Emulating Direct Products by Index-Shuffle Graphs

Bojana Obrenić
Department of Computer Science
Queens College and Graduate Center of CUNY
Flushing, NY 11367
obrenic@sunset.cs.qc.edu

Abstract

In the theoretical framework of graph embedding and network emulations, we show that the index-shuffle graph (a bounded-degree hypercube-like interconnection network, recently introduced by [Baumslag and Obrenić (1997): Index-Shuffle Graphs, ...]) efficiently approximates the hypercube in general computations, by emulating the direct-product structure of the hypercube.

In the direct product $G = G_1 \times G_2 \times \cdots \times G_k$ let any factor G_i be an instance of any of the three following graphs: cycle, complete binary tree, X-tree. Given an N-node index-shuffle graph Ψ_n, where $N = 2^n$, and any collection of 2^ℓ copies of G, such that: $|G_i| \leq 2^{n_i}$, for $i = 1, \ldots k$, where $\ell + \sum_{i=1}^{k} n_i \leq n$ and $2^{\lceil \log_2 k \rceil} \cdot (\max_{1 \leq i \leq k} n_i) \leq n$, Ψ_n emulates any factor G_i in all copies of G in this collection with slowdown $O(\log k + \log n_i) = O(\log \log N)$.

As a consequence of these and previous results, the index-shuffle graph emerges as a uniquely "universal" bounded-degree hypercube substitute. The index-shuffle graph emulates (multiple copies of) **multi-dimensional tori and meshes of trees (***or X***-trees)** *with slowdown* **doubly logarithmic** *in the size of the graph, which currently cannot be achieved by either butterflies or shuffles. Furthermore, the* **butterfly** *is emulated by its (like-sized) index-shuffle graph with slowdown* **triply logarithmic** *in the size of the graph, which is currently impossible by shuffles. Finally, the index-shuffle graph* **contains** *the (equal-sized)* **shuffle-exchange** *graph, thus demonstrating communication power not known to be present in the hypercube itself.*

1. Introduction

Bounded-Degree Hypercube Approximations. Various bounded-degree graph families are derived from the hypercube in an attempt to achieve the efficiency of the hypercube while keeping the graph degree constant and low. The families of *butterfly-like graphs* (cf. [21, 15]) and *shuffle-like graphs* (cf. [25, 24, 10]) have a low diameter and high bisection width, which enable them to execute algorithms which demand fast but arbitrary one-to-one communication among the processors. Several other families have found their application areas, although they are inferior to butterflies and shuffles in their ability to perform the task of routing arbitrary message permutations because of their relatively high diameter or relatively low bisection width. Such are the families of *meshes* (cf. [18]), *meshes of trees* (cf. [16, 17]), and *X-trees.* (cf. [1, 2])

The Lack of Universality in Hypercube Substitutes. While standard families of hypercube approximations jointly cover a broad area of hypercube applications, none of them is known to be able to subsume the communication power of the others in general computations. In some cases procedures are known for adapting parallel programs written for one of these networks so as to be subsequently executed efficiently on some of the others. However, these adaptations require qualitative changes in the processor computation model of the target network. In other cases, not even adaptations conditional on such redesign of processor programming model are as yet known. Finally, in some cases, it is proved that one network cannot be efficiently substituted for another.

Index-Shuffle Graphs. The index-shuffle graph (cf. [6]) is a degree-5 hypercube derivative, with logarithmic diameter and high bisection width—in these aspects it is similar to butterflies and shuffles. The results of [6] show that index-shuffle graphs can be employed efficiently in place of butterflies and shuffles, in the execution of *general computations.* Additionally, index-shuffle graphs are the only bounded-degree family currently known to be able to execute efficiently *on-line leveled hypercube algorithms.* The index-shuffle graph is significantly superior to the hypercube itself in the task of emulating the shuffle-exchange graph (which is performed without slowdown).

New Results. Whereas the index-shuffle graph is known to be efficient as a substitute for two leading *individual* hypercube derivative families, this paper shows that it is not necessarily in this role that the communication power of the index-shuffle graph is fully manifested. We show that the index-shuffle graph efficiently supports the direct-product structure of the hypercube itself. While the n-dimensional hypercube is the n-fold direct product of the 2-node path, we study k-fold direct products in the equal-sized n-dimensional index-shuffle graph, where $1 \leq k \leq n$. These products are of the form: $G = G_1 \times G_2 \times \cdots \times G_k$. Each factor in this product is (independently of the others) either a cycle or a complete binary tree or an X-tree. Hence, the product is characterized by a degree of structural heterogeneity, since the factors in general need not be isomorphic. They also need not be of equal size, provided the following size constraint is satisfied: $2^{\lceil \log_2 k \rceil} \cdot (\max_{1 \leq i \leq k} n_i) \leq n$. Any factor G_i, of size $|G_i| \leq 2^{n_i}$, is emulated with a slowdown of the form: $O(\log n_i + \log k) = O(\log \log N)$, which consists of two terms. The first term is doubly logarithmic in the size of the emulated factor graph, while the second term is logarithmic in the number of factors k. Thus, the slowdown component whose origin is in multidimensionality increases gracefully with the number of dimensions, while the slowdown component responsible for supporting

the factor structure remains invariant and small. At no additional cost, 2^ℓ multiple identical copies of the product are emulated, provided that their total size does not exceed that of the index-shuffle graph: $\ell + \sum_{i=1}^{k} n_i \leq n$. This ability to support a varying number of copies of varying sizes of several graph structures with no additional slowdown is found in the hypercube (cf. [12]), but not to a comparable extent in any of its bounded-degree derivatives.

Our emulation algorithms require no upgrade in the processor capabilities and no change in the processor programming model. The routing algorithm may run even in SIMD architectural mode where the source address of a message is the only input argument. The complexity of evaluating the routing decision at any individual step is as low as linear in the length of the binary representation of the source address.

Consequences of New Results. The index-shuffle graph efficiently substitutes any of the four standard families: butterflies, shuffles, meshes, and meshes of trees, in the execution of general computations; it efficiently substitutes the hypercube in the on-line leveled hypercube algorithms. In each of these five computation classes, the index-shuffle graph is shown to be asymptotically exponentially faster than any of the other four standard bounded-degree hypercube substitutes. (cf. [3, 4, 7, 9, 11, 13, 18, 20, 23].) The only candidate exception is Schwabe's [22] embedding of the mesh of trees into its like-sized shuffle-like graph with constant dilation. This embedding, however, assigns two guest nodes to a single host node. This modification in the formal setting may induce non-trivial changes in the meaning of the resulting emulation by forcing the programming model of host processors to change from a single-processor to a multiprocessor. This requires time-sharing, memory sharing, and I/O-interleaving. In a parallel architecture consisting of a large number of relatively simple processing elements, this modification may be prohibitively complex. Similarly, the formal model in [14], where algorithms are given that translate computations between shuffles and butterflies incurring only a constant slowdown, relies on data circulation through the network to an extent that prevents processors from performing on-line I/O and severely restricts the size of their local memories.

1.1. Formal Model and Preliminaries

Parallel Architectures. Parallel architectures consist of *identical processing elements*, each of which has some number of *interprocessor communication links* that connect it with other processing elements. All the links together form the *interconnection network*. The computation of the parallel architecture develops as a sequence of *pulses*; each pulse is either a *computation* step or a *communication* step. In a computation step each processing element computes in the context of its local memory. In a communication step a processing element may send a *message* to its neighbor(s).

Interconnection networks are represented as *graphs*, whose *nodes* stand for processing elements, and whose *edges* stand for bidirectional interprocessor communication links. The communication capabilities of various interconnection networks are compared by determining the efficiency with which one interconnection network *emulates* another. The theoretical emulation setting is defined via the formal notion of *embedding* one graph in another (cf. [18]).

Graph Embedding. An embedding of an undirected *guest* graph G in an undirected *host* graph H comprises two mappings: an injective *assignment* of the *nodes* of G to *nodes* of H, plus a *routing* that associates with each *edge* of G a *path* in H that connects the images of the edge's endpoints. The efficiency of an embedding is described in terms of its costs: **(1)** *dilation:* the length of the longest path in H that routes an edge of G; **(2)** *congestion:* the maximum number of edges of G routed over a single edge of H; **(3)** *expansion:* the ratio $(|H|/|G|)$.

Emulations. In an emulation of a guest graph G by a host graph H, each host processor in H emulates the guest processor which is its pre-image in G under the embedding assignment; this assignment is fixed throughout the emulation, thus enabling an automatic, step-by-step (program-independent) translation of every guest program into a functionally equivalent host program. The processors of guest architecture G and host architecture H have identical computational power. For every computation step of the guest architecture the host architecture executes one computation step, identical to the original guest step. For every *communication step* of the guest architecture the host architecture executes a *sequence* of one or more communication steps, which takes a message from an image in H of an embedded guest processor to the image in H of its embedded neighbor in G over the path in H that routes the embedded edge between the two processors of G.

An emulation associates a communication *schedule* with an embedding. Given a guest communication step, the embedding specifies where the paths in the host are that route the messages for that step, while the schedule specifies for each message the time steps at which it traverses individual links on its path or is held waiting at a node. The emulation *slowdown* is equal to the length in the number of steps of the longest host communication sequence required for emulating an arbitrary guest communication step. The emulation *queue size* is the largest number of messages held waiting at any node at any time step.

Given an embedding of a bounded-degree graph G into a bounded-degree graph H with dilation Δ and congestion K, the straightforward lower bound on the slowdown of any emulation based on this embedding is $\max(\Delta + K)$. Leighton, Maggs, and Rao [19] show that for every such embedding there exists an emulation schedule with slowdown $O(\Delta + K)$. In this paper we provide explicit constructions of our schedules with slowdown of that order and with constant queue size.

Notation and Conventions. Let $Z_n = \{0, 1, \ldots, n-1\}$. The lowercase Greek letters α and β denote variables with values in Z_2. For integer $k \geq 0$, Z_2^k denotes the set of all strings of length k over Z_2. $Z_2^0 = \{\lambda\}$ is the singleton set consisting of the *empty string* λ. For $x \in Z_2^k$, $|x| = k$ is the length of string x. Let $\alpha^k \in Z_2^k$ be the length-k string all of whose elements are equal to α. Given a string $x = \alpha_{k-1}\alpha_{k-2}\ldots\alpha_i\ldots\alpha_m\ldots\alpha_1\alpha_0 \in Z_2^k$, $x^{[m]} = \alpha_m$ is the bit at position m in x; $x^{[i,m]} = \alpha_i \cdots \alpha_m$ is the substring consisting of bits at positions m through i. For any string x, x^R is the reversal of x. Let $\overline{\beta} = 1 - \beta$ and $\overline{\beta x} = \overline{\beta}\,\overline{x}$.

For integers i and k such that $i < 2^k$, let $\mathcal{B}_k(i) \in Z_2^k$ denote the length-k binary representation of i, so that $i = \sum_{j=0}^{k-1} \beta_j 2^j$ just when $\mathcal{B}_k(i) = \beta_{k-1}\ldots\beta_1\beta_0$ and also $\mathcal{B}_k^{-1}(\beta_{k-1}\ldots\beta_1\beta_0) = i$.

For a graph $G = (N, E)$, the node-set N of G is denoted by $\mathcal{N}(G)$, while the edge-set E of G is denoted by $\mathcal{E}(G)$. The *size* of graph G, denoted by $|G|$, is the number of its nodes $|\mathcal{N}(G)|$.

Functions are composed from right to left, so that function fg applied to x yields $f(g(x))$, while the value of the composition $\prod_{j=0}^{m} f_j$ at point x equals the value of $\prod_{j=0}^{m-1} f_j$

at point $f_m(x)$. Let f^0 be the identity mapping, and let $f^n = f^{n-1}f$, for $n > 0$.

Standard Graph Families. The length-$(N-1)$ **path** P_N consists of a sequence of N nodes with an edge between every node and its successor: $\mathcal{N}(P_N) = Z_N$; $\mathcal{E}(P_N) = \{(i, i+1) \mid 0 \leq i < N-1\}$. Nodes 0 and $N-1$ are the **endpoints** of P_N. The **distance** between nodes u and $u+k$ in path P_N is k.

The length-N **cycle** L_N consists of a sequence of N nodes with an edge between every node and its successor $(\bmod N)$: $\mathcal{N}(L_N) = Z_N$;
$\mathcal{E}(L_N) = \{(i, (i+1 \bmod N)) \mid 0 \leq i \leq N-1\}$.

Given graphs G and G', their **direct product** $G \times G'$ has the node-set $\mathcal{N}(G \times G') = \mathcal{N}(G) \times \mathcal{N}(G')$ and contains an edge $((u,u'),(v,v')) \in \mathcal{E}(G \times G')$ iff either $u = v \wedge (u',v') \in \mathcal{E}(G')$ or $u' = v' \wedge (u,v) \in \mathcal{E}(G)$. Graphs G and G' are the **factors** of $G \times G'$.

The node-set of the height-h **complete binary tree** T_h comprises all binary strings of length not exceeding h: $\mathcal{N}(T_h) = \bigcup_{0 \leq k \leq h} Z_2^k$. The edge-set of T_h consists of two subsets: $\mathcal{E}(T_h) = E_L \cup E_R$, such that:
$E_L = \{(x, x0) \mid x \in Z_2^k, 0 \leq k < h\}$;
$E_R = \{(x, x1) \mid x \in Z_2^k, 0 \leq k < h\}$.
Edges in E_L connect every tree node x of length $|x| < h$ to its **left child** $x0$, while edges in E_R connect every such node to its **right child** $x1$. Node x is the **parent** of nodes $x0$ and $x1$. The empty string λ is the **root** of the tree; the 2^k strings of length k are its **level-k** nodes; the 2^h strings of length h are its **leaves.**

The height-h **X-tree** graph X_h is obtained from the complete binary tree T_h by augmenting the edge-set of T_h so that all nodes at each level $k, 0 < k \leq h$ are connected by **level-k** edges in lexicographic order into the path P_{2^k}.

The node-set of the dimension-n **hypercube** Q_n consists of all binary strings of length n: $\mathcal{N}(Q_n) = Z_2^n$. The edge-set of Q_n consists of n subsets:
$E_k = \{(\alpha_{n-1} \cdots \alpha_k \cdots \alpha_1\alpha_0 \ , \ \alpha_{n-1} \cdots \overline{\alpha_k} \cdots \alpha_1\alpha_0)\}$.
Dimension-k edges connect every node to the node that differs from it only in bit-position k.

1.2. Definition of Index-Shuffle Graphs

Recall that the shuffle permutation within a set of nodes indexed by n-bit strings is usually defined by means of a circular shift of index sequences. This definition is valid under the assumption that the entire set of 2^n indices is found in the graph. A shuffle permutation that operates on sets of arbitrary size is defined as follows:

Definition 1 *The n-element* **shuffle**, *for $n = 2k + \alpha$, is the permutation $\sigma_n : Z_n \to Z_n$ such that:*

$$\sigma_n(x) = \begin{cases} 2x & \text{if } x < k + \alpha \\ 2(x - k - \alpha) + 1 & \text{if } x \geq k + \alpha \end{cases}$$

This definition of the shuffle permutation is consistent with the standard one when n is a power of 2.

Definition 2 (Index-Shuffle Graphs) *The node-set of the dimension-n* **index-shuffle** *graph Ψ_n consists of all binary strings of length n: $\mathcal{N}(\Psi_n) = Z_2^n$. The edge-set of Ψ_n consists of three subsets: $\mathcal{E}(\Psi_n) = E_e \cup E_\sigma \cup E_\chi$.*

Exchange *edges E_e connect every node to the node whose index differs only in bit-position 0:*
$E_e = \{(\alpha_{n-1}\alpha_{n-2} \cdots \alpha_1\alpha_0 \ , \ \alpha_{n-1}\alpha_{n-2} \cdots \alpha_1\overline{\alpha_0})\}$.
For definiteness, arc $(\alpha_{n-1} \cdots \alpha_1 0 \ , \ \alpha_{n-1} \cdots \alpha_1 1)$ (which rewrites 0 to 1) is called the **exchange arc** *and labeled $\overset{\circ}{e}$, while arc $(\alpha_{n-1} \cdots \alpha_1 1 \ , \ \alpha_{n-1} \cdots \alpha_1 0)$ (which rewrites 1 to 0) is called the* **un-exchange arc**, *and labeled $\overset{\bullet}{e}$.*

Shuffle *edges E_σ connect every node with the node whose index is obtained by a left circular shift by one bit position:*
$E_\sigma = \{(\alpha_{n-1}\alpha_{n-2} \cdots \alpha_1\alpha_0 \ , \ \alpha_{n-2} \cdots \alpha_1\alpha_0\alpha_{n-1})\}$.
For definiteness, arc $(\alpha_{n-1}\alpha_{n-2} \cdots \alpha_1\alpha_0 \ , \ \alpha_{n-2} \cdots \alpha_1\alpha_0\alpha_{n-1})$ (which shifts the index to the left) is called the **shuffle arc** *and labeled $\overleftarrow{s}$, while arc $(\alpha_{n-2} \cdots \alpha_1\alpha_0\alpha_{n-1} \ , \ \alpha_{n-1}\alpha_{n-2} \cdots \alpha_1\alpha_0)$ (which shifts the index to the right) is called the* **un-shuffle arc** *and labeled $\overrightarrow{s}$.*

Index-Shuffle *edges E_χ connect every node with the node whose index is obtained by an n-element shuffle:*
$E_\chi = \{(\alpha_{n-1} \cdots \alpha_1\alpha_0 \ , \ \alpha_{\sigma_n^{-1}(n-1)} \cdots \alpha_{\sigma_n^{-1}(1)}\alpha_{\sigma_n^{-1}(0)})\}$.
For definiteness, arc $(\alpha_{n-1} \cdots \alpha_1\alpha_0 \ , \ \alpha_{\sigma_n^{-1}(n-1)} \cdots \alpha_{\sigma_n^{-1}(1)}\alpha_{\sigma_n^{-1}(0)})$ (which shuffles the index) is called the **index-shuffle arc** *and labeled $\overleftarrow{\imath}$, while arc $(\alpha_{n-1} \cdots \alpha_1\alpha_0 \ , \ \alpha_{\sigma_n(n-1)} \cdots \alpha_{\sigma_n(1)}\alpha_{\sigma_n(0)})$ (which un-shuffles the index) is called the* **index-un-shuffle arc** *and labeled $\overrightarrow{\imath}$.*

2. New Emulations

The assignments of all embeddings constructed in this paper enable constant-slowdown emulations if the hypercube is employed as the host in place of the index-shuffle graph. This fact is consistent with our view of the index-shuffle graph as primarily a substitute for the hypercube rather than for any (set) of its derivatives. Furthermore, an N-node hypercube is embedded into its equal-sized index-shuffle graph with dilation $O(\log\log N)$ [6]. Composing the two assignments guarantees dilation of $O(\log\log N)$ for embeddings of these guests into the index-shuffle graph. However, currently, the congestion of the embedding of the hypercube into the index-shuffle graph is not known. Its order is certainly $\Omega(\log N)$, since that is the factor by which the number of guest edges exceeds the number of host edges. Hence, our emulation algorithms build on constant-dilation embeddings of guest graphs into the hypercube, and provide these embeddings with scheduling schemes for the index-shuffle graph that avoid congestion.

Routing Vocabulary. In the hypercube, two nodes that are at distance d differ in exactly d bits. In the embeddings presented in this paper, endpoints of embedded edges also differ in exactly 2 bits. However, in the index-shuffle graph such nodes are at a distance greater than 2. The routing path that connects such endpoints is said to **flip** the required bits. Generally, **flipping the kth bit** of a node index $x = \alpha_{n-1}, \ldots \alpha_k, \ldots, \alpha_1, \alpha_0$ means traversing a routing path from node x to node $x' = \alpha_{n-1}, \ldots \overline{\alpha_k}, \ldots, \alpha_1, \alpha_0$, which differs from x at bit position k; α_k is called the **flip bit**.

It is convenient to imagine a flipping route as a sequence of **rewriting** operations that a message performs on its node index to transform it from the source index to the destination index. Envision the message as if it operates on individual bits in the index, rather then on the entire index. The n bits

of an index are distinguishable, independent, and named by the position they occupy in the index of the source node at the beginning of the flipping route. The route **toggles** the flip bit while leaving the others **intact**.

Each of the four arcs: $\{\overleftarrow{s}, \overrightarrow{s}, \overleftarrow{\imath}, \overrightarrow{\imath}\}$ permutes the index bits. The flip bit can be toggled only if it is at position 0. Hence, the strategy of the message is to use arcs $\{\overleftarrow{s}, \overrightarrow{s}, \overleftarrow{\imath}, \overrightarrow{\imath}\}$ to permute the bits so as to bring the flip bit to position 0 as quickly as possible. After this first, **downhill** routing stage, when the flip bit is at position 0, one of the arcs $\{\overset{\circ}{e}, \overset{\bullet}{e}\}$ toggles the flip bit. Finally, all bits are restored through the **uphill** routing stage, which permutes them so as to reverse the trajectory followed in the downhill stage. The pairs $(\overleftarrow{s}, \overrightarrow{s})$ and $(\overleftarrow{\imath}, \overrightarrow{\imath})$ are mutually inverse if interpreted as permutations on the sequence of index bits; the **reversal** of a route is thus obtained by reversing the sequence of the arcs in the route and then substituting for each arc its inverse. Thus, the destination index differs from the source in that the flip bit is toggled, while the others are intact. We write τ for the **current position** of the flip bit during the flipping route, and φ for the **flip bit itself**. The **current index** is denoted by w.

General Routing Rules. The just described simple strategy (downhill; toggle; uphill) is not feasible because of its unknown and possibly high congestion. To avoid collisions, our schedules demand that messages rewrite their indices in more sophisticated ways. Most importantly, messages toggle bits other than the flip bit. Since two successive toggles amount to remaining in the intact state, every bit is toggled an even number of times, except for the flip bit φ, which is toggled an odd number of times.

Add to the routing vocabulary the **identity** arc, $\widehat{n}$, by convention applied to those indices whose messages do not move at a given step; $\widehat{n}$ is evidently its own inverse. Arcs $\{\overleftarrow{s}, \overrightarrow{s}, \overleftarrow{\imath}, \overrightarrow{\imath}, \widehat{n}\}$ are total permutations of the node set of the index-shuffle graph, while $\overset{\circ}{e}$ and $\overset{\bullet}{e}$ are partial. Our schedules are designed to be **legal** in that they never apply any of the two arcs $\{\overset{\circ}{e}, \overset{\bullet}{e}\}$ to a node where that arc is undefined.

If messages occupy distinct nodes at some time point in a schedule, and if *all* messages then traverse the *same* arc, then no two messages collide at this step. In contrast, if various arcs are employed at the same step, collisions may occur. Our congestion-avoiding schemes rely on specific properties of the node assignments that link the position of the flip bit τ with the value of some characteristic subsequences of the index w. This dependence serves to prevent two messages from rewriting two distinct nodes by two different arcs into the same node.

Routing Patterns. Our schedules are composed as short sequences of recurring **pattern** schedule segments. Each pattern is characterized by its length in the number of steps (always $O(\log\log N)$ in N-node graph), by its queue-size (constant), and by its input-output specification (constraints on τ and w, before and after the rewriting encapsulated in the pattern.) If messages occupy distinct nodes initially, then each pattern guarantees that they also end at distinct nodes.

Routing Syntax. Hereafter, all schedules refer to the n-dimensional index-shuffle graph Ψ_n as the emulation host; the number of nodes of Ψ_n is $N = 2^n$. Let $\Lambda = \lfloor \log_2 n \rfloor + 1$; assume that n is sufficiently large so that $n > \Lambda$. The flip bit is initially at position τ_0; the initial index is w_0. The final position of the flip bit is τ_f; the final index is w_f. Recall that functions are composed from right to left; for example, sequence $\langle \overleftarrow{s}\, \overset{\circ}{e} \rangle$ rewrites index 1100 into 1011.

The pre-conditions of a pattern specify the set of indices which the pattern rewrites. All messages occupying such indices perform in parallel the rewriting sequence defined by the pattern. The operation is synchronous—traversing one arc takes precisely one time step, always and to all messages. Only the communication steps are counted—the time consumed by local computation, required to support the scheduling algorithm, is neglected. Such computation is assumed to be performed during the schedule step which it immediately precedes. The complexity of the local computation associated with any communication step is at worst linear in the number of bits in the index.

A specification of an arc sequence enclosed in angle brackets $\langle \boldsymbol{p} \rangle$ stands for a schedule **segment** that is run until all the initially active messages complete their rewriting, specified by the arc sequence $\boldsymbol{p}$. Note that $\boldsymbol{p}$ depends on the message that applies it—$\boldsymbol{p}$ is not a simple arc sequence, but an encoding of an arc sequence, which may include conditions and iterations, with index w as a parameter. Rewriting sequences of *all* messages in the segment $\langle \boldsymbol{p} \rangle$ have the same length—those that are actually shorter are "padded" at the end with identity arcs $\widehat{n}$. Hence, $\langle \boldsymbol{p}' \rangle \langle \boldsymbol{p} \rangle \neq \langle \boldsymbol{p}' \boldsymbol{p} \rangle$. In the 2-segment sequence on the left-hand side all messages apply the rewriting sequence $\boldsymbol{p}$, starting at the same time step; when they all complete $\boldsymbol{p}$ then they all apply $\boldsymbol{p}'$, again starting simultaneously. In the segment on the right-hand side all messages start simultaneously and apply the entire sequence $\boldsymbol{p}'\boldsymbol{p}$.

The **length** of a segment $\langle \boldsymbol{p} \rangle$, written $\|\langle \boldsymbol{p} \rangle\|$, is the number of steps required for the congestion-free completion of $\langle \boldsymbol{p} \rangle$. The **queue-size** of the segment $\langle \boldsymbol{p} \rangle$ is denoted by $\Upsilon\langle \boldsymbol{p} \rangle$.

2.1. Routing Library

The library consists of five routing patterns.

Pattern (1): SEQUENCING. If two segments $\langle \boldsymbol{p} \rangle$ and $\langle \boldsymbol{p}' \rangle$ are congestion-free, then so is their **composition** $\langle \boldsymbol{p}' \rangle \langle \boldsymbol{p} \rangle$.
$\|\langle \boldsymbol{p}' \rangle \langle \boldsymbol{p} \rangle\| = \|\langle \boldsymbol{p} \rangle\| + \|\langle \boldsymbol{p}' \rangle\|$;
$\Upsilon\langle \boldsymbol{p}' \rangle \langle \boldsymbol{p} \rangle = \max(\Upsilon\langle \boldsymbol{p} \rangle, \Upsilon\langle \boldsymbol{p}' \rangle)$.

In contrast, the **concatenation** $\langle \boldsymbol{p}'\boldsymbol{p} \rangle$ is not necessarily congestion-free even if both $\langle \boldsymbol{p} \rangle$ and $\langle \boldsymbol{p}' \rangle$ are—some messages may commence their $\boldsymbol{p}'$ stage while others are still in $\boldsymbol{p}$, thus causing collisions. Assume that no two messages applying $\boldsymbol{p}'$ in $\langle \boldsymbol{p}'\boldsymbol{p} \rangle$ collide at any time step. Observe that this assumption does not follow from the fact that $\langle \boldsymbol{p}' \rangle$ is congestion-free, since in $\langle \boldsymbol{p}' \rangle$ all messages start the rewriting sequence $\boldsymbol{p}'$ at the same time step, which is not the case for $\boldsymbol{p}'$ in $\langle \boldsymbol{p}'\boldsymbol{p} \rangle$. In this case, at the cost of maintaining both queues, the concatenation $\langle \boldsymbol{p}'\boldsymbol{p} \rangle$ is run congestion-free by interleaving $\boldsymbol{p}$ with $\boldsymbol{p}'$. At the first and all subsequent odd-numbered steps, only those messages applying $\boldsymbol{p}$ are active. At the second and all subsequent even-numbered steps, only those messages applying $\boldsymbol{p}'$ are active. The generalization to concatenations of k schedules is straightforward:
$\|\langle \boldsymbol{p}_k \cdots \boldsymbol{p}_1 \rangle\| = k \cdot \sum_{i=1}^{k} \|\langle \boldsymbol{p}_i \rangle\|$;
$\Upsilon\langle \boldsymbol{p}_k \cdots \boldsymbol{p}_1 \rangle = \sum_{i=1}^{k} \Upsilon\langle \boldsymbol{p}_i \rangle$.
If no two messages applying $\boldsymbol{p}'$ in $\langle \boldsymbol{p}'\boldsymbol{p} \rangle$ collide at any time

step, then $\langle \boldsymbol{p}'\boldsymbol{p}\rangle$ remains congestion-free even if $\langle \boldsymbol{p}\rangle$ allows several messages to have the same final destination, but is otherwise congestion-free.

Pattern (2): REVERSAL. The **reversal** $\overleftarrow{\boldsymbol{p}}$ of a schedule sequence $\boldsymbol{p}$ is obtained by reversing $\boldsymbol{p}$ as a sequence of arc labels and then substituting every occurrence of each arc in $\boldsymbol{p}^R$ by the inverse of that arc.

Every legal schedule sequence may be followed by its reversal, which, as expected, undoes the rewriting performed by the original schedule.
$\langle \overleftarrow{\boldsymbol{p}}\,\boldsymbol{p}\rangle(w_0) = \langle \overleftarrow{\boldsymbol{p}}\rangle\langle \boldsymbol{p}\rangle(w_0) = w_0$.

Pattern (3): PRIMARY FLIP. The pre-condition for the **primary flip** pattern $\boldsymbol{f}$ is:
$w_0 = x00^k,\ \tau_0 = k,$ where $k \geq 0$. Hence, $\boldsymbol{f}$ flips a $\varphi = 0$ to 1 if all bits to the right of φ are equal to 0. The reversal $\overleftarrow{\boldsymbol{f}}$ of $\boldsymbol{f}$ flips a $\varphi = 1$ to 0 if all bits to the right of φ are equal to 0.

The downhill stage $\boldsymbol{f}_d$ of $\boldsymbol{f}$ effects: $\tau \leq 1$; $\boldsymbol{f}_d$ is as follows:

$\tau \leftarrow \tau_0$
while $(\tau \geq 2)$ **do**
  if (τ is odd) **do** $\langle \overset{\circ}{\boldsymbol{e}}\overrightarrow{\boldsymbol{s}}\rangle$ **else do** $\langle \widehat{\boldsymbol{n}\,\boldsymbol{n}}\rangle$ **endif**
  $\langle \overrightarrow{\boldsymbol{s}}\,\overrightarrow{\boldsymbol{i}}\rangle$
  $\tau \leftarrow \lfloor \tau/2 \rfloor - 1$
endwhile

The toggle stage $\boldsymbol{f}_t$ of $\boldsymbol{f}$, which toggles φ to 1, while leaving τ intact, given $\tau \leq 1$, is as follows:

if $(\tau = 0)$ **do** $\langle \overset{\circ}{\boldsymbol{e}}\rangle$ **endif**
if $(\tau = 1)$ **do** $\langle \overleftarrow{\boldsymbol{s}}\,\overset{\circ}{\boldsymbol{e}}\,\overrightarrow{\boldsymbol{s}}\rangle$ **endif**

The uphill stage $\boldsymbol{f}_u$ of $\boldsymbol{f}$, which takes $\varphi = 1$ to position $\tau_f = \tau_0 = k$ and otherwise restores w, is the reversal of the downhill stage: $\boldsymbol{f}_u = \overleftarrow{\boldsymbol{f}}_d$. The primary flip is the concatenation: $\boldsymbol{f} = \langle \boldsymbol{f}_u\boldsymbol{f}_t\boldsymbol{f}_d\rangle$.

By a straightforward arithmetic argument it is verified that the downhill stage $\boldsymbol{f}_d$ (and the entire $\boldsymbol{f}$) correctly computes τ within the **while**-loop. Each pass through this loop modifies τ by a factor of at least 2, whence $\|\boldsymbol{f}\| \approx \log \tau_0 \approx \Lambda$. The toggle stage $\boldsymbol{f}_t$ is congestion-free, with queue size $\Upsilon\boldsymbol{f}_t = 2$, since two distinct messages, one with $\tau = 0$ and the other with $\tau = 1$, may toggle at the same index in succession.

We proceed to prove that the downhill stage is congestion-free, with queue-size $\Upsilon\boldsymbol{f}_d = 2$. The loop invariant in $\boldsymbol{f}_d$ is: $w^{[\tau,0]} = 0$, and no two messages operate on the same index. The claim is true before the first pass—observe that $\varphi = 0 = w^{[0]}$ holds even if $\tau_0 = 0$. Assume that the claim holds after some number of passes through the loop. A message that sees an odd τ may temporarily visit an already occupied node, by applying $\overrightarrow{\boldsymbol{s}}$. The node-disjointness is restored at the very next step, since $\overset{\circ}{\boldsymbol{e}}$ rewrites into a node with $w^{[0]} = 1$, which is unoccupied by the inductive hypothesis. These messages do not collide among themselves, because $\langle \overset{\circ}{\boldsymbol{e}}\overrightarrow{\boldsymbol{s}}\rangle$ is a permutation. The remainder of the loop, $\langle \overrightarrow{\boldsymbol{s}}\,\overrightarrow{\boldsymbol{i}}\rangle$, as a permutation, preserves the node-disjointness and also unshuffles 1 from $w^{[0]}$ to $w^{[n-1]}$, thus restoring the invariant.

The uphill sequence $\boldsymbol{f}_u$ in $\boldsymbol{f} = \langle \boldsymbol{f}_u\boldsymbol{f}_t\boldsymbol{f}_d\rangle$ does not inherit from its reversal $\boldsymbol{f}_d$ the property of being congestion-free, since the messages do not start $\boldsymbol{f}_u$ in $\boldsymbol{f}$ simultaneously. Indeed, executing the composition $\langle \boldsymbol{f}_u\rangle\langle \boldsymbol{f}_t\rangle\langle \boldsymbol{f}_d\rangle$ would allow for accumulation of messages at some nodes at the toggle stage. Therefore, a separate proof is required that $\boldsymbol{f}_u$ is congestion-free in $\boldsymbol{f}$. Precisely, we prove that two messages that start $\boldsymbol{f}_u$ at different time steps do not collide—those messages that start $\boldsymbol{f}_u$ simultaneously do not collide because $\boldsymbol{f}_u = \overleftarrow{\boldsymbol{f}}_d$ and $\boldsymbol{f}_d$ is congestion-free.

Assume that a message that has completed m passes through the **while**-loop collides at node w with a message that has completed $m' > m$ passes through the loop. The flip bit of the fist message is at position τ, such that $2^{m+1} - 2 \leq \tau \leq 2^{m+2} - 3$, which is straightforwardly verified by induction on m, given that each pass through the uphill loop, by construction, modifies τ to $2(\tau + 1) + \alpha$, for some $0 \leq \alpha \leq 1$. Likewise, the flip bit of the second message is at position τ', such that $2^{m'+1} - 2 \leq \tau' \leq 2^{m'+2} - 3$. Since $m' \geq m+1$, we have: $\tau \leq 2^{m+2} - 3 \leq 2^{m'+1} - 3 < 2^{m'+1} - 2 \leq \tau'$. Recall that all the bits to right of the flip bit are equal to 0, while the flip bit is 1. According to the first message, $w^{[\tau]} = 1$, because it is the flip bit. According to the second message, $w^{[\tau]} = 0$, since $\tau < \tau'$ is strictly lower than the position of the flip bit τ', whence the contradiction.

Pattern (4): FOLDING. Let $j < k$, let K = $2^{\lceil \log_2 k \rceil}$ and let n' be the largest number such that $K \cdot n' \leq n$. There are no pre-conditions for the application to any index of $\boldsymbol{o}_k^j$, the k-**layered folding** pattern with **face** j **up**. The effect of $\langle \boldsymbol{o}_k^j\rangle(w_0)$ is that n' bits which occupy positions j mod K of w_0 appear compacted as the length-n' prefix of w_f: $w_f^{[n'-1,0]} = w_0^{[j+(n'-1)K]} \cdots w_0^{[j+K]} w_0^{[j]}$. The pattern $\boldsymbol{o}_k^j$ is as follows:

$\tau \leftarrow j\ ;\ \rho \leftarrow K$
while $(\rho > 1)$ **do**
  if (τ is odd) **do** $\langle \overrightarrow{\boldsymbol{s}}\rangle$ **endif**
  $\langle \overrightarrow{\boldsymbol{i}}\rangle$
  $\tau \leftarrow \lfloor \tau/2 \rfloor\ ;\ \rho \leftarrow \rho/2$
endwhile

Pattern $\boldsymbol{o}_k^j$ is a permutation and thereby congestion-free with queue-size $\Upsilon\boldsymbol{o}_k^j = 1$. The length of the folding pattern is: $\|\langle \boldsymbol{o}_k^j\rangle\| \approx \log k \approx \Lambda$. To verify the post-condition on w, fix a bit position $\tau_0 = mK + j$. We show that $\tau_f = m$. The property of the **while**-loop is that after ℓ passes through the loop $\tau = (m \cdot K/2^\ell) + \lfloor j/2^\ell \rfloor$. The loop is executed exactly $\log_2 K$ times, whence the claim.

The k-**layered unfolding** pattern with **face** j **up** is the reversal $\overleftarrow{\boldsymbol{o}_k^j}$ of the corresponding folding pattern.

Pattern (5): ADVANCED FLIPS. All advanced flip patterns inherit their (asymptotic) length, constant queue-size, and property of being congestion-free from the primary flip, from which they are derived by enclosing $\boldsymbol{f}$ within matching pairs of folding and unfolding stages. The purpose of these is to "fold away" the bits that violate the pre-condition of the primary flip, thereby turning "face up" the required all-0 prefix to the flip bit φ.

The pre-condition for the **shadowed flip** pattern $\boldsymbol{s}$ is: $w_0 = x010^{k-1},\ \tau_0 = k+1,$ where $k > 0$. Call the bit immediately to the right of the flip bit the shadow. The shadowed flip $\boldsymbol{s}$ flips a $\varphi = 0$ to 1 if its shadow is equal to 1, while all the other bits to the right of the shadow are equal to 0. The reversal $\overleftarrow{\boldsymbol{s}}$ of $\boldsymbol{s}$ flips a $\varphi = 1$ to 0 if all bits to the right of its shadow 1 are equal to 0.

Pattern $\boldsymbol{s}$ is a composition of two stages: $\boldsymbol{s} = \langle \boldsymbol{s}_1 \rangle \langle \boldsymbol{s}_0 \rangle$. For $i = 0, 1$, at stage i active are those messages with $\tau_0 = i \bmod 2$; $\boldsymbol{s}_i = \langle \boldsymbol{o}_2^i \rangle \langle \boldsymbol{f} \rangle \langle \overleftarrow{\boldsymbol{o}_2^i} \rangle$. To verify the correctness of $\boldsymbol{s}_i$, observe that the initial folding transforms $w_0 = x\varphi 10^{k-1}$ into $y\varphi 0^{\lfloor (k-1)/2 \rfloor}$, (where $\varphi = 0$) to which the primary flip pattern applies.

The composition of a primary flip and a shadowed flip enables the **double flip**, which rewrites $w_0 = x\varphi_1\varphi_2 0^k$ into $w_f = x\overline{\varphi_1\varphi_2}0^k$. Hence, the double flip flips a pair of adjacent bits that follow a prefix consisting entirely of 0-valued bits.

Finally, a **repeated flip** rewrites $w_0 = x\varphi_1 10^{k_1}\varphi_2 10^{k_2}$ into $w_0 = x\overline{\varphi}10^k\overline{\varphi'}10^{k'}$, where $k, k' \geq 0$. Say that φ is at position τ_0, while φ' is at position τ'. The repeated flip is routed in two successive stages. At the first stage active are those messages for which $\tau_0 \neq (\tau_0' \bmod 4) \vee \varphi' \neq 1$ and $\tau_0 + 1 \neq \tau_0' \bmod 4$. They apply 4-layered folding with face $(\tau_0 \bmod 4)$ up to enable the primary flip of φ, followed by the corresponding unfolding. At the second stage active are those messages for which $\tau_0 = (\tau_0' \bmod 4) \wedge \varphi' = 1$ or $\tau_0 + 1 = \tau_0' \bmod 4$. These messages start out by applying a primary flip or a shadow flip, as required, to turn the 1 temporarily to 0 at the offending bit position. Next a segment identical to the one routed at the first stage is applied to flip φ. Finally, the temporarily annulled 1 is restored. Both stages are completed by applying the shadow flip to φ'.

2.2. Emulations Compiled

The routing library enables congestion-free flipping of one or more bits in specific index configurations, with constant queue-size and with a slowdown of the order of $\log n$ in the n-dimensional N-node index-shuffle graph $\boldsymbol{\Psi}_n$. Such configurations are all characterized by particular values of the index prefix that precedes the flip bit. The folding pattern enables several such index configurations to be interleaved in an index and still simultaneously eligible for rewriting, with an additional cost which is logarithmic in the number of these factors. Given an emulated product graph: $G = G_1 \times G_2 \times \cdots \times G_k$, it is thus sufficient to specify an emulation of any factor G_i by the index-shuffle graph $\boldsymbol{\Psi}_n$, as if G_i is emulated in isolation. If each of these emulations has slowdown of the order $O(\log n) = O(\log\log N)$, then the folding pattern guarantees that the entire product is emulated with slowdown: $O(\log k + \log n) = O(\log\log N)$, subject to the size constraints:[1] $|G_i| \leq 2^{n_i},$ for $i = 1, \ldots k$, and $2^{\lceil \log_2 k \rceil} \cdot (\max_{1 \leq i \leq k} n_i) \leq n$. Moreover, say that ℓ index bits are left unoccupied by the embedding assignment of the factors. These ℓ bits together form the encoding of 2^ℓ different copies of the entire product G, which is emulated by $\boldsymbol{\Psi}_n$ at no additional cost, subject to the size constraint: $\ell + \sum_{i=1}^k n_i \leq n$. A product of graphs is emulated by index-shuffle graph with the claimed cost whenever each factor of the product has an embedding into $\boldsymbol{\Psi}_n$ such that each endpoint of every embedded edge is rewritten into the other endpoint by one of the available flip patterns.

CYCLES. The nodes of a length-2^m cycle are assigned to nodes of the dimension-n N-node index-shuffle graph $\boldsymbol{\Psi}_n$, where $m \leq n$, on the basis of the dimension-m binary reflected Gray code (cf. [8].)

Dimension-m binary reflected Gray code $\Gamma^{(m)}$ is the sequence comprising 2^m length-m binary strings, defined recursively as follows.

$$\begin{array}{ll} \Gamma_0^{(0)} = 0 & \Gamma_i^{(m+1)} = \begin{cases} 0\Gamma_i^{(m)} & \text{if } i < 2^m \\ 1\Gamma_{2^{m+1}-1-i}^{(m)} & \text{if } i \geq 2^m \end{cases} \\ \Gamma_1^{(0)} = 1 & \end{array}$$

Dimension-m binary reflected Gray-code cycle $\mathcal{L}_m$ is a cycle whose nodes are elements of $\Gamma^{(m)}$, with an edge between every node and its successor in $\Gamma^{(m)}$, plus an edge between its first node $\Gamma_0^{(m)} = 0^m$, and its last node is $\Gamma_{2^m-1}^{(m)} = 10^{m-1}$.

Edge Structure and Routing Patterns. By induction, one verifies that every edge in $\mathcal{L}_m$ is of the form $(x\varphi\alpha 0^{i-1},\ x\overline{\varphi}\alpha 0^{i-1})$, where $x \in Z_2^{m-1-i}$. Hence, all edges in the 2^m-length cycle $\mathcal{L}_m$ are routed by the shadowed flip pattern in the index-shuffle graph $\boldsymbol{\Psi}_n$ with constant queue-size and slowdown $O(\Lambda) = O(\log\log N)$.

TREES. Let T_h be a height-h complete binary tree and let $\boldsymbol{\Psi}_n$ be a dimension-n index-shuffle graph such that $n \geq h + 1$. The assignment maps tree node $x \in Z_2^k$, where $0 \leq k \leq h$, into index-shuffle node $yx10^{h-k} \in Z_2^n$, where $y \in Z_2^{n-h-1}$ is an arbitrary string, identical for all nodes.

Edge Structure and Routing Patterns. The embedded endpoints of a **right-child** edge of T_h are of the form: $(yx10^{h-k},\ yx110^{h-k-1})$ In the index-shuffle graph $\boldsymbol{\Psi}_n$, the right-child edges rewrite $z100^m$ into $z110^m$, by the primary flip pattern, with constant queue-size and slowdown $O(\Lambda) = O(\log\log N)$.

The embedded endpoints of a **left-child** edge of T_h are of the form: $(yx10^{h-k},\ yx010^{h-k-1})$ In the index-shuffle graph $\boldsymbol{\Psi}_n$, the left-child edges rewrite $z100^m$ into $z010^m$, by the double flip pattern, with constant queue-size and slowdown $O(\Lambda) = O(\log\log N)$.

X-TREES. The embedding of the height-h X-tree X_h into the dimension-$(h+1)$ Gray code cycle $\mathcal{L}_{h+1}$ is defined recursively as follows. For $h = 1$, the root λ of X_1 is

[1] For the sake of simplicity, these constraints are stated unnecessarily strictly. One "layer" of index bits in a k-layered folding can be further "layered", to allow for a more efficient "packing" of factors.

embedded into node $01 \in \mathcal{N}(\mathcal{L}_1)$; its left child 0 is assigned to 00, while its right child 1 is assigned to 11; node 10 is left unoccupied.

Recursively, assume there is an assignment A_h that maps nodes of X_h to nodes of $\mathcal{L}_{h+1}$ so that the root λ is assigned to node 010^{h-1}, while node 10^h is left unoccupied. To construct the assignment $A_{h+1} : \mathcal{N}(X_{h+1}) \to \mathcal{N}(\mathcal{L}_{h+2})$, take two copies of X_h. Append 0 to the left of each node name in the first copy to generate the left subtree of X_{h+1}, and append 1 to the left of each node name in the second copy to generate the right subtree of X_{h+1}. For each node $0x$ in copy 0, let $A_{h+1}(0x) = 0A_h(x)$. For each node $1x$ in copy 1, let $A_h(x)$ appear as the ℓth node $\Gamma_\ell^{(h+1)}$ in the cycle $\mathcal{L}_{h+1}$; A_{h+1} assigns $1x$ to $\Gamma_{2^{h+1}+\ell}^{(h+2)}$. Finally, $A_{h+1}(\lambda) = 010^h$, while node 10^{h+1} remains unoccupied.

Edge Structure and Routing Patterns. By induction on h, one verifies that the embedded edges of X_h are as follows.

The embedded endpoints of a **left-child** edge of X_h are of the form: $(x\varphi\varphi'0^k, x\overline{\varphi}\overline{\varphi'}0^k)$ In the index-shuffle graph Ψ_n, the left-child edges are therefore routed by the double flip pattern, with constant queue-size and slowdown $O(\Lambda) = O(\log\log N)$.

The embedded endpoints of a **level** edge of X_h are of the form: $(x\varphi\alpha 0^k\varphi'\alpha' 0^{k'}, x\overline{\varphi}\alpha 0^k\overline{\varphi'}\alpha' 0^{k'})$ where $k, k' \geq 0$ and α or α' may be missing. In the index-shuffle graph Ψ_n, the level edges are therefore routed by the repeated flip pattern, with constant queue-size and slowdown $O(\Lambda) = O(\log\log N)$.

The **right-child** edges are routed in the index-shuffle graph Ψ_n by composing the routing segments of the corresponding left-child and level edges, with constant queue-size and slowdown $O(\Lambda) = O(\log\log N)$.

3. Conclusion

The relevance of our results is best assessed in the light of the fact that *index-shuffle graphs currently do not have a congestion-control theorem.* Given a graph H and a positive integer function $f^{[c]}$, graph H has a *congestion control theorem* with *slowdown* $f^{[c]}$ if every dilation-δ embedding of any degree-1 graph G' into H can be rerouted so that the resulting emulation of G' by H has constant queue-size and slowdown $f^{[c]}(\delta)$.

Currently, as shown by Bhatt et al. [7], butterflies, like meshes, have a congestion control theorem with linear slowdown: $f^{[c]}(\delta) = O(\delta)$; shuffle-like graphs have a congestion control theorem with slowdown $f^{[c]}(\delta) = O(\delta^2)$. It is currently unknown (cf. [18]) whether the hypercube has a congestion control theorem such that $f^{[c]}(3) \leq \Delta$ for any constant Δ that does not depend on the size of the host hypercube.

In the absence of a *general* congestion control theorem (with linear slowdown) for index-shuffle graphs, the dilation-$O(\log\log N)$ embeddings of our guest graphs into the index-shuffle graph cannot be straightforwardly transformed into emulations with the slowdown of the same order. The emulations presented here are enabled by a sophisticated routing scheme that employs *specific* properties of the guest graphs in order to avoid congestion—an approach analogous to that pursued in [20].

Acknowledgment. This research was supported in part by a grant from the City University of New York PSC-CUNY Research Award Program. Azriel Genack and an anonymous referee contributed many helpful comments on the presentation.

References

[1] A. Aggarwal. A comparative study of x-tree, pyramid, and related machines. In *25th IEEE FOCS*, pages 89–99, 1984.
[2] A. A.M. Despain and D. Patterson. X-tree – a tree structured multiprocessor architecture. In *5th Intl. Symp. on Computer Architecture*, pages 144–151, 1978.
[3] F. Annexstein, M. Baumslag, and A. L. Rosenberg. Group action graphs and parallel architectures. *SIAM J. Comput.*, 19:544–569, 1990.
[4] M. Baumslag, M. C. Heydemann, J. Opatrny, and D. Sotteau. Embeddings of shuffle-like graphs in hypercubes. Technical report, Univ. Paris-Sud, 1990.
[5] M. Baumslag and B. Obrenić. Index-shuffle graphs. In *8th IEEE Symposium on Parallel and Distributed Processing*, pages 160–168, 1996.
[6] M. Baumslag and B. Obrenić. Index-shuffle graphs. *Intl. J. of Foundations of Computer Science*, 1997. Special Isssue on Interconnection Networks, to appear (see also [5].).
[7] S. N. Bhatt, F. R. K. Chung, J.-W. Hong, F. T. Leighton, B. Obrenić, A. L. Rosenberg, and E. J. Schwabe. Optimal emulations by butterfly-like networks. *J. ACM*, 43:293–330, 1996.
[8] R. Chamberlain. Gray codes, fast fourier transforms and hypercubes. *Parallel Computing*, 6:225–233, 1988.
[9] M. Chan. Dilation-2 embedding of grids into hypercubes. In *Intl. Conf. on Parallel Processing*, pages 295–298, 1988.
[10] de Bruijn N. G. A combinatorial problem. *Proc. Koninklijke Nederlandsche Akademie van Wetenschappen (A)*, 49, Part 2:758–764, 1946.
[11] K. Efe. Embedding mesh of trees in the hypercube. *J. Parallel Distr. Comput.*, 11:222–230, 1991.
[12] D. S. Greenberg and S. Bhatt. Routing multiple paths in hypercubes. *Math. Syst. Th.*, 24:295–321, 1991.
[13] D. S. Greenberg, L. S. Heath, and A. L. Rosenberg. Optimal embeddings of butterfly-like graphs in the hypercube. *Math. Syst. Th.*, 23:61–77, 1990.
[14] R. Koch, F. T. Leighton, b. Maggs, S. Rao, A. L. Rosenberg, and E. J. Schwabe. Work-preserving emulations of fixed-connection networks. In *21st ACM STOC*, pages 227–240, 1990.
[15] C. P. Kruskal and M. Snir. A unified theory of interconnection network structure. *Theoretical Comput. Sci.*, 48:75–94, 1986.
[16] F. Leighton. *Complexity Issues in VLSI: Optimal Layouts for the Shuffle-Exchange Graph and Other Networks*. MIT Press, Cambridge, Mass, 1983.
[17] F. Leighton. New lower bound techniques for vlsi. *Math. Syst. Th.*, 17:47–70, 1984.
[18] F. T. Leighton. *Introduction to Parallel Algorithms and Architectures: Arrays, Trees, Hypercubes*. Morgan Kaufmann, San Mateo, Calif, 1992.
[19] F. T. Leighton, B. M. Maggs, and S. B. Rao. Packet routing and job-shop scheduling in O(congestion + dilation) steps. *Combinatorica*, 14:167–280, 1994.
[20] B. Obrenić. An approach to emulating separable graphs. *Math. Syst. Th.*, 26:41–63, 1993.
[21] F. P. Preparata and J. E. Vuillemin. The cube-connected cycles: a versatile network for parallel computation. *Commun. ACM*, 24:300–309, 1981.
[22] E. Schwabe. Embedding meshes of trees into deBruijn graphs. *Inf. Process. Lett.*, 43 (5):237–240, 1992.
[23] E. J. Schwabe. *Efficient Embeddings and Simulations for Hypercubic Networks*. PhD thesis, MIT, 1991.
[24] J. T. Schwartz. Ultracomputers. *ACM Trans. Prog. Lang. Syst.*, 2:484–521, 1980.
[25] H. Stone. Parallel processing with the perfect shuffle. *IEEE Trans. Comput.*, C-20:153–161, 1971.