedding of an N-leaf complete ternary tree T_3 in an M-leaf complete binary tree T_2, $N < M < 3N$ has either load $l > 2^{\log^\alpha N}$, for some fixed $\alpha < 1$, or has congestion $c = \Omega(\sqrt{\log\log N})$.*

Proof: The proof has the following outline. We begin by assuming that $l \leq 2^{\log^\alpha N}$ (otherwise, we're done). Next, 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 edges between S and $\overline{S}$ is at least the number of 1's in w, minus 1. 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, minus one. Next, we construct a path $v_0, v_1, \ldots, v_{(\log_2 M)/2}$ in T_2 from the root v_0 to a node $v_{(\log_2 M)/2}$ at depth $(\log_2 M)/2$ 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 the number of leaves in S, L, can be expressed as a series of $|E_S|+1$ terms, both positive and negative, where each term is a 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 from a base 3 number can increase the number of 1-digits by at most one, 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. To start, it helps to have a lower bound on the number of leaves of T_3 mapped to the subtree rooted at the last node, $v_{(\log_2 M)/2}$ on the path. Since at most $2^{\log^\alpha N}$ leaves can be mapped to any node on the path, L_i is given by the following recurrence.

$$L_i \geq \begin{cases} N & i=0 \\ \frac{1}{2}\left(L_{i-1}-2^{\log^\alpha N}\right) & i>0. \end{cases}$$

It is not difficult to show that for $0 \leq i \leq (\log_2 M)/2$, for some fixed $\gamma > 0$, and for sufficiently large N, $L_i \geq L_{i-1}(1-1/N^\gamma)/2$. Informally, this lower bound implies that embedding some leaves in nodes on the path doesn't change the load of any subtree by very much. Now let b be the number of unfair splits on the path. Since the load of the embedding is at least as large as the number of leaves of T_3 mapped to the subtree rooted at the node $v_{(\log_2 M)/2}$ on the end of the path divided by the number of nodes in the subtree $(2\sqrt{M}-1)$, we have

$$l \geq \frac{N}{2\sqrt{M}}\left(\frac{1}{2}\left(1-\frac{1}{N^\gamma}\right)\right)^{((\log_2 M)/2)-b}\left(\frac{1}{2}+\varepsilon\right)^b.$$

Since $1+x \geq e^{x/2}$ for $0 \leq x \leq 1$, $M < 3N$, and $N^\gamma > 2$ and $(\log_2(3N))/N^\gamma < 1$ for sufficiently large N, we have $l \geq \frac{1}{6e}e^{\varepsilon b}$. Let s be the length of the longest sequence of consecutive fair splits. Then $s \geq (\log_2 M)/b \geq (\varepsilon \log_2 M)/\ln(l/6e)$.

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 $j \leq i \leq j+s-1$ such that there are at least u 1's in w_i, where u is a value that will be specified later. 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

$L_i/2$ and that no leaves of T_3 are mapped to the nodes $v_0, v_1, \ldots, v_{(\log_2 M)/2}$ on the path. Since we divide L_j by two s times, $L_j \geq 2^s$. Hence, the number of significant digits, t, in the base three number w_j is at least $(\log_3 2)s$. Suppose that the number of 1's in w_j is smaller than u (otherwise we're done). The 1's in w_j partition it into at most u 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, a 0-digit follows a 2-digit (or vice versa) only if before the division, at least one of the digits was a 1. Hence a 0 follows a 2 (or vice versa) in at most most $2u$ places in w_{j+1}. If there are at least u 1's in w_{j+1}, then we're done. Otherwise, there are at most $3u$ positions in w_{j+1} in which a maximal substring consisting only of 0's or only of 2's can terminate. Thus, there must be a substring consisting only of 0's or only of 2's of length at least $(t-u)/3u$. To choose the right value for u, we set $u = (t-u)/3u$, which has solution $u = \Theta(\sqrt{t}) = \Theta(\sqrt{s})$. Whether the substring consists of all 0's or all 2's, after at most s divisions by 2 it 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. On the other hand, as many as $2^{\log^\alpha N}$ leaves may be embedded at the node v_i itself, which reduces the number of leaves embedded in its subtrees. For $\varepsilon = 1/3^t$, adding εL_i does not change the t most significant bits unless a carry propagates in. We need to show that our substring of $u = \Theta(\sqrt{t})$ 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, $3^{u/2}$ carries are required to modify the $u/2$ least significant 0's in the substring. Since at most one carry occurs at each of the s splits, and $s \ll 3^{u/2}$, the length of the longest string of all 0's or all 2's never drops below $u/2$. As we shall see, $2^{\log^\alpha N}/L_i \leq 1/3^t$. An argument similar to the one above shows that even after subtracting $1/3^t$ as many as s times, the length of the longest string of all 0's or all 2's never drops below $u/2$.

To finish, we need to choose a value for t. To make the lower bound strong, we want to make t as large as possible, while satisfying the following constraints.

1. $l \leq 2^{\log^\alpha N}$, for some fixed $\alpha > 0$,

2. $s = \varepsilon \log_2 M / \ln 6el$,

3. $\varepsilon = 1/3^t$,

4. $N < M < 3N$,

5. $2^{\log^\alpha N}/L_i \leq 1/3^t$ (for $0 \leq i \leq (\log M)/2 - 1$), and

6. $t = (\log_3 2)s$.

It is possible to satisfy all of these constraints by choosing $t = \Omega(\log\log N)$. In this case, the congestion is at least $u/2 = \Omega(\sqrt{t}) = \Omega(\sqrt{\log\log N})$. □

4.3.3 A lower bound on slowdown

We conclude with the main theorem of this section.

Theorem 4.12 *Any static emulation of an N-leaf complete ternary tree by an M-leaf complete binary tree, $N < M < 3N$, has slowdown at least $\Omega(\sqrt{\log\log N})$.*

Proof: Any redundant guest network G' for the guest G contains as a subnetwork an N-leaf complete ternary tree directed from the root to the leaves. Hence, the lower bound of Lemma 4.11 applies to the guest network G'. Since either the load is greater than $2^{\log^\alpha N}$, for some fixed $\alpha > 0$, or the congestion is greater than $\Omega(\log\log N)$, by Lemmas 4.8 and 4.9 the slowdown must be at least $\Omega(\log\log N)$. □

5 Emulations by butterfly networks

In this section we explore the ability of butterfly networks to emulate other networks. We begin in Sections 5.1 and 5.2 with short proofs that butterflies can perform work-preserving emulations of larger butterflies and bounded-degree trees. Next, in Section 5.3, we show that a butterfly can perform a real-time emulation of a mesh. This result is surprising, since any embedding of an $\Omega(N)$-node mesh into a N-node butterfly has dilation at least $\Omega(\log N)$. In Section 5.4, we prove that an N-node shuffle-exchange network can be embedded in an $O(N)$-node butterfly with constant load and congestion, and $O(\log N)$ dilation. One consequence of this result is a new and optimal three-dimensional VLSI layout of the shuffle-exchange network, which is described in Section 5.5. Finally, in Section 5.6 we prove that a butterfly network can perform a real-time emulation of a shuffle-exchange network. This result, along with the proof in Section 6.4 that a shuffle-exchange network can perform a real-time emulation of a butterfly network, resolves a long open question regarding the relative computing power of these networks.

5.1 Work-preserving emulations of larger butterflies

Theorem 5.1 [8] *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, Obrenić, Rosenberg, Schwabe 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 $O(\log\log N)$-slowdown work-preserving emulation 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 show that an N-node butterfly network can emulate an N-node mesh with constant slowdown. Rather than launching into this proof directly, we begin in Section 5.3.1 by showing that an N-node butterfly can emulate an N-node mesh with slowdown $O(\log\log N)$. This result is much easier to prove, and it illustrates the main idea used in the constant slowdown emulation: redundant computation. We conclude in Section 5.3.2 with the constant slowdown emulation. Unlike the $O(\log\log N)$-slowdown emulation, this emulation is recursive. In addition, it uses a much more sophisticated mapping of the mesh to the butterfly.

5.3.1 An emulation with slowdown $O(\log\log N)$

Theorem 5.2 *An $O(N)$-node butterfly can emulate T steps of a $\sqrt{N}\times\sqrt{N}$ mesh in $O(T\log\log N + \log N)$ steps.*

Proof: The trick is to divide the mesh into slightly overlapping submeshes, as shown in Figure 2. Each $\log^2 N \times \log^2 N$ submesh overlaps its neighbors in

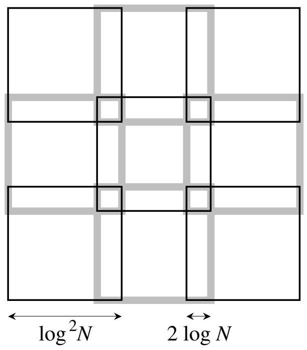

Figure 2: The division of the mesh into submeshes. Each $\log^2 N \times \log^2 N$ submesh overlaps its neighbors in either $2 \log N$ rows or $2 \log N$ columns.

either $2 \log N$ rows or $2 \log N$ columns. Since the submeshes overlap, some mesh nodes appear in as many as four submeshes. We call two nodes in neighboring submeshes *mates* if they correspond to the same mesh node.

A $\Theta(N)$-node butterfly can be broken into $\Theta(\log^4 N)$-node subbutterflies by removing all of the edges between consecutive levels every $\Theta(\log \log N)$ levels. Each $\log^2 N \times \log^2 N$ submesh is emulated by a different $\Theta(\log^4 N)$-node subbutterfly. Since a single mesh node may be emulated by several subbutterflies, the butterfly performs redundant computation.

A $\Theta(\log^4 N)$-node subbutterfly emulates a $\log^2 N \times \log^2 N$ submesh as follows. Each submesh node is emulated by a distinct subbutterfly node. For each guest time step, the subbutterfly node performs the local computation of the corresponding submesh node. In addition, for each guest time step, each submesh edge is emulated by routing a packet in the subbutterfly.

Routing in the subbutterfly is accomplished as follows. Suppose that we want to route packets in an arbitrary one-to-one fashion among all $n(1 + \log n)$-nodes of an n-input butterfly. For the moment, let us ignore the level of the destination of each packet. Viewed this way, an arbitrary permutation of packets among the nodes is a $(1 + \log n)$-to-$(1 + \log n)$ pattern among the columns. It is well known that any such pattern can be decomposed into $1 + \log n$ disjoint one-to-one patterns (permutations) among the columns (for a nice proof, see [23]). It is also well known that it is possible to establish paths with congestion 2 between the inputs of a butterfly network in any permutation (the butterfly emulates a Beneš network [2, 29]). As a consequence, it is possible to route any one permutation in $O(\log n)$ steps by first routing each packet to the input in the column of its origin, then to the input in the column of its destination, and then to its final destination. Furthermore, the routing of these permutations can be pipelined so that all $(\log n) + 1$ permutations are routed in $O(\log n)$ steps.

Each node (x, y) in a submesh has (at most) four outgoing edges. These edges lead to the nodes labeled $(x, y+1)$, $(x, y-1)$, $(x+1, y)$, and $(x-1, y)$ (if these nodes exist). We call these edges the *North*, *South*, *East*, *and West* edges. The set of North edges (or South, or East, or West edges), form a permutation. Hence, since an $O(\log^4 N)$-node subbutterfly can route any permutation of $O(\log^4 N)$ packets in $O(\log\log N)$ steps, the time to emulate the North edges for one submesh step is $O(\log\log N)$. The other three types of edges (South, East, or West) can also be emulated in $O(\log\log N)$ steps. Hence, the time to emulate one step of a submesh on the butterfly is $O(\log\log N)$.

There is one small problem with this emulation scheme: the nodes on the borders of a submesh cannot be emulated by the corresponding subbutterfly nodes because they require inputs from mesh neighbors that the subbutterfly does not emulate. As a consequence, nodes at distance δ from the border can be emulated for only δ steps. Fortunately, every node at distance $\delta \leq \log N$ from the border of one submesh has a mate at a distance of $2\log N - \delta \geq \log N$ in a neighboring submesh. Thus, every mesh node can be emulated (in at least one place) for at least $\log N$ steps in some subbutterfly.

In order to supply the missing inputs, we will route a path in the butterfly to each node on the border of each submesh from one of its mates that is a distance of $2\log N$ from the corresponding border in another submesh (and a distance of at least $\log N$ from any other border of that submesh). With a little work, we can route these paths with constant congestion. The trick is to find a mapping of the submesh nodes to the subbutterfly nodes in which at most one path endpoint lies in any column of the N-node butterfly. If such a mapping can be found, then we can route constant congestion paths by first routing the path from its origin to the input node in the same column, then routing the path to the column of its destination using Beneš-style routing, and then routing within that column to its destination.

One way to find the mapping is to choose it at random. In each $\log^2 N \times \log^2 N$ submesh, at most $8\log^2 N$ nodes are endpoints of paths (the $4\log^2 N - 4$ on the border, and the $4\log^2 N - 4$ at distance $2\log N$ from the border). Suppose that in each $\Theta(\log^4 N)$-node subbutterfly, we randomly select

$16\log^2 N$ of the $\Theta(\log^4 N/\log\log N)$ columns as candidates for the origins or destinations of paths, and then discard any candidate that is selected by more than one subbutterfly. What is the expected number of discarded columns? Each column passes through $\Theta(\log N/\log\log N)$ different subbutterflies. Each of these subbutterflies selects the column as a candidate with probability $\Theta(\log\log N/\log^2 N)$. Thus, the probability that any one column is discarded is at most $\Theta(1/\log N)$, and the expected number of columns that are discarded is $\Theta(\log N)$. One nice structural property of the butterfly network is that if two columns both pass through the same subbutterfly, then they will not both pass through any other subbutterfly. Thus, within one subbutterfly, whether or not one candidate is discarded is independent of whether or not any other candidates are discarded. As a consequence, the number of discarded candidates will be $\Theta(\log N)$ in every subbutterfly, with high probability, and the number of remaining candidates in each subbutterfly will be $16\log^2 N - o(\log^2 N) \geq 8\log^2 N$ (for N larger than some constant).

To emulate $T \geq \log N/\log\log N$ steps of the mesh, the T steps are broken into *blocks* of $\log N/\log\log N$ consecutive steps. A block is emulated as follows. Each subbutterfly begins to emulate the corresponding submesh in a step-by-step fashion. The time to emulate a single step of a submesh is $O(\log\log N)$. After each step, the nodes that are at distance $2\log N$ from the borders of the submeshes send copies of the inputs that they have received from their neighbors down the paths to their mates. These inputs will move along the paths towards their mates in a pipelined fashion, advancing across one butterfly edge every $O(1)$ butterfly steps. After $O(\log N)$ butterfly steps, the butterfly nodes emulating mesh nodes on the borders of the submeshes begin to receive their missing inputs, and begin to emulate those nodes. The emulation of a border node, and hence of an entire submesh, is completed $O(\log N)$ steps later. Thus, the time to emulate a block of $\log N/\log\log N$ steps is $O(\log N)$. Iterating this process as necessary yields the desired result. □

5.3.2 A constant slowdown emulation

In this section we show that an N-input butterfly network can emulate an $N\log N$-node mesh with constant slowdown. As in Theorem 5.2, the mesh is mapped to the butterfly in a redundant fashion. The emulation differs from that of Theorem 5.2, however, in two respects. First, subbutterflies recursively emulate submeshes. Second, the mapping is much more sophisticated, and has only constant load and congestion. The mapping is described in the following pair of lemmas.

Lemma 5.3 *For any positive integer k, there is an embedding of an m_0-node mesh in an n_0-input butterfly, where $n_0 = 2^{(2^{2k})}$ and $m_0 = n_0 \log n_0 = (2^{(2^{2k-1}+k)})^2$, with load 1, congestion $12m_0$, and dilation at most $3 \log n_0$ in which, for each mesh node, 10 additional paths are routed to arbitrarily chosen butterfly outputs.*

Proof: The nodes of the mesh can be mapped to the nodes of the butterfly in an arbitrary one-to-one pattern. For each mesh node, it is necessary to route paths to two neighbors (for example, to the North and East neighbors). If shortest paths are used, then these edges will have dilation at most $3 \log n_0$. Furthermore, since the total number of mesh nodes is m_0, the congestion due to these paths is at most $2m_0$. In addition, we must route 10 paths from each mesh node to arbitrary butterfly outputs. These paths also have dilation at most $3 \log n_0$, and, since there are $10m_0$ of them, congestion at most $12m_0$. $\square$

Lemma 5.4 *For $i \geq 0$, it is possible to find a mapping of an m_{i+1}-node mesh to an n_{i+1}-input butterfly with the following properties.*

1. *(a)* $n_{i+1} = n_i^2$,

   (b) $m_{i+1} = 2n_i m_i - (n_i m_i)^{o(1)}$, *and*

   (c) $n_{i+1} \leq m_{i+1} \leq n_{i+1} \log n_{i+1}$, *and*

2. *the m_{i+1}-node mesh is covered with overlapping m_i-node submeshes where neighboring submeshes overlap in $k_i = n_i^{1/3} \pm o(n_i^{1/3})$ rows or columns, and*

3. *(a) a path is routed from each node on a border column of a submesh to its mate in the interior of another submesh, and*

   (b) a path is routed from each node on a border row of a submesh to its mate in the interior of another submesh, and

4. *(a) a path is routed from each node on a border column of the m_{i+1}-node mesh to an output of the n_{i+1}-input butterfly, and*

   (b) a path is routed from each node on a border row of the m_{i+1}-node mesh to an output of the n_{i+1}-input butterfly, and

5. *for some value* $k_{i+1} = n_{i+1}^{1/3} \pm o(n_{i+1}^{1/3})$*,*

   (a) *two paths are routed from each node at a distance of* k_{i+1} *from a border column of the* m_{i+1}*-node mesh to outputs of the* n_{i+1}*-input butterfly, and*

   (b) *two paths are routed from each node at a distance of* k_{i+1} *from a border row of the* m_{i+1}*-node mesh to outputs of the* n_{i+1}*-input butterfly, and*

6. (a) *two paths are routed from each node in the middle column of the* m_{i+1}*-node mesh to outputs of the* n_{i+1}*-input butterfly, and*

   (b) *two paths are routed from each node in the middle row of the* m_{i+1}*-node mesh to outputs of the* n_{i+1}*-input butterfly, and*

7. *each of the preceding eight sets of paths can be routed to any set of* $n_{i+1}^{3/4}$ *consecutive outputs of the* n_{i+1}*-input butterfly in any permutation, and*

8. *the mapping has dilation* $d_{i+1} = 7 \log n_{i+1}$*, and congestion* $c_{i+1} \leq \max\{40, 12m_0\}$*, and*

9. *the mapping has load 1 at the inputs and outputs of the* n_{i+1}*-input butterfly and load* $l_{i+1} \leq 2$ *elsewhere.*

Proof: The construction is recursive. Assume that we have a mapping of an m_i-node mesh to an n_i-input butterfly that satisfies Properties 1 through 9. For $i = 0$, these properties are satisifed by the embedding of Lemma 5.3, which provides 10 paths from each mesh node to butterfly outputs, thus satisfying Properties 4 through 7.

To satisfy Property 1a, we set $n_{i+1} = n_i^2$. We can view the n_{i+1}-input butterfly as two back-to-back collections of n_i n_i-input subbutterflies. We call these two groups the *top group* and the *bottom group* Let t_i denote the ith butterfly in the top group, for $0 \leq i \leq n_i - 1$, and let b_j denote the jth butterfly in the bottom group, for $0 \leq j \leq n_i - 1$. The top group spans levels 0 through $\log n_i$ of the n_{i+1}-input butterfly, and the bottom group spans levels $\log n_i$ through levels $2 \log n_i$. The subbutterflies in each group have their outputs on level $\log n_i$. Thus, all of the inputs of the n_{i+1}-input butterfly are inputs of subbutterflies in the top group, and all of the outputs are inputs of subbutterflies in the bottom group. Note that the nodes on level $\log n_i$ appear as outputs in subbutterflies in both the top and bottom groups. In particular, the jth output of t_i is the same node as the ith output of b_j.

The first $n_i^{7/8}$ subbutterflies in the top group, $t_0, \ldots, t_{n_i^{7/8}-1}$, and the first $n_i^{7/8}$ subbutterflies in the bottom group, $b_0, \ldots, b_{n_i^{7/8}-1}$, will be used only for routing new paths. (In fact, we could pick any set of $n_i^{7/8}$ consecutive subbutterflies.) Let us call these subbutterflies the *routing* subbutterflies. In every other n_i-input subbutterfly, we will recursively map an m_i-node mesh. Let us call these subbutterflies the *computing* subbutterflies.

The routing subbutterflies by themselves form a capable routing network. First, within each subbutterfly, it is possible to route paths between the outputs in any permutation with congestion 2 and dilation $2 \log n_i$ using Beneš-style routing [2, 29]. In addition, each subbutterfly t_i shares one node with each bottom subbutterfly b_j. We call the nodes in level $\log n_i$ that are outputs of routing butterflies in both the top and the bottom groups *shared nodes*. Since there are $n_i^{7/8}$ routing subbutterflies in the top group, and each of these shares a distinct output with each of the $n_i^{7/8}$ routing subbutterflies in the bottom group, there are a total of $n_i^{7/4}$ shared nodes. Suppose that we wish to route a collection of paths from the $n_i^{7/4}$ shared nodes back to themselves in some arbitrary permutation. Let us view these nodes as being arranged on an $n_i^{7/8} \times n_i^{7/8}$ grid, where the nodes in each row of the grid belong to the same top subbutterfly and the nodes in each column belong to the same bottom subbutterfly. It is well known that any permutation on the nodes of the grid can be written as the composition of three permutations: one on the rows, one on the columns, and one more on the rows. The first and third permutations can be performed by the top subbutterflies, and the second by the bottom subbutterflies. The routing paths will have congestion 4 and dilation at most $6 \log n_i$, and will reside entirely within the routing subbutterflies.

In order to satisfy Property 2, we cover the m_{i+1}-node mesh with m_i-node meshes in an overlapping fashion. Let us call the m_i-node meshes *submeshes*. Each submesh in the covering overlaps each of its (at most) 8 neighboring submeshes in $k_i = n_i^{1/3} \pm o(n_i^{1/3})$ rows or columns. We place the first m_i-node submesh on the m_{i+1}-node mesh so that its middle row and column coincide with the middle row and column of the m_{i+1}-node mesh, and proceed outward. Let h_i denote the number of submeshes used to cover the entire m_{i+1}-node mesh. Since there are $2(n_i - n_i^{7/8})$ computing subbutterflies, $h_i = 2(n_i - n_i^{7/8})$. Using h_i submeshes, we can cover a mesh of size at least

$$\begin{aligned} m_{i+1} &\geq h_i(\sqrt{m_i} - n_i^{1/3} - o(n_i^{1/3}))^2 && (1) \\ &\geq h_i(\sqrt{m_i} - 2n_i^{1/3})^2 && (2) \\ &\geq 2n_i m_i - 2n_i^{7/8} m_i - 8\sqrt{m_i} n_i^{4/3} - 8n_i^{37/24} && (3) \\ &\geq 2n_i m_i - (n_i m_i)^{o(1)}. && (4) \end{aligned}$$

Inequality 2 holds provided that that n_0 and m_0 (and hence n_i and m_i) are sufficiently large, and Inequality 4 holds because inductively $m_i \leq n_i \log n_i$ by Property 1c. The largest mesh that can be constructed from h_i m_i-node submeshes has size at most

$$\begin{aligned} m_{i+1} &\leq h_i m_i && (5) \\ &= 2(n_i - n_i^{7/8}) m_i && (6) \\ &\leq 2(n_i - n_i^{7/8}) n_i \log n_i && (7) \\ &\leq n_{i+1} \log n_{i+1}. && (8) \end{aligned}$$

Inequalities 4 and 6 imply Property 1b. Furthermore, since $m_i \geq n_i$ and $n_{i+1} = n_i^2$, Inequalities 4 and 8 imply Property 1c, for sufficiently large n_i and m_i.

In order to satisfy Property 3a, a path must be routed between each node that is on a border column in one m_i-node submesh and the corresponding copy of the node that is a distance of $n_i^{1/3} \pm o(n_i^{1/3})$ from the border column of an overlapping submesh. Each submesh has $2\sqrt{m_i}$ nodes on its border columns and $2\sqrt{m_i}$ nodes that are a distance of $k_i = n_i^{1/3} \pm o(n_i^{1/3})$ from the border columns. A node that is a distance of k_i from a border column may have a mate on the border column of two different submeshes (and three if the node is also a distance of k_i from a border row). Hence, to satisfy Property 3a, we may have to route

$$\begin{aligned} 6\sqrt{m_i} &\leq 6\sqrt{n_i \log n_i} && (9) \\ &\leq n_i^{3/4}/2 && (10) \end{aligned}$$

paths out of each submesh. (Inequality 10 holds provided that n_0 and m_0, and hence n_i and m_i, are sufficiently large.) Since the number of submeshes, h_i is smaller than $2n_i$, the total number of paths to route is at most $n_i^{7/4}$.

We can assume inductively (via Properties 4a, 5a, and 7) that, for each submesh, it is possible to route these paths in any permutation to any set of $n_i^{3/4}/2$ consecutive outputs of the corresponding n_i-input subbutterfly. So, in each computing subbutterfly we choose to route these paths to the first set of $n_i^{3/4}/2$ subbutterfly outputs, which are shared with the routing subbutterflies. In choosing the outputs, we must also ensure that at most $n_i^{7/8}$ paths are sent to any one routing subbutterfly, but this is easy. Hence, we can use the routing subbutterflies to connect corresponding pairs of paths with dilation $6 \log n_i$, and congestion 4. Thus Property 3a is satisfied. Property 3b can be satisfied in a similar fashion.

In order to satisfy Property 4a, we must route a path from each node that is on a border column of the m_{i+1}-node mesh to one of the first $n_{i+1}^{3/4}$ outputs of the n_{i+1}-input butterfly. A node on a border column of the m_{i+1}-node mesh is also on a border column of some m_i-node submesh. Because a node on a border column of the m_{i+1}-node mesh is missing a neighbor on either its East or West side, the corresponding node on the border column of the m_i-node submesh does not have a mate that is a distance of k_i from the border column of another m_i-node submesh. (Unless that mate also happens to be a distance of k_i from a border *row* of another submesh.) Hence inductively there is a path from each node on a border column of the m_i-node mesh to an output of the corresponding n_i-input subbutterfly that was not used when we established Property 3a earlier. This spare path is used to reach an input of a routing subbutterfly. The routing subbutterflies are then used to connect the path to an output of the n_{i+1}-input butterfly. The paths have dilation $7 \log n_i$ and congestion 5. Thus, Property 4a is established. The proof for Property 4b is similar.

To satisfy Property 5a, we must route paths from the nodes that are at a distance of $k_{i+1} = n_{i+1}^{1/3} \pm o(n_{i+1}^{1/3})$ from a border column of the m_{i+1}-node mesh to outputs of the n_{i+1}-input butterfly. Here we have some freedom in choosing k_{i+1}, and we will choose a value that maps each node at distance k_{i+1} from a border column of the mesh onto the middle column of an m_i-node submesh. We will then use the fact that we can inductively route two paths from each node on the middle column of any submesh to the outputs of the corresponding subbutterfly. In particular, we route paths from the nodes in the middle column of the submesh that contains the column that is at a distance of $n_{i+1}^{1/3}$ from the border of the m_{i+1}-node mesh. This column is at a distance of $n_{i+1}^{1/3} \pm (\sqrt{m_i}/2) = n_{i+1}^{1/3} \pm o(n_{i+1}^{1/3})$ from a border column of the m_{i+1}-node mesh. The paths from these nodes to the outputs of the corresponding subbutterflies exist due to the inductive assumption of Properties 6a and 7. Extending these paths via the routing butterflies establishes Property 5a. Property 5b can be proved in a similar way.

To satisfy Property 6a, it is necessary to route paths from the nodes in the middle column of the m_{i+1}-node mesh to outputs of the butterfly. By our arrangement of the overlapping submeshes, these nodes also lie in the middle columns of submeshes, and hence we again use the fact that we can recursively route paths from the middle column of any submesh to the outputs of the corresponding subbutterfly. We note that particular nodes from which we route are all distinct from those used in the previous paragraph to establish Property 5a, so that none of the paths in the subbutterflies are reused. Thus we ensure Property 6a. The proof for Property 6b is similar.

Now that all of the paths required in Properties 4 through 6 have been routed to the routing butterflies, we note that they can be permuted arbitrarily within the routing butterflies to ensure Property 7.

The bounds on the congestion, dilation, and load required by Properties 8 and 9 are derived as follows. The new paths in the embedding use edges only in the routing subbutterflies. The congestion due to these paths is at most 40, since establishing each of the 8 sets of paths specified by Properties 3 through 6 requires congestion at most 5. Since nothing else is embedded in these subbutterflies, the total congestion is $c_{i+1} = \max\{40, 12m_0\}$. The dilation is $7\log n_i + 7\log n_i = 7\log n_{i+1}$, since each recursively-defined path was of length $7\log n_i$ and no path was extended by more than an additional $7\log n_i$. The load at the input and output nodes of the n_{i+1}-input butterfly is 1, because these nodes are also inputs and outputs of n_i-input subbutterflies, which by induction have load 1. The load of the shared nodes in level $\log n_i$ is at most 2 because although each of these nodes appears in both a top and bottom subbutterfly, they are outputs in both. $\square$

Theorem 5.5 *An N-input butterfly can emulate an $N\log N$-node mesh with constant slowdown.*

Proof: For simplicity, let us assume that $N = 2^{(2^{2k})}$, for some positive integer k. With some effort it is possible to remove this assumption, but the details of the proof become (even more) messy.

The proof uses Lemma 5.4 to recursively map an $N\log N$-node mesh to an N-input butterfly. For $i \geq 0$, the sizes of the butterfly and mesh are given by the recurrences

$$
\begin{aligned}
n_{i+1} &= n_i^2 && (11)\\
m_{i+1} &= 2n_i m_i - (n_i m_i)^{o(1)}. && (12)
\end{aligned}
$$

Solving these recurrences yields $m_i = n_i \log n_i - o(n_i \log n_i)$. Hence, for $N = n_i$, an $(N \log N - o(N \log N))$-node mesh is embedded in an N-input butterfly. (The mesh can easily emulate an $N \log N$-node mesh with constant slowdown.)

The running time can be bound as follows. Let $T(N)$ denote the time to emulate $N^{1/3}/2$ mesh steps on an N-input butterfly. Then $T(N)$ is given by the recurrence $T(1) = O(1)$ and $T(N) = N^{1/6}T(\sqrt{N}) + O(\log N)$. This recurrence has the following derivation. The N-input butterfly begins to emulate each m_{i-1}-node submesh in an n_{i-1}-input subbutterfly, where $n_{i-1} = \sqrt{N}$ and $m_{i-1} = n_{i-1} \log n_{i-1} - o(n_{i-1} \log n_{i-1}) = (\sqrt{N} \log N)/2 - o(\sqrt{N} \log N)$. Nodes that are a distance of k_{i-1} from the border of a submesh, where $k_{i-1} = n_{i-1}^{1/3} \pm o(n_{i-1}^{1/3} = N^{1/6} \pm o(N^1/6)$, immediately begin to send packets to their mates on the borders of other submeshes. These packets begin to arrive at the border nodes after $O(\log N)$ steps, and the emulation ends after each of the border nodes has completed its computation. The border nodes complete their computations within $(N^{1/3}/2)/(N^{1/6}/2))T(\sqrt{N}) = N^{1/6}T(\sqrt{N})$ steps, because this is the time for the $\sqrt{N}$-input subbutterflies to recursively emulate $N^{1/3}/2$ steps of the corresponding submeshes. This recurrence has solution $T(N) = O(N^{1/3})$. □

5.4 Embedding the shuffle-exchange network in the butterfly

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

In a constant congestion embedding, very few edges of the shuffle-exchange network can be mapped to long (i.e., greater than constant length) paths in the butterfly. In addition, the long paths must not overlap each other very often. To this end, our embedding satisfies the following two Conditions.

1. At most two shuffle-exchange nodes are embedded in any one butterfly node, and
2. in each butterfly column, at most 16 shuffle-exchange nodes have a neighbor that is embedded more than distance 2 away in the butterfly.

Condition 1 ensures that the load is constant. To see why Condition 2 ensures that the congestion is constant, it helps to consider the long (i.e., dilation greater than 2) and short paths separately. The short paths contribute only constant congestion because the load of the embedding is constant, the degrees of the shuffle-exchange and butterfly networks are constant, and the lengths of the short paths are constant. Hence, at most a constant number of short paths can reach any butterfly edge. The long paths can be routed with constant congestion (and $O(\log N)$ dilation) because the inputs and outputs of a Beneš network can be connected in any permutation by a set of disjoint paths [2], and a Beneš network is simply two back-to-back butterfly networks. Thus, 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. When there are at most a constant number of endpoints of long paths in any single butterfly column, we can first route a path from each endpoint to the input of its column, which leaves us with a constant number of permutations to route from the inputs.

Our embedding maps the nodes of an N-node shuffle-exchange network (where $N = 2^n$) to the nodes of an $(n+3-\log n)2^{n+3-\log n}$-node butterfly (with wraparound). Note that $(n+3-\log n)2^{n+3-\log n} \approx 8N$. Each node in the N-node shuffle-exchange network has n bits in its label. A node in the butterfly can be specified by a column represented by an $(n+3-\log n)$-bit string $x_{n+2-\log n}\cdots x_0$, and a level in the range $[0, n+2-\log n]$. An edge from level i to level $i+1$ in the butterfly network connects two nodes whose column strings differ only in the bit with index $n+2-\log n-i$.

The key to proving that the mapping of the shuffle-exchange network to the butterfly network has load 2 is to show how to invert the embedding. Hence, in defining the rules for mapping shuffle-exchange nodes to butterfly nodes, we want to use operations that are easy to undo. We begin by associating a shuffle-exchange node with a particular column of the butterfly. The column is chosen by removing $\log n$ consecutive bits from the node's label, none of which is the least significant bit. The level in the butterfly is then determined by the position of the string that was removed. In particular, we associate a shuffle-exchange node w with a column C of the butterfly as follows,

1. Find the longest string of consecutive zeros in w that does not include the least significant bit. Break ties by choosing the leftmost.

2. Pick out a string a_s of $\log n$ bits as follows.

(a) If possible choose the $\log n$ bits following the zeros and preceding the least significant bit,

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

(c) otherwise choose the last $\log n$ bits of the string of zeros. (Note that in this case there is a unique longest string of zeros, and it has length at least $n - 2\log n$.)

3. Remove a_s from w. (Note that if a_s lies inside the longest string of zeros, then after its removal there remains a string of at least $n - 3\log n$ consecutive zeros.)

4. Extend the string of zeros on the left by a 1 and on the right by 01. (The string now has length $n + 3 - \log n$ and contains a unique longest string of zeros.) Call the new string x.

5. Treat the bits of a_s as a number, s. (Since there are $\log n$ bits in s, the number will lie in the range $[0, n-1]$.) Perform a cyclic shift on x so that exactly $s+1$ bits appear after the string of zeros. The resulting string is C.

Symbolically, we map a shuffle-exchange node with label $w = z0^k a_s yb$ to column $C = u10^{k+1}1v$, where u, v, w, z, a_s, and C are strings, k is an integer, b is a single bit, $ybz = vu$, and $|v| = s$, or we map $w = za_s0^kyb$ to column $C = u10^{k+1}1v$. Note that, by construction, C contains a unique longest string of zeros (allowing wraparound) of length $k+1$, and the bit b does not lie inside the string of zeros.

In mapping a shuffle-exchange node to a butterfly node, we choose the level l to mark the position of the least significant bit b in C. In particular, if b is in bit position p, then $l = n + 3 - \log n - p$.

Before proceeding, let us introduce a little notation. We define a *necklace* to be a maximal set of shuffle-exchange nodes that are connected only by shuffle edges. Alternatively, a necklace can be viewed as a set of nodes whose labels are identical up to cyclic shifts. The *label* of a necklace is the lexicographical minimum of the labels of its nodes. We can specify a shuffle-exchange node by the label of its necklace and the position in the necklace's label of the nodes least significant bit. We define the *domain* of a butterfly node to be the set of shuffle-exchange nodes that are mapped to it.

We now prove that Condition 1 is satisfied. That is, given a butterfly node $\langle l, C\rangle$ we show that at most two shuffle-exchange nodes can possibly be mapped to $\langle l, C\rangle$. Recall that if a shuffle-exchange node labeled w is mapped to a butterfly node labeled $\langle l, C\rangle$, then all of the bits of w appear in the string C except for a_s. The missing bits can be recovered by finding the length, s, of the string that lies between the longest string of consecutive zeros and the pth bit of C (where $p = n + 3 - \log n - l$). We know that a_s belongs either directly before or directly after the zeros. Thus, there are at most two possible choices for the bits in w (ignoring cyclic shifts). The domain of $\langle l, C\rangle$, therefore, consists of nodes from at most two necklaces. The value and position of the least significant bit, b, of a node's label can be determined by examining the pth bit of C. Thus only two shuffle-exchange nodes can be mapped to any node in the butterfly.

To conclude, we show that, within any column of the butterfly, at most a 16 edges lead from shuffle-exchange nodes to distant neighbors.

The exchange edges can easily be shown to have dilation two. Notice that when mapping a shuffle-exchange node to a butterfly node, the value of the least significant bit in the shuffle-exchange node is ignored. Thus if two shuffle-exchange nodes are connected by an exchange edge, their images in the butterfly are $\langle C, l\rangle$ and $\langle C', l\rangle$, where C and C' differ only in bit position $p = n + 3 - \log n - l$). The bit that differs can be changed by traversing a cross edge from level l in column C to level $l + 1$ in column C', then a straight edge within column C' to return to level l. Thus, an exchange edge is embedded in a path of length 2, where the first edge on the path is used to change the least significant bit and the second edge is used to return to the original level in the butterfly.

We now prove that at most 16 shuffle edges leave any column of the butterfly. A shuffle edge connects two nodes in a necklace whose labels differ by a rotation of one bit position. The two nodes are mapped to the same necklace if the rotation does not affect the choice of the longest string of consecutive zeros, and does not change the position of a_s relative to the zeros. In this case, the two nodes are mapped to a pair of butterfly nodes in the same column that are connected by a butterfly edge. There are eight possible circumstances, however, in which two nodes connected by a shuffle edge may be mapped to different columns. First, if the leftmost bit of a node's label belongs to the longest string of zeros, then a left rotation may change the length or position of the longest string of zeros. Second, if exactly $\log n$ bits precede the zeros, then a left rotation may change the position of a_s. Third, if the bit immediately to the left of the least significant bit b belongs to the string of zeros, then a right rotation may change the length or position of the longest string of zeros. Finally, if there are precisely $\log n$

bits between the string of zeros and b, then a right rotation may change the position of a_s. In addition to these four cases, the opposite rotation in each case may also lead to a different column. Hence, for any necklace mapped to C, there are at most 8 shuffle edges that leave the column. Since at most two necklaces are mapped to any column, the number of shuffle edges that leave the column is at most 16.

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

Theorem 5.6 *An N-node shuffle-exchange network can be embedded in a $\Theta(N)$-node butterfly network with load $O(1)$, congestion $O(1)$, and dilation $O(\log N)$.*

5.5 Layouts for the shuffle-exchange network 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 [30]. Since the N-node shuffle-exchange network 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 network with equivalent area and volume.

5.6 A real-time emulation of the shuffle-exchange network

In this section, we prove the following theorem:

Theorem 5.7 *T steps of an arbitrary computation on an N-node shuffle-exchange network 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 network nodes to the nodes of the butterfly so that after $O(n)$ steps, each node that 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 $\mathcal{S}$ 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

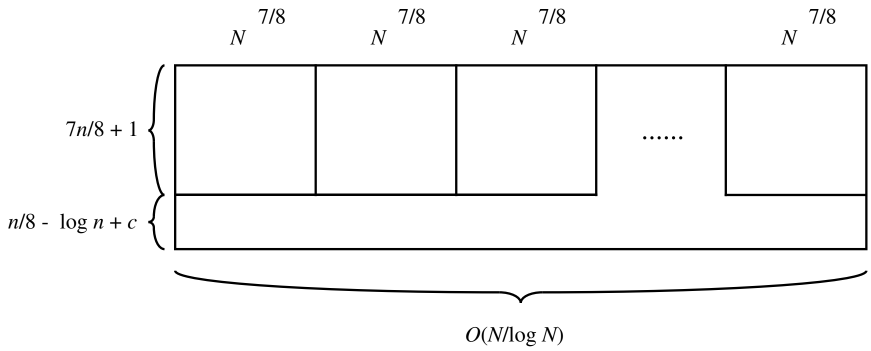

Figure 3: Embedding the set of butterflies indexed by $\mathcal{S}$ in one large butterfly ($n = \log N$).

longest run of zeroes and are lexicographically least among their cyclic shifts, the number of strings in $\mathcal{S}$ is $O(2^{n/8}/n) = O(N^{1/8}/\log N)$.

Next we observe that $O(N^{1/8}/\log N)$ butterflies without wraparound consisting of $\frac{7}{8}\log N + 1$ levels and $2^{\frac{7}{8}\log N} = N^{7/8}$ columns each can be embedded in an $O(N)$-node butterfly with unit 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. This is done by lining up the butterflies, with their columns interleaved, as the top $\frac{7}{8}\log N + 1$ 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. Figure 3 illustrates this embedding of the collection of butterflies into one large butterfly; for the sake of clarity, the small butterflies are not interleaved in the figure.

We will map the nodes of the shuffle-exchange network 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 network 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 $\langle j - n/8, x_{j-n/8} \cdots x_1 x_n \cdots x_{j+1}\rangle$ in butterfly $x_j \cdots x_{j+1-n/8}$. Thus X is mapped to as many locations as there are values of j that 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 $Y \in \mathcal{S}$, $l \in \{0, \ldots, 7n/8\}$ 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, $x_{l+n/8} \cdots x_{l+1} = Y$. Furthermore, we must have $x_l \cdots x_1 x_n \cdots x_{l+n/8+1} = C$, which determines the remaining bits of X. Thus for each node in the collection of butterflies, there is only one shuffle-exchange node X that could possibly be mapped to that node (though there may in fact be none mapped there).

The following lemma shows that any image of a node that does not reside 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 5.8 *Let $g(X)$ be an image of shuffle-exchange node X in butterfly Y that is in a level other than $n/8$ or $3n/4$ of Y. Then all the neighbors of X in the shuffle-exchange network 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 $j - n/8$ and column $C = x_{j-n/8} \cdots x_1 x_n \cdots x_{j+1}$. It is easy to verify that $x_{n-1} \cdots x_1 x_n$ and $x_1 x_n \cdots x_2$ (the neighbors of X across shuffle edges) are mapped to $\langle j - n/8 + 1, C\rangle$ and $\langle j - n/8 - 1, C\rangle$ in butterfly Y, and that $x_n \cdots x_2 \overline{x_1}$ (the neighbor of X across an exchange edge) is mapped to $\langle j - n/8, C'\rangle$ in butterfly Y, where C' differs from C in only the $(j - n/8 + 1)$th bit from the left. This proves the lemma. $\square$

For each X, assign the initial state of node X to all the nodes in $G(X)$. From Lemma 5.8 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 that 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 substring 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

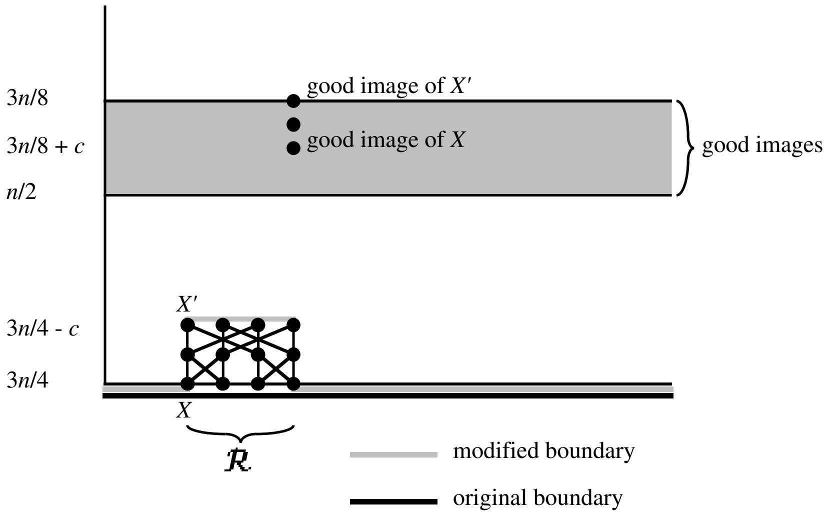

Figure 4: Adjusting the boundary for X and those nodes which differ from X in the $n/8$ lowest order bits. Note that all good images remain at distance at least $n/8$ from the boundary.

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 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 network 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 5.9 *T steps of an N-node shuffle-exchange network 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 [9]. □

6 Emulations by shuffle-exchange networks

6.1 Work-preserving emulations of larger shuffle-exchange networks

Theorem 6.1 [8] *For any $M \geq N$, an N-node shuffle-exchange network can perform a work-preserving emulation of an M-node shuffle-exchange network.*

Proof: An M-node shuffle-exchange network can be embedded in an N-node shuffle-exchange network 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 network 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 network. Whether or not this emulation can be made real-time remains an open question.

6.3 Embedding small butterflies in the shuffle-exchange network

In this section we show how to embed $M/\log M$ distinct $M\log M$-node butterfly networks in an $N = M^2$ shuffle-exchange network with load $l = 2$, congestion $c = O(1)$, and dilation $d = 3$. A similar result was proved by Raghunathan and Saran [25]. 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 network can be represented by a $2k$-bit string.

To map $M/\log M$ butterflies to the shuffle-exchange network, 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 $c_{\log M-1}\cdots c_0$, and cyclically shifting the string so that the image of $c_{\log M-p-1}$ is at the rightmost bit position. In other words, for $W^i = w^i_{\log M-1}\cdots w^i_0$, node $\langle p, C\rangle$ of butterfly i is mapped to shuffle-exchange node

$$w^i_{\log M-p-2}c_{\log M-p-2}\cdots w^i_0 c_0 w^i_{\log M-1}c_{\log M-1}\cdots w^i_{\log M-p-1}c_{\log M-p-1}.$$

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 cyclic shifts needed 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 network 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 $\langle p+1, c_{\log M-1}\ldots\overline{c_{\log M-p-1}}\ldots c_0\rangle$.

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

Proof: We will show how to map the initial states of the butterfly nodes to the nodes of the shuffle-exchange network so that after $O(r)$ steps, each node that 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 network 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 network 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 network 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 network 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. In the following, we use $\sigma_k(X)$ to denote the result of cyclically shifting the bits of the string X to the left k positions. 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 network. Since all the W^i's and W^j's are from distinct nondegenerate necklaces (that is, from distinct necklaces each containing $\log M$ distinct shuffle-exchange nodes), 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

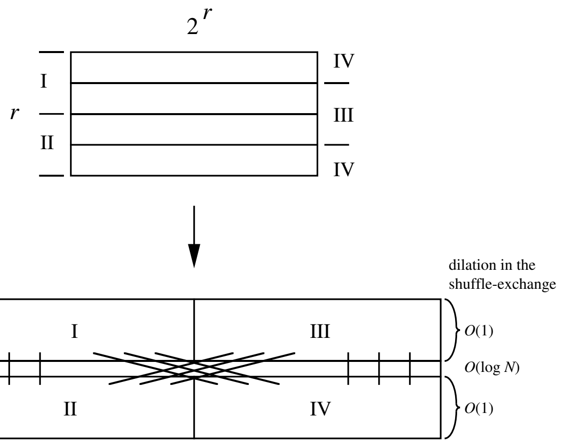

Figure 5: Mapping the butterfly nodes to the four half-butterflies.

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 subnetworks 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 network with constant load, dilation and congestion.

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 network. For each butterfly node v, we begin with the initial state of v at the two locations in the shuffle-exchange network 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 network contain the state of v at time $r/8$.

Consider the nodes $v = \langle l, C \rangle$ of the butterfly that 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 within 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 by all the paths together. 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 network (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 that 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 network. This suffices to prove the theorem. $\square$

6.5 Application to sorting on a shuffle-exchange network

It is known that an N-node butterfly can sort N packets with high probability in $O(\log N)$ steps [14, 24, 26]. The result does not directly extend to the shuffle-exchange network because the shuffle-exchange network 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 network and thus yields an algorithm for sorting N packets on an N-node shuffle-exchange network 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 network. 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 network 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. We also thank Alf-Christian Achilles and Bruce Parker for pointing out an error in an earlier version of the proof of Theorem 5.5.

References

[1] M. J. Atallah. On multidimensional arrays of processors. *IEEE Transactions on Computers*, 37(10):1306–1309, October 1988.

[2] V. E. Beneš. *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 Computer Society Press, 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] R. J. Cole, B. M. Maggs, and R. K. Sitaraman. On the benefit of supporting virtual channels in wormhole routers. In *Proceedings of the 8th Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 131–141, 1996.

[7] M. R. Fellows. *Encoding Graphs in Graphs*. PhD thesis, Department of Computer Science, University of California, San Diego, CA, 1985.

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

[9] D. S. Greenberg, L. S. Heath, and A. L. Rosenberg. Optimal embeddings of butterfly-like graphs in the hypercube. *Mathematical Systems Theory*, 23:61–77, 1990.

[10] 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 Computer Society Press, August 1980.

[11] 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.

[12] C. P. Kruskal, L. Rudolph, and M. Snir. A complexity theory of efficient parallel algorithms. In *Proceedings of the 15th International Colloquium on Automata, Languages and Programming.* Volume 317 of

Lecture Notes in Computer Science, pages 333–346. Springer–Verlag, 1988.

[13] 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.

[14] F. T. Leighton, B. M. Maggs, A. G. Ranade, and S. B. Rao. Randomized routing and sorting on fixed-connection networks. *Journal of Algorithms*, 17(1):157–205, July 1994.

[15] 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.

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

[17] F. Meyer auf der Heide. Efficiency of universal parallel computers. *Acta Informatica*, 19:269–296, 1983.

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

[19] F. Meyer auf der Heide and R. Wanka. Time-optimal simulations of networks by universal parallel computers. In *Proceedings of the 6th Symposium on Theoretical Aspects of Computer Science*, pages 120–131, 1989.

[20] P. M. Lewis II, R. E. Stearns, and J. Hartmanis. Memory bounds for recognition of context-free and context-sensitive languages. In *Proceedings of the IEEE Symposium on Circuit Theory and Logical Design*, pages 191–202, 1965.

[21] 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.

[22] M. S. Paterson, W. L. Ruzzo, and L. Snyder. Bound on minimax edge length for complete binary trees. In *Proceedings of the 13th Annual ACM Symposium on Theory of Computing*, pages 293–299, May 1981.

[23] N. Pippenger. Telephone switching networks. In *Proceedings of Symposia in Applied Mathematics*, volume 26, pages 101–133. American Mathematical Society, 1982.

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

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

[26] 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.

[27] 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.

[28] 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.

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

[30] 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.