PRODUCT-SHUFFLE NETWORKS:
Toward Reconciling Shuffles and Butterflies

Arnold L. Rosenberg

Computer and Information Science Department
University of Massachusetts

COINS Technical Report 89-113

PRODUCT-SHUFFLE NETWORKS: Toward Reconciling Shuffles and Butterflies

Arnold L. Rosenberg
Department of Computer and Information Science
University of Massachusetts
Amherst, MA 01003

October 27, 1989

Keywords: Interconnection network; parallel architecture; network emulation; butterfly network; cube-connected cycles network; shuffle-exchange network; deBruijn network

Acknowledgment of Support: A portion of this research was supported by NSF Grant CCR-88-12567.

Abstract

We study *product-shuffle* (*PS*) networks, which are direct products of deBruijn networks, as interconnection networks for parallel architectures. PS networks can be viewed as generalizing both butterfly-oriented networks (such as the butterfly and cube-connected cycles networks) and shuffle-oriented networks (such as the deBruijn and shuffle-exchange networks), in the sense that

- PS networks can emulate both butterfly-oriented and shuffle-oriented networks of any size, via emulations that are *work preserving*, i.e., preserve the processor-time product;
- PS networks share many computationally valuable structural features of various butterfly- and shuffle-oriented networks, including *pancyclicity*, *logarithmic diameter*, and *large complete-binary-tree subnetworks*;
- PS networks overcome certain computational deficiencies of butterfly- and shuffle-oriented networks, by containing as subnetworks moderate size *meshes* and *meshes of trees*, networks which butterfly- and shuffle-oriented networks cannot emulate efficiently.

Finally, PS networks attain their communication power at modest cost: they are 8-valent, and they enjoy VLSI layouts that consume only modestly more area than the best layouts of like-sized butterfly- and shuffle-oriented networks.

1. Goals of the Study

The boolean hypercube and its bounded-degree derivatives, such as the *butterfly-oriented* butterfly and cube-connected cycles (CCC) networks and the *shuffle-oriented* shuffle-exchange and deBruijn networks, are among today's dominant interconnection networks for massively parallel architectures. Indeed, architectures based on these networks have been built in both industry and academia.

Among these interconnection networks, the hypercube is the clear favorite because of its efficiency on a broad class of algorithms [5, 7, 9, 19] and its structural uniformity that simplifies programming [18]. The major shortcoming of the hypercube is its high valence.[1] The technological difficulties attendant to implementing such high-valence networks have led to the development of several butterfly-oriented bounded-degree "approximations" of the hypercube, most notably the butterfly and CCC networks [15]. These networks were constructed with a certain important genre of hypercube-algorithm, called *ascend-descend* algorithms [15], in mind and so can emulate the hypercube with little or no slowdown on a

[1] The N-node hypercube has valence (= maximum node-degree) $\log_2 N$.

large, important class of computational problems. Yet, in a sense, butterfly-oriented networks just replace one implementational problem with another, since they use $N \cdot \log_2 N$ nodes (processors) to emulate the N-node hypercube. Further, algebraic, transformations [2] of these large networks yield the smaller, shuffle-oriented bounded-degree "approximations" of the hypercube, most notably the shuffle-exchange[2] [20] and deBruijn [8, 14] networks. Shuffle-oriented networks have only as many nodes as does the hypercube, yet they avoid its large valence; and, on certain computational tasks (including ascend-descend algorithms) they afford one computational efficiency (roughly) equal to that of the butterfly and CCC.

Butterfly- and shuffle-oriented networks are roughly equivalent approximations of the hypercube on a broad class of computational tasks, but it is not clear whether or not one of these network families majorizes the other on *general* computations. Confusingly enough, there is evidence that butterfly- and shuffle-oriented networks have incomparable strengths and weaknesses, and there is countervailing evidence that the two families of networks are equivalent in power. Distinguishing the two families are properties such as the following. The N-node deBruijn network has the computationally useful properties of being *pancyclic* [22],[3] of having diameter exactly $\log_2 N$, and of containing an $(N-1)$-node complete binary tree as a subnetwork; butterfly-oriented networks enjoy none of these. In contrast, the butterfly network enjoys both node-transitivity and a recursive decomposition structure; neither is shared by shuffle-oriented networks. The symmetry and decomposability of butterfly networks are quite useful in developing efficient algorithms. For instance, the efficient routing and sorting algorithms for the butterfly network, in [16] and [17], respectively, exploit the symmetry and recursive structure of the butterfly, hence are not easily transported to any shuffle-oriented network. Further, circumstantial evidence separating the two families (and, indeed, suggesting that shuffle-oriented networks are more powerful than butterfly-oriented ones) arises from studies in [2, 9] of of emulations of one interconnection network by another. These studies use the same strong notion of emulation as we do in this paper.

1. (a) Every even-order (resp., odd-order) butterfly and CCC network can be emulated *with no time loss* (resp., with at most a factor of 2 time loss) by the smallest hypercube that is big enough to hold it [9];

   (b) every known emulation of an N-node shuffle-oriented network by a hypercube incurs slowdown $\Omega(\log N)$ (which is pessimal).

2. (a) An N-node shuffle-oriented network can emulate a like-size butterfly-oriented network with slowdown $O(\log \log N)$;

[2]The shuffle-exchange network can also be derived directly from the hypercube, via a geometric transformation.

[3]That is, the network contains cycles of all lengths $\leq N$.

(b) every known emulation of an N-node shuffle-oriented network by a like-size butterfly-oriented network incurs slowdown $\Omega(\log N)$ (which is pessimal).

Blurring the apparent algorithmic distinctions between the two families of networks is the recent work in [10] which presents a computational framework wherein either of the butterfly or deBruijn network can emulate the other efficiently. In particular, these emulations yield an algorithm for sorting on the deBruijn network that rivals the efficiency of the Reif-Valiant [17] algorithm for butterfly networks.

The present study is motivated in part by the unresolved questions implicit in the preceding paragraph: Which, if either, of the butterfly- and shuffle-oriented networks is the more powerful? Under what circumstances are computationally more complicated emulations of the type introduced in [10] to be preferred to the purely structure-oriented emulations of [2, 9], and vice versa?

> **Remark.** The major argument in favor of the structured-oriented framework is that its emulations yield algorithms that translate programs for the emulated network to equivalent programs for the emulating network. The major argument in favor of the computationally more complicated framework is that it significantly expands the class of networks that a given network can emulate efficiently.

Further motivating our study is the fact that the constructions in both [10] and [2] suggest that efficient emulations of either of these network families by the other are likely to be rather sophisticated and complicated. There would be value, therefore, in having a family of networks which retains most of the structural simplicity of both butterfly- and shuffle-oriented networks, but which can emulate both families of networks *simply* — the emulation procedure should be simple to specify — and *efficiently* — the host architecture should be able to emulate any of the guest architectures *in a work-preserving manner*, i.e., so that the processor-time product is preserved.[4] In this paper, we study such a "least upper bound" family, the *product-shuffle* (*PS*) networks; each PS network is a direct product of two deBruijn networks.

The goals presented thus far can be satisfied by a variety of interconnection networks; indeed, the most simple such network would just superpose a butterfly-oriented network on a like-sized shuffle-oriented one. (Formally, this would amount to taking the union of the edge-sets of the two networks.) Why, then, should we bother with PS networks, which introduce the additional complication of the direct-product structure? The answer

[4] More precisely, an emulation of an n-processor architecture $\mathcal{A}$ by an $(m \leq n)$-processor architecture $\mathcal{B}$ is *work preserving* if $\mathcal{B}$ can emulate any sequence of T computational steps of $\mathcal{A}$ in at most $c(n/m)T$ steps, for some fixed *overhead constant* c.

lies in our willingness to suffer a (very) modest amount of structural complication in return for a (very) considerable amount of simplicity and efficiency in computation. The following example should clarify our concerns. It is shown in [4] that neither butterfly- nor shuffle-oriented networks can emulate meshes (and a variety of other planar networks) efficiently, *using the simple notion of emulation that we study here.* This is a quite serious deficiency, given the importance of mesh-oriented parallel algorithms.[5] We show that every "reasonably shaped" PS network can emulate moderate-size meshes *with no slowdown*, since it contains the meshes *as subnetworks.* Using the stronger notion of emulation in [10], one can emulate meshes on butterfly- or shuffle-oriented networks with only constant-factor slowdown; but, the constants are nontrivial, and the emulation algorithm is rather complicated. There are further beneficial consequences of the direct-product structure of PS networks:

- PS networks contain moderate-size copies of the computationally important mesh-of-trees network [12]; cf. Section 3.4.
- The structure facilitates using PS networks to obtain simple and efficient *load-balanced* work-preserving emulations of other networks; cf. Sections 2.2.B, 4.1, and 4.2.
- The structure facilitates finding rather compact VLSI layouts for PS networks; cf. Section 4.4.

It is not clear that any "superposed," composite butterfly-plus-shuffle network could match all of the benefits that ensue from the direct-product structure of PS networks.

The remainder of the paper is organized into three sections. In Section 2, we formalize the topics of our study and present some simple basic results. Section 3 verifies the large family of subnetworks of PS networks, that we have been alluding to. In Section 4, we compare PS networks with butterfly- and shuffle-oriented networks, demonstrating efficient work-preserving emulations of the latter two families by the former family, and indicating the impossibility of efficient converse emulations. These results show that, within our framework of highly structured emulations (as opposed to the more general framework of [10]), PS networks are strictly more powerful than either butterfly- or shuffle-oriented networks, by more than any constant factor. Moreover, the added power comes at only moderate cost, in that PS networks are 8-valent (in contrast to the 4-valence of butterfly- and shuffle-oriented networks), and PS networks admit VLSI layouts that are only modestly bigger than the best layouts of like-sized butterfly- or shuffle-oriented networks.

[5]The parallel algorithm literature abounds with linear-algebraic and numerical algorithms that conform naturally to the structure of a mesh.

We have thus far used the term "shuffle-oriented network" to refer ambiguously to the deBruijn and shuffle-exchange networks, and the term "butterfly-oriented network" to refer ambiguously to the butterfly and CCC networks. This ambiguity is justified by the fact that the deBruijn and shuffle-exchange networks can each emulate the other with only a factor-of-2 slowdown, and the same is true of the butterfly and CCC networks. We focus on the deBruijn and butterfly networks in the sections that follow, since they yield smaller constant factors in our emulations, and they have richer families of subnetworks.

2. The Formal Framework

2.1. Interconnection Networks as Graphs

As is customary in structural studies of parallel architectures, we restrict attention to arrays of identical processing elements (PEs), and we view the architectures and their underlying interconnection networks as undirected graphs.

> A *directed* graph $\mathcal{G}$ is specified by a set $V_{\mathcal{G}}$ of *nodes* and a multisubset $A_{\mathcal{G}} \subseteq V_{\mathcal{G}} \times V_{\mathcal{G}}$ called *arcs*. One obtains an *undirected* graph $\mathcal{G}'$ from a directed graph $\mathcal{G}$, by "symmetrizing" the set of arcs: one replaces each arc of $\mathcal{G}$ with a pair of mated arcs having opposing directions. We refer to each mated pair of arcs as an *edge* of the graph $\mathcal{G}'$.[6]

The **nodes** of the graph represent the **PEs** of the array, and the **edges** of the graph represent the inter-PE **communication links**. We henceforth use the term "graph" instead of "network."

A. Notation and Terminology

- In phrases like "for all n," n always ranges over the positive integers. For all n, $Z_n =_{\text{def}} \{0, 1, \ldots, n-1\}$, and $\lambda(n) =_{\text{def}} \lceil \log n \rceil$. (All logarithms in the paper are to the base 2.)
- For any set S and positive integer k, S^k denotes the set of all length-k strings of elements of S, and $|S|$ denotes the cardinality of S.
- Given graphs $G = (V_{\mathcal{G}}, E_{\mathcal{G}})$ and $H = (V_{\mathcal{H}}, E_{\mathcal{H}})$, the *(direct) product graph* $\mathcal{G} \times \mathcal{H}$ has node-set $V_{\mathcal{G}} \times V_{\mathcal{H}}$. Let u and v be nodes of $\mathcal{G}$, and let x and y be nodes of $\mathcal{H}$. Then $(\langle u, x\rangle, \langle v, y\rangle)$ is an edge of $\mathcal{G} \times \mathcal{H}$ just when either (u, v) is an edge of $\mathcal{G}$ and $x = y$, or (x, y) is an edge of $\mathcal{H}$ and $u = v$.

[6] Note that we allow self-loops and parallel edges.

- The *degree* of node v of graph $\mathcal{G}$ is the number of edges of $\mathcal{G}$ incident to (i.e., involving) v. The *valence* of $\mathcal{G}$ is the maximum degree of any of its nodes.

B. The Graphs of Interest

Let m, n be positive integers.

The *order-n* **deBruijn graph** $\mathcal{D}(n)$ and the *order-n* **shuffle-exchange graph** $\mathcal{S}(n)$ are the undirected versions of the following directed graphs. Both $\mathcal{D}(n)$ and $\mathcal{S}(n)$ have the set of nodes Z_2^n. For all $\beta \in Z_2$ and $x \in Z_2^{n-1}$:

- In both $\mathcal{D}(n)$ and $\mathcal{S}(n)$, each node of the form βx is connected via a *shuffle arc* to node $x\beta$.
- Additionally:
  - in $\mathcal{D}(n)$, each node of the form βx is connected via a *shuffle-exchange arc* to node $x\bar{\beta}$;
  - in $\mathcal{S}(n)$, each node of the form $x\beta$ is connected via an *exchange arc* to node $x\bar{\beta}$.

Figure 1 depicts the (directed version of the) order-3 deBruijn graph $\mathcal{D}(3)$.

The *order-n* **butterfly graph** $\mathcal{B}(n)$ and the *order-n* **cube-connected cycles graph** (**CCC**, for short) $\mathcal{C}(n)$ have the set of nodes $V_n = Z_n \times Z_2^n$. We call ℓ the *level* of node $\langle \ell,\ x \rangle \in V_n$. For each $\ell \in Z_n$ and each $\vec{\beta} = \beta_0\beta_1 \cdots \beta_{n-1} \in Z_2^n$:

- In both $\mathcal{B}(n)$ and $\mathcal{C}(n)$, each level-ℓ node $v = \langle \ell,\ \vec{\beta} \rangle$ is connected via a *straight-edge* with node $\langle (\ell + 1) \bmod n,\ \vec{\beta} \rangle$;
- Additionally:
  - in $\mathcal{B}(n)$, node v is connected via a *cross-edge* with node
$$\langle (\ell + 1) \bmod n,\ \beta_0\beta_1 \cdots \beta_{\ell-1}\bar{\beta}_\ell\beta_{\ell+1} \cdots \beta_{n-1} \rangle;$$
  - in $\mathcal{C}(n)$, node v is connected via a *level-edge* with node
$$\langle \ell,\ \beta_0\beta_1 \cdots \beta_{\ell-1}\bar{\beta}_\ell\beta_{\ell+1} \cdots \beta_{n-1} \rangle.$$

Figure 2 depicts the order-3 butterfly graph $\mathcal{B}(3)$.

The *order-*(m, n) **product-shuffle graph** (**PS graph**, for short) $\mathcal{P}(m, n)$ is the product graph $\mathcal{D}(m) \times \mathcal{D}(n)$.

Remark. For brevity, we study only the *base-2* versions of our graph families, by dint of our using Z_2 as the underlying alphabet of the graphs' node-sets. One can easily define arbitrary *base-d* versions of the graphs[7] and extend our results with only clerical changes. In [2], we deal with the general, base-d versions of our graph families.

2.2. Emulation via Graph Embeddings

In defining the emulation of one architecture $\mathcal{A}$ by another architecture $\mathcal{B}$, we assume that the PEs of $\mathcal{B}$ are sufficiently powerful to emulate the PEs of $\mathcal{A}$ step for step — so no delay is incurred because of computational steps. We restrict attention to emulations that honor a pulsed computation regimen: Architecture $\mathcal{B}$ alternates phases that emulate one *computation* step of architecture $\mathcal{A}$, with phases that emulate one *communication* step of array $\mathcal{A}$.[8] The slowdown incurred by an emulation arises from two sources. First, we allow emulations that require one PE of $\mathcal{B}$ to play the role of several PEs of $\mathcal{A}$; second, architecture $\mathcal{B}$ must emulate on its interconnection graph communication steps that are tailored to the (possibly very different) structure of the interconnection graph underlying architecture $\mathcal{A}$. The second type of delay results both from mismatched adjacency structures and from congested communication lines. Our study of emulations is based on the following notion of graph embeddings and their costs.

A. Graph Embeddings

An *embedding* of the graph $\mathcal{G}$ in the graph $\mathcal{H}$ is specified by:

- a (possibly many-to-one) *assignment* α of the nodes of $\mathcal{G}$ to the nodes of $\mathcal{H}$:

$$\alpha : V_{\mathcal{G}} \to V_{\mathcal{H}}$$

[A PE of $\mathcal{H}$ must emulate all of the PEs of $\mathcal{G}$ assigned to it via α.]

- a *routing* ρ of each edge (u, v) of $\mathcal{G}$ along a path in $\mathcal{H}$ connecting $\alpha(u)$ and $\alpha(v)$.[9] *[$\mathcal{H}$ must emulate each communication along edge (u, v) of $\mathcal{G}$ by transmitting the "message" along the path $\rho(u, v)$.]*

[7] For illustration, the *base-d order-n deBruijn graph* has node-set Z_d^n and edges connecting each node of the form δx, where $\delta \in Z_d$ and $x \in Z_d^{n-1}$, to all nodes of the form $x\gamma \in Z_d^n$.

[8] This regimen of having $\mathcal{B}$ mimic the exact form of the computation by $\mathcal{A}$ motivates our using the term "emulate" rather than "simulate."

[9] A *length-ℓ path* in $\mathcal{H}$ from node $x \in V_{\mathcal{H}}$ to node $y \in V_{\mathcal{H}}$ is a sequence π of nodes

$$x = z_0, z_1, \ldots, z_{\ell-1} = y$$

such that, for each $0 \leq i < \ell - 1$, $(z_i, z_{i+1}) \in E_{\mathcal{H}}$. By abuse of notation, we write "$(z_i, z_{i+1}) \in \pi$".

B. Slowdown Incurred by an Emulation

A number of factors induce slowdown when architecture $\mathcal{H}$ emulates architecture $\mathcal{G}$. We account for these factors very conservatively, by assuming that in each step of an emulated computation, every PE of $\mathcal{G}$ performs a computation step followed by a communication with all of its neighbors. Clearly, most algorithms will not exercise the resources of $\mathcal{G}$ so exhaustively, and so will be emulated by $\mathcal{H}$ with less slowdown than our accounting procedures indicate. Say that we have an embedding (α, ρ) of $\mathcal{G}$ in $\mathcal{H}$.

- The *load* of the embedding is the maximum number of PEs of $\mathcal{G}$ assigned to any one PE of $\mathcal{H}$:

$$\text{load}(\alpha, \rho) = \max_{v \in V_{\mathcal{H}}} |\alpha^{-1}(v)|$$

[Load induces slowdown because, in each computation phase of the emulation, each PE of $\mathcal{H}$ must emulate a computation step by each PE of $\mathcal{G}$ assigned to it.]

- The *dilation* of the embedding is the maximum amount that the routing ρ "stretches" any edge of $\mathcal{G}$:

$$\text{dilation}(\alpha, \rho) = \max_{(u,v) \in E_{\mathcal{G}}} \text{Length}(\rho(u, v))$$

[Dilation incurs slowdown because every message that crosses link e in $\mathcal{G}$ must traverse path $\rho(e)$ in $\mathcal{H}$.]

- The *I/O-congestion* of the embedding is the ratio of the valence of $\mathcal{H}$ to the valence of $\mathcal{G}$:

$$\text{I/O-congestion}(\alpha, \rho) = \frac{\text{valence}(\mathcal{H})}{\text{valence}(\mathcal{G})}$$

[I/O-congestion incurs slowdown because at each computation step, $\mathcal{G}$ needs poll only valence($\mathcal{G}$) I/O ports, while $\mathcal{H}$ must poll valence($\mathcal{H}$) ports.]

- The *edge-congestion* of the embedding is the maximum number of edges of $\mathcal{G}$ that ρ routes over a single edge of $\mathcal{H}$:

$$\text{edge-congestion}(\alpha, \rho) = \max_{e \in E_{\mathcal{H}}} |\{e' \in E_{\mathcal{G}} \ : \ e \in \rho(e')\}|$$

[Edge-congestion incurs slowdown because the messages that want to cross a congested edge must be queued up. (For simplicity, we are giving each edge of $\mathcal{H}$ the same capacity as a single edge of $\mathcal{G}$.)]

The slowdown due to load and I/O-congestion seem to be unavoidable. In contrast, one can avoid the slowdown due to edge-congestion by increasing the bandwidth of $\mathcal{H}$'s communication links, at the cost of increased hardware and increased layout area. Another

avenue for mitigating the effects of edge-congestion is to orchestrate the communication phases of $\mathcal{H}$, so that message traffic is spread uniformly along the paths of $\mathcal{H}$ that are used to emulate the links of $\mathcal{G}$; this ploy, which allows one to amortize edge-congestion over the paths that create dilation, is used to decrease the slowdown of the emulations in [2]. Another form of orchestrating the communication phases of emulations leads to the following result, which guarantees that load, dilation, and edge-congestion can always be made to combine additively, rather than multiplicatively (as a naive analysis would suggest). The main analysis leading to this result appears in [13]; its extension to the current framework appears in [10].

Proposition 2.1. [10, 13] *Say that one can embed the graph $\mathcal{G}$ in the graph $\mathcal{H}$, with load L, dilation D, and edge-congestion C. Then the architecture $\mathcal{H}$ can emulate T steps of the architecture $\mathcal{G}$ on a general computation in $O(L + C + D)T$ steps.*

Aside from its assuring us that proper scheduling can make the costs of an emulation combine additively rather than multiplicatively, Proposition 2.1 also demonstrates that our purely graph-theoretic formalism is, indeed, modelling the algorithmic situation that we want it to.

Proposition 2.1 points out the importance of balancing loads in emulations, i.e., of keeping the quantity

$$\max_{u,v \in V_{\mathcal{H}}} \left| |\alpha^{-1}(u)| - |\alpha^{-1}(v)| \right|$$

bounded by a constant. All of our emulations will have balanced loads.

Since I/O-congestion is a property only of the structures of $\mathcal{G}$ and $\mathcal{H}$, and not of any embedding of $\mathcal{G}$ in $\mathcal{H}$, we shall not mention it further.

Obviously, one strives to make emulations as "efficient" as possible. One can argue (cf. [10]) that the most important notion of efficiency resides in the notion of *work preservation*, which arises from the following reasoning. When the n-PE architecture $\mathcal{G}$ operates for T steps, it can perform nT atomic operations. If the $(m \leq n)$-PE architecture $\mathcal{H}$ emulates these T steps of $\mathcal{G}$, it requires at least $\lceil n/m \rceil T$ steps to perform the same amount of work. Allowing a (hopefully small) constant factor leeway as overhead for the emulation, we say that the emulation of $\mathcal{G}$ by $\mathcal{H}$ is *work preserving* if $\mathcal{H}$ can emulate *any* T steps of $\mathcal{G}$ in at most $O(\lceil n/m \rceil)T$ steps; cf. footnote 4. We shall strive for work-preserving emulations throughout.

C. Two Simple Emulations

We are finally in a position to formalize our discussion at the end of Section 1, concerning the "equivalence" of the deBruijn and shuffle-exchange graphs, on the one hand, and the butterfly and CCC graphs, on the other hand.

Proposition 2.2. *For all n, we have the following work-preserving emulations:*
(a) *One can embed either of the shuffle-exchange graph $\mathcal{S}(n)$ or the deBruijn graph $\mathcal{D}(n)$ in the other, with load* 1, *dilation* 2, *and edge-congestion* 2.
(b) *One can embed either of the CCC graph $\mathcal{C}(n)$ or the butterfly graph $\mathcal{B}(n)$ in the other, with load* 1, *dilation* 2, *and edge-congestion* 2.

Proof Sketch. For all four of the indicated embeddings, we employ the *identity* assignment. Ignoring edges that are shared by the respective pairs of graphs in the embeddings (hence use the identity routing), we route

- edge $(\beta x, x\bar{\beta})$ of $\mathcal{D}(n)$ along the following length-2 path in $\mathcal{S}(n)$:

$$\beta x \leftrightarrow x\beta \leftrightarrow x\bar{\beta}$$

- edge $(x\beta, x\bar{\beta})$ of $\mathcal{S}(n)$ along the following length-2 path in $\mathcal{D}(n)$:

$$x\beta \leftrightarrow \beta x \leftrightarrow x\bar{\beta}$$

- edge $(\langle \ell, \vec{\beta} \rangle, \langle (\ell+1) \bmod n, \vec{\beta}' \rangle)$ of $\mathcal{B}(n)$ along the following length-2 path in $\mathcal{C}(n)$:

$$\langle \ell, \vec{\beta} \rangle \leftrightarrow \langle \ell, \vec{\beta}' \rangle \leftrightarrow \langle (\ell+1) \bmod n, \vec{\beta}' \rangle$$

- edge $(\langle \ell, \vec{\beta} \rangle, \langle \ell, \vec{\beta}' \rangle)$ of $\mathcal{C}(n)$ along the following length-2 path in $\mathcal{B}(n)$:

$$\langle \ell, \vec{\beta} \rangle \leftrightarrow \langle (\ell+1) \bmod n, \vec{\beta}' \rangle \leftrightarrow \langle \ell, \vec{\beta}' \rangle$$

□

2.3. Structural Characteristics of the Graphs of Interest

The *diameter* (maximum inter-node distance) of a graph $\mathcal{H}$ bounds above both the dilation of any embedding into $\mathcal{H}$ and the time required for any single-node broadcast in $\mathcal{H}$. Therefore, the following table places our emulation results in perspective and provides an interesting comparison of $\mathcal{P}(m,n)$ with its "competitors." One noteworthy point is that PS graphs share diameter (exactly) $\log_2 N$ with deBruijn graphs and hypercubes, although deBruijn graphs acquire their small diameter with valence 4, while PS graphs have valence 8 and hypercubes have valence $\log_2 N$.

Proposition 2.3. *For all n:*

	GRAPH	SIZE	VALENCE	DIAMETER
(a)	$\mathcal{D}(n)$	$N = 2^n$	4	$\log N$
(b)	$\mathcal{B}(n)$	$N = n2^n$	4	$2\log N - 2\log\log N$
(c)	$\mathcal{P}(m,n)$	$N = 2^{m+n}$	8	$\log N$

Proof. Parts (a,b) being well known, we concentrate on part (c). One can proceed from any node $\langle x, y\rangle$ of $\mathcal{P}(m,n)$ to any other node $\langle x', y'\rangle$ by

1. proceeding from node $\langle x, y\rangle$ to node $\langle x', y\rangle$ in at most Length$(x) = m$ steps by mimicking the way one would proceed from node x to node x' in $\mathcal{D}(m)$;
2. proceeding from node $\langle x', y\rangle$ to node $\langle x', y'\rangle$ in at most Length$(y) = n$ steps by mimicking the way one would proceed from node y to node y' in $\mathcal{D}(n)$.

□

3. Computationally Important Subgraphs of $\mathcal{P}(m,n)$

PS graphs contain a variety of computationally useful graphs *as subgraphs*, i.e., as graphs that can be emulated with unit load, edge-congestion, and dilation, hence can be emulated with no slowdown.

3.1. Cycles

The n-node **cycle** $\mathcal{C}(n)$ is the graph whose nodes comprise the set Z_n and whose edges connect each node v with node $(v+1)$ mod n.

It is well known that $\mathcal{D}(n)$ is *hamiltonian* in that it contains the cycle $\mathcal{C}(2^n)$ as a subgraph. In fact, it satisfies the following stronger property.

Lemma 3.1. [22] *For all n, the deBruijn graph $\mathcal{D}(n)$ is* pancyclic*; that is, for all $1 \leq k \leq 2^n$, the k-node cycle $\mathcal{C}(k)$ is a subgraph of $\mathcal{D}(n)$.*[10]

PS graphs share this property, whose computational benefits are exploited in the emulations in [2] and in our Section 4.

[10] Since we allow self-loops and parallel edges, it makes sense to talk about cycles of lengths 1 and 2.

Theorem 3.1. *For all m, n except for $m = n = 1$, the PS graph $\mathcal{P}(m, n)$ is pancyclic.*

Proof. For any choice of m, n other than $m = n = 1$, and for any integer $1 \leq c \leq 2^{m+n}$, we show algorithmically that the cycle $C(c)$ is a subgraph of $\mathcal{P}(m, n)$. Our algorithm assumes that the cycles promised by Lemma 3.1 can be produced algorithmically; cf. [22]. (Note that $\mathcal{P}(1, 1)$ is (essentially) a 4-cycle, whence its exclusion from the Theorem.)

Assume, with no loss of generality, that $m \leq n$ (or else, interchange the roles of m and n in what follows). If the desired cycle length c satisfies $1 \leq c \leq 2^n$, then $C(c)$ is a subgraph of $\mathcal{P}(m, n)$, by Lemma 3.1. Let us restrict attention, therefore, to values of c in the range $2^n < c \leq 2^{m+n}$, in which case we must have $m > 0$.

Now, every integer c in the indicated range admits a unique representation in the form

$$c = a2^n + b$$

with $0 < a \leq 2^m$ and $0 \leq b < 2^n$. The overall strategy of our algorithm is to "hook together" hamiltonian cycles from a of the 2^m copies of $\mathcal{D}(n)$ that comprise $\mathcal{P}(m, n)$, together with a length-b cycle from one additional copy of $\mathcal{D}(n)$ whenever $b > 0$. (In fact, technical difficulties in "hooking up" these cycles will cause us to deviate from this strategy slightly.) To the end of implementing this strategy, we invoke Lemma 3.1 to find a length-d cycle in $\mathcal{D}(m)$, where

$$d = \begin{cases} a & \text{if } b = 0 \\ a + 1 & \text{if } b > 0, \end{cases}$$

and we use this cycle in the natural way to select and order d "consecutive" copies of $\mathcal{D}(n)$, from the 2^m copies that comprise $\mathcal{P}(m, n)$; call the selected copies $\mathcal{D}_0, \mathcal{D}_1, \ldots, \mathcal{D}_{d-1}$.

We describe the mechanism for "hooking the cycles together" via an analysis of cases.

Case 1: $b = 0$, so $d = a$

This is the easiest case, since we have only to "hook together" a set $C_0, C_1, \ldots, C_{a-1}$ of cycles, each C_i being a copy within $\mathcal{D}_i$ of a hamiltonian cycle C of $\mathcal{D}(n)$. We start by selecting any two *independent* edges (x, y) and (u, v) of $\mathcal{D}(n)$, that both lie on the cycle C; since $n \geq 2$, we are sure that these edges exist. Next, we let x_i, y_i, u_i, v_i $(0 \leq i < a)$ denote the instances of the nodes x, y, u, v, respectively, in copy $\mathcal{D}_i$ of $\mathcal{D}(n)$. Assume that the nodes x, y, u, v lie in clockwise order around the cycle C in $\mathcal{D}(n)$, so that each cycle C_i has the form

$$y_i, P_i, u_i, v_i, Q_i, x_i$$

where P_i and Q_i are the intermediate paths that define the cycle.

We are now ready to find a length-c cycle in $\mathcal{P}(m, n)$.

1. Trace the cycle C_0 in $\mathcal{D}_0$ in clockwise order, from node y_0 to node x_0, leaving out the edge that connects the two nodes.

2. Trace the following path to complete the cycle:

$$x_0 \leftrightarrow x_1 \leftrightarrow Q_1 \leftrightarrow v_1 \leftrightarrow v_2 \leftrightarrow Q_2 \leftrightarrow x_2 \leftrightarrow x_3 \leftrightarrow Q_3 \leftrightarrow v_3 \leftrightarrow \cdots$$
$$\leftrightarrow q_{a-1} \leftrightarrow Q_{a-1} \leftrightarrow r_{a-1} \leftrightarrow s_{a-1} \leftrightarrow P_{a-1} \leftrightarrow t_{a-1} \leftrightarrow \cdots$$
$$u_3 \leftrightarrow P_3 \leftrightarrow y_3 \leftrightarrow y_2 \leftrightarrow P_2 \leftrightarrow u_2 \leftrightarrow u_1 \leftrightarrow P_1 \leftrightarrow y_1 \leftrightarrow y_0$$

where

$$q, r, s, t = \begin{cases} x, v, u, y & \text{respectively, if } a \text{ is even} \\ v, x, y, u & \text{respectively, if } a \text{ is odd.} \end{cases}$$

The paths P_i and Q_i and the edges (x_i, y_i) and (u_i, v_i) come from the copies of $\mathcal{D}(n)$, while the edges (x_i, x_{i+1}), (y_i, y_{i+1}), (u_i, u_{i+1}), and (v_i, v_{i+1}) come from the copy of $\mathcal{D}(m)$ we used to order the copies of $\mathcal{D}(n)$.

The reader should be able to fill in details, with the help of Figure 3. □-Case 1

Case 2: $b > 0$, so $d = a + 1$ and $a < 2^m$

The added challenge in this case arises from the need to append a cycle of length b to the chain of a hamiltonian cycles created in Case 1. The mechanism we use depends on the value of b.

Case 2.a: $b \geq 3$

We must alter the procedure of Case 1 in two ways: we must find a copy of a length-b cycle in copy $\mathcal{D}_{a+1}$ of $\mathcal{D}(n)$, and we must ensure that we can "hook" this new cycle to the chain of hamiltonian cycles. The first of these tasks is trivial, by Lemma 3.1; let us call the length-b cycle B. In order to accomplish the second task, we invoke a strong property of $\mathcal{D}(n)$:

Claim. *For any path $x \leftrightarrow y \leftrightarrow z$ in $\mathcal{D}(n)$ involving three distinct nodes, there is a hamiltonian cycle of $\mathcal{D}(n)$ that contains either the edge (x, y) or the edge (y, z).*

Verification. The Claim follows from standard facts about deBruijn graphs.

Fact 1. For all n, $\mathcal{D}(n)$ is the line-graph of $\mathcal{D}(n-1)$.

Fact 2. As a consequence of Fact 1, one can construct a hamiltonian cycle in $\mathcal{D}(n)$ from any eulerian cycle in $\mathcal{D}(n-1)$.

Fact 3. Given any eulerian graph $\mathcal{G}$ and any 2-edge path π in $\mathcal{G}$ whose removal does not disconnect $\mathcal{G}$, one can construct an eulerian cycle in $\mathcal{G}$ which contains π.

Fact 4. The only 2-edge paths whose removal disconnect $\mathcal{D}(n)$ are the paths both of whose edges are incident to either node $\vec{0}$ or node $\vec{1}$.

Because of Fact 1, a 3-*node* path

$$x \leftrightarrow y \leftrightarrow z$$

in $\mathcal{D}(n)$ results from a 3-*edge* path

$$W \leftrightarrow X \leftrightarrow Y \leftrightarrow Z$$

in $\mathcal{D}(n-1)$. Since at most two of these edges can both be incident to either node $\vec{0}$ or node $\vec{1}$ in $\mathcal{D}(n-1)$, it follows by Facts 3 and 4 that there is an eulerian cycle in $\mathcal{D}(n-1)$ passing through either the path

$$W \leftrightarrow X \leftrightarrow Y$$

or the path

$$X \leftrightarrow Y \leftrightarrow Z.$$

Fact 2 assures us that, in the former case, there is a hamiltonian cycle in $\mathcal{D}(n)$ passing through edge (x, y), while in the latter case, there is a hamiltonian cycle passing through edge (y, z). □-Claim

By dint of the Claim and the fact that $b \geq 3$, we can find an edge e of the length-b cycle B in $\mathcal{D}_{a+1}$, that lies on a hamiltonian cycle of $\mathcal{D}(n)$. Let us choose edge e as the edge (x, y) of Case 1 if a is even or as the edge (u, v) of Case 1 if a is odd. We then alter the trajectory of the length-c cycle after the initial path within $\mathcal{D}_0$, by replacing the length-2 path

$$r_{a-1} \leftrightarrow s_{a-1}$$

with the path

$$r_{a-1} \leftrightarrow r_a \leftrightarrow S \leftrightarrow s_a \leftrightarrow s_{a-1}$$

where r, s are as in Case 1, and S is the path within cycle B that connects nodes r_a and s_a in $\mathcal{D}_a$ once edge (r_a, s_a) is removed. □-Case 2.a

Case 2.b: $b = 2$

We proceed exactly as in Case 1, except that we alter the trajectory of the length-c cycle after the initial path within $\mathcal{D}_0$, by replacing the length-2 path

$$r_{a-1} \leftrightarrow s_{a-1}$$

with the length-4 path

$$r_{a-1} \leftrightarrow r_a \leftrightarrow s_a \leftrightarrow s_{a-1}$$

where r, s are as in Case 1. □-Case 2.b

Case 2.c: $b = 1$

We branch immediately on the value of n.

Case 2.c.i: When $n = 2$, we proceed exactly as in Case 1, until we have to deal with copies $a-1$ and a of $\mathcal{D}(n)$. At that point we replace the path

$$q_{a-1} \leftrightarrow Q_{a-1} \leftrightarrow r_{a-1} \leftrightarrow s_{a-1} \leftrightarrow P_{a-1} \leftrightarrow t_{a-1}$$

from Case 1 with a length-5 path (note that $5 = 2^n + 1$) of the form

$$q_{a-1} \leftrightarrow p_1 \leftrightarrow p_2 \leftrightarrow p_3 \leftrightarrow t_{a-1}$$

in the subgraph of $\mathcal{P}(m,n)$ induced on copies $a-1$ and a of $\mathcal{D}(n)$. An exhaustive analysis which depends on the independence of edges (x,y) and (u,v) assures us that this length-5 path exists. □-Case 2.c.i

Case 2.c.ii: When $n > 2$, we alter Case 1 by insisting that at least one of the independent edges (x,y) and (u,v) not touch either node $\vec{0}$ or node $\vec{1}$ of $\mathcal{D}(n)$. (Note that this is impossible when $n = 2$.) Say, without loss of generality, that node $\vec{0}$ is not touched by either edge.

Having thus restricted the choice of these edges, we proceed exactly as in Case 2.b ($b = 2$), with the following exception. Once having found the cycle produced in Case 2.b -- which cycle has length $c+1$ — we remove the instance of node $\vec{0}$ of $\mathcal{D}(n)$ from whichever of P_{a-1} or Q_{a-1} contains an instance of $\vec{0}$. (One must, because of our restriction.) Since every hamiltonian cycle in $\mathcal{D}(n)$ contains the path

$$1\vec{0} \leftrightarrow \vec{0} \leftrightarrow \vec{0}1,$$

the elision of node $\vec{0}$ does not cut our cycle: it just shortens it, as desired. □-Case 2.c.ii

This case analysis completes the proof. □

Very little of the proof of Theorem 3.1 depends on properties that are peculiar to deBruijn graphs. In fact, F. Annexstein and M. Baumslag [personal communication] have observed that an altered version of the proof will establish the following.

Proposition 3.1. *Let $\mathcal{G}$ and $\mathcal{H}$ be pancyclic graphs, one of which — say $\mathcal{G}$ — has an even number of nodes. Suppose that for every integer $2 \leq \ell \leq |\mathcal{G}|$, $\mathcal{G}$ has a length-ℓ cycle that shares an edge with a hamiltonian cycle. Then the product graph $\mathcal{G} \times \mathcal{H}$ is pancyclic, except when $|\mathcal{G}| = |\mathcal{H}| = 2$.*

3.2. Meshes

The $m \times n$ **(toroidal) mesh** $\mathcal{M}(m,n)$ is (isomorphic to) the product graph $\mathcal{C}(m) \times \mathcal{C}(n)$.

One corollary of the main result in [4] is that no butterfly- or shuffle-oriented graph $\mathcal{G}$ can emulate meshes with only constant slowdown (using our notion of emulation); it follows, of course, that $\mathcal{G}$ cannot contain a large mesh as a subgraph. In contrast, PS graphs contain moderate size meshes as subgraphs, as indicated in the following corollary of Theorem 3.1.

Corollary 3.1. *For all m, n and all $p \leq 2^m$ and $q \leq 2^n$, the PS graph $\mathcal{P}(m,n)$ contains the mesh $\mathcal{M}(p,q)$ as a subgraph.*

3.3. Complete Binary Trees

The height-n **complete binary tree** $\mathcal{T}(n)$ is the graph whose $2^n - 1$ nodes comprise the set $\bigcup_{k=0}^{n-1} Z_2^k$ of binary strings of length $< n$ and whose edges connect each node x of length $< n-1$ with nodes $x0$ and $x1$.

Complete binary trees are very useful computational structures, most obviously for broadcasting, but also for emulations [4]. Thus, the following obvious result points out one of the most useful properties of deBruijn graphs; cf. [14].

Lemma 3.2. *For all n, the deBruijn graph $\mathcal{D}(n)$ contains the complete binary tree $\mathcal{T}(n)$ as a subgraph, rooted at node $\vec{0}1$.*

While PS graphs cannot match the fact that the N-node deBruijn graph contains the $(N-1)$-node complete binary tree, they do come within a factor of 2 of matching it.

Theorem 3.2. *For all m, n, the PS graph $\mathcal{P}(m,n)$ contains the complete binary tree $\mathcal{T}(m+n-1)$ as a subgraph.*

Proof. We find an instance of $\mathcal{T}(m+n-1)$ rooted at node $v_0 = \langle \vec{0}1, \vec{0}1 \rangle$ of $\mathcal{P}(m,n)$, as follows. We first invoke Lemma 3.2 to find a copy of $\mathcal{T}(n)$ in "copy $\vec{0}1$" of $\mathcal{D}(n)$, rooted at node v_0 and having leaves of the form $\langle \vec{0}1, x \rangle$ for some $x \in Z_2^n$. We then invoke Lemma 3.2 once for each "copy" of $\mathcal{D}(m)$ (2^{n-1} times in all) to find a copy of $\mathcal{T}(m)$ rooted at each of these leaves. □

To place Theorem 3.2 in perspective, the efficient embedding of complete binary trees in butterfly graphs presented in [4] promises only constant (as oppose to unit) dilation and utilizes only roughly one-eighth of the nodes of the host butterfly. (These constants can be improved somewhat, but not to unity.)

3.4. Meshes of Trees

The $m \times n$ **mesh of trees** $\mathcal{MT}(m,n)$ (m and n being powers of 2) is obtained from the $m \times n$ mesh by

- eliminating all mesh edges
- erecting a copy of the complete binary tree $\mathcal{T}(\lambda(m))$ along each column, using the column nodes as leaves of the tree
- erecting a copy of the complete binary tree $\mathcal{T}(\lambda(n))$ along each row, using the row nodes as leaves of the tree.

Mesh of trees graphs have been shown to be quite powerful computationally [12]. One can prove, using Lemma 3.2, that PS graphs contain at least moderate size such graphs as subgraphs.

Theorem 3.3. *For all m, n and all powers of 2 $p \leq m/2$, $q \leq n/2$, the PS graph $\mathcal{P}(m,n)$ contains the mesh of trees $\mathcal{MT}(p,q)$ as a subgraph.*

Proof. Let us denote by x_i, $1 \leq i \leq p$, the i^{th} leaf leaf of $\mathcal{T}(\lambda(2p))$ and by y_i, $1 \leq i \leq q$, the i^{th} leaf of $\mathcal{T}(\lambda(2q))$. By Lemma 3.2, $\mathcal{P}(m,n)$ contains the product graph $\mathcal{T}(\lambda(2p)) \times \mathcal{T}(\lambda(2q))$ as a subgraph. Consequently, $\mathcal{P}(m,n)$ contains as a subgraph every tree of the form $\{x_i\} \times \mathcal{T}(\lambda(2q))$, as well as every tree of the form $\mathcal{T}(\lambda(2p)) \times \{y_i\}$. The union of all of these subgraphs is $\mathcal{MT}(p,q)$. □

4. PS vs. Butterfly vs. deBruijn Networks

In this section we demonstrate that, relative to our notion of network emulation, PS graphs have strictly more communication power than either shuffle- or butterfly-oriented graphs. Our demonstration consists of efficient embeddings of both deBruijn graphs (Section 4.1) and butterfly graphs (Section 4.2) in PS graphs, followed by a proof that PS graphs cannot be embedded efficiently in either of the other two families (Section 4.3). We close with a discussion in Section 4.4 of the price one pays for the additional power of PS graphs.

Our embeddings of deBruijn and butterfly graphs in PS graphs are presented in two stages, the first assuming that the guest and host graphs in the embedding have the same number of nodes and the second assuming that the host PS graph is smaller than the guest.

4.1. PS Networks Emulating deBruijn Networks

We consider first the (technically) easier problem of emulating deBruijn graphs on PS graphs.

Lemma 4.1. *For all m and n, one can embed the deBruijn graph $\mathcal{D}(m+n)$ in the PS graph $\mathcal{P}(m,n)$, with load 1, dilation 2, and edge-congestion 2.*

Proof. Each node x of $\mathcal{D}(m+n)$ is a length-$(m+n)$ binary string. Let $(x)_m$ denote the length-m prefix of this string, and let $[x]_n$ denote the length-n suffix of this string. The assignment function of the desired embedding is given by:

$$\alpha(x) = \langle (x)_m, [x]_n \rangle$$

for all $x \in Z_2^{m+n}$. Each edge of $\mathcal{D}(m+n)$ has the form $(\beta w, w\delta)$ for $\beta \in Z_2$, $w \in Z_2^{m+n-1}$, and $\delta \in \{\beta, \bar{\beta}\}$. Let us write each $w \in Z_2^{m+n-1}$ in the form $w = u\gamma v$, with $u \in Z_2^{m-1}$, $\gamma \in Z_2$, and $v \in Z_2^{n-1}$. Then the routing function ρ of the embedding realizes the edge

$$(\beta w, w\delta) = (\beta u\gamma v, u\gamma v\delta)$$

via the following length-2 path in $\mathcal{P}(m,n)$:

$$\begin{aligned} \alpha(\beta w) = \alpha(\beta u\gamma v) = \langle \beta u, \gamma v \rangle &\leftrightarrow \langle u\gamma, \gamma v \rangle \\ &\leftrightarrow \langle u\gamma, v\delta \rangle = \alpha(u\gamma v\delta) = \alpha(w\delta) \end{aligned}$$

This embedding clearly has dilation 2. The claimed edge-congestion follows from the facts that the first edge in the length-2 path identifies the edge of $\mathcal{D}(m+n)$ being emulated, up to the identity of δ, while the second edge identifies the edge being emulated, up to the identity of β. □

Theorem 4.1. *For all n and all p and q with $p+q \leq n$, one can embed the deBruijn graph $\mathcal{D}(n)$ in the PS graph $\mathcal{P}(p,q)$, with load 2^{n-p-q}, dilation 4, and edge-congestion $2^{n-p-q+2}$. This leads to a* work-preserving *emulation of $\mathcal{D}(n)$ by $\mathcal{P}(p,q)$.*

Proof. We use Lemma 4.1 to embed $\mathcal{D}(n)$ in $\mathcal{P}(n-p-q, p+q)$, with load 1, dilation 2, and edge-congestion 2.

We then use a *projection embedding* to embed $\mathcal{P}(n-p-q, p+q)$ in $\mathcal{D}(p+q)$, with load 2^{n-p-q}, dilation 1, and edge-congestion 2^{n-p-q}. The projection embedding assigns each node $\langle x, y \rangle$ of $\mathcal{P}(n-p-q, p+q)$ to node y of $\mathcal{D}(p+q)$ and routes edges in the naive (edge-to-edge) way.

Finally, we use Lemma 4.1 a second time, to embed $\mathcal{D}(p+q)$ in $\mathcal{P}(p,q)$, with load 1, dilation 2, and edge-congestion 2.

Since our cost measures multiply when embeddings are composed, we can invoke Proposition 2.1 to complete the proof. □

4.2. PS Networks Emulating Butterfly Networks

Lemma 4.2. [2] *For all n, one can embed the butterfly graph $\mathcal{B}(n)$ in the PS graph $\mathcal{P}(\lambda(n), n)$, with load 1, dilation 2, and edge-congestion 2.*

Proof. We sketch the proof from [2]. By the pancyclicity of deBruijn graphs (Lemma 3.1), it suffices to embed $\mathcal{B}(n)$ in the product graph $\mathcal{C}(n) \times \mathcal{D}(n)$, with unit load, dilation 2 and edge-congestion 2.

We label the nodes of $\mathcal{B}(n)$ with strings from Z_2^n via the following inductive procedure that is implicit in [2]; cf. Figure 4.

1. Label node $\langle 0, \vec{0} \rangle$ of $\mathcal{B}(n)$ with the string $\vec{0}$.
2. If level-ℓ node v ($\ell \in Z_n$) is labelled with string $L(v)$, then label the straight-edge (resp., the cross-edge) neighbor of node v on level $(\ell + 1)$ mod n with the *shuffle* (resp., the *shuffle-exchange*) of $L(v)$.

Now, isolate any two consecutive levels of the labelled $\mathcal{B}(n)$, together with the 2^{n+1} edges that connect the levels; cf. Figure 5. Produce the 2^n-node graph $\mathcal{G}_n$ from the isolated levels by identifying like-labelled nodes and eliminating self-loops. Our labelling procedure guarantees that:

Claim. *For any two consecutive levels of $\mathcal{B}(n)$, the graph $\mathcal{G}_n$ is isomorphic to $\mathcal{D}(n)$.*

The result is now direct: To embed $\mathcal{B}(n)$ in $\mathcal{C}(n) \times \mathcal{D}(n)$:

- Assign level-ℓ node v of $\mathcal{B}(n)$ to node $L(v)$ of copy ℓ of $\mathcal{D}(n)$, where $L(v) \in Z_2^n$ is the label assigned to node v by the indicated procedure.
- Route edge $(\langle \ell, x \rangle, \langle \ell', y \rangle)$ of $\mathcal{B}(n)$ within $\mathcal{C}(n) \times \mathcal{D}(n)$ via the length-2 path:

$$\langle \ell, L(\langle \ell, x \rangle) \rangle \leftrightarrow \langle \ell, L(\langle \ell', y \rangle) \rangle \leftrightarrow \langle \ell', L(\langle \ell', y \rangle) \rangle.$$

  Thus, we first route within a copy of $\mathcal{D}(n)$ and then between copies.

The described embedding clearly has unit load and dilation 2. The claimed edge-congestion is incurred because nodes in both $\mathcal{D}(n)$ and $\mathcal{B}(n)$ have two "successors" and two "predecessors." □

Theorem 4.2. *For all n and all p and q with $q \leq n$ and $\lambda(q) \leq p \leq n$, one can embed the butterfly graph $\mathcal{B}(n)$ in the PS graph $\mathcal{P}(p, q)$, with load and edge-congestion $\lceil n/q \rceil 2^{n-q}$ and dilation 2. This leads to a* work-preserving *emulation of $\mathcal{B}(n)$ by $\mathcal{P}(p, q)$.*

Proof. Our embedding proceeds in stages.

We begin by embedding $\mathcal{B}(n)$ in a graph $\mathcal{G}(n,q)$ which is defined implicitly via the following embedding, (α_1, ρ_1), where both α_1 and ρ_1 are *surjective* mappings. For each node $\langle \ell, x \rangle$ of $\mathcal{B}(n)$,

$$\alpha_1(\langle \ell, x \rangle) = \langle \ell, [x]_q \rangle.$$

(Recall that $[x]_q$ is the length-q suffix of the string x.) ρ_1 routes each edge of $\mathcal{B}(n)$ naively, via an edge of $\mathcal{G}(n,q)$. The embedding thus defined has load and edge-congestion 2^{n-q} and unit dilation.

In order to see what the next stage of our composite embedding needs to accomplish, let us consider what the graph $\mathcal{G}(n,q)$ looks like. $\mathcal{G}(n,q)$ consists of n levels. The induced graph on levels $0, 1, \ldots, q$ of $\mathcal{G}(n,q)$ is (isomorphic to) the induced graph on the first $q+1$ levels of $\mathcal{B}(q+1)$. The remainder of $\mathcal{G}(n,q)$ consists of 2^q node-disjoint length-$(n-q+1)$ paths, each path having the form

$$\langle q, x \rangle \leftrightarrow \langle q+1, x \rangle \leftrightarrow \cdots \leftrightarrow \langle n-1, x \rangle \leftrightarrow \langle 0, x \rangle$$

for some length-q binary string x.

Now we embed $\mathcal{G}(n,q)$ in $\mathcal{G}(p,q)$, by "folding" the "dangling paths" at levels $p, p+1, \ldots, n$ of $\mathcal{G}(n,q)$ into the top p levels of the graph, and then identifying levels 0 and p of the graph. The "folding" is accomplished via the assignment function α_2 defined by:

$$\alpha_2(\langle q+k, x \rangle) = \alpha_2(\langle n-k, x \rangle) = \langle k \bmod n, x \rangle$$

for all $x \in Z_2^q$ and all

$$0 \leq k < \frac{n - p + ((n-p) \bmod 2)}{2};$$

when $n-p$ is even, the definition of α_2 must be completed by the assignment

$$\alpha_2\left(\left\langle \frac{n+p}{2}, x \right\rangle\right) = \left\langle \left(\frac{n-p}{2}\right) \bmod n, x \right\rangle.$$

Once again, we employ the naive (edge-to-edge) routing to complete the specification of the embedding. One verifies easily that this embedding has load and edge-congestion $O(\lceil n/p \rceil)$ and unit dilation.

Finally, we embed $\mathcal{G}(p,q)$ in $\mathcal{P}(p,q)$. This can be done indirectly by invoking the proof (rather than statement) of Lemma 4.2. In that proof, $\mathcal{B}(n)$ is embedded in $\mathcal{C}(n) \times \mathcal{D}(n)$ with unit load and with dilation and edge-congestion 2. Precisely the same reasoning embeds $\mathcal{G}(p,q)$ in $\mathcal{C}(p) \times \mathcal{D}(q)$ with the same costs. Our embedding of $\mathcal{G}(p,q)$ in $\mathcal{P}(p,q)$ is completed by noting that, by Lemma 3.1, $\mathcal{C}(p) \times \mathcal{D}(q)$ is a subgraph of $\mathcal{P}(p,q)$.

Since our cost measures multiply when embeddings are composed, we can invoke Proposition 2.1 to complete the proof. □

4.3. The Converse Emulations

In the framework of our strong notion of emulation, PS graphs are strictly more powerful than either butterfly or deBruijn graphs, in the sense of the following result. Note how much stronger the result is for butterfly graphs than for deBruijn graphs, both in terms of quantification and dilation.

Theorem 4.3. (a) *For all m, n, any embedding of the PS graph $\mathcal{P}(m,n)$ in any butterfly graph must have dilation $\Omega(\min(m,n))$.*
(b) *For all m, n, any embedding of the PS graph $\mathcal{P}(m,n)$ in the deBruijn graph $\mathcal{D}(m+n)$ must have dilation $\Omega(\log \min(m,n))$.*

Proof. Let $M = 2^{\min(m,n)}$. By Corollary 3.1, $\mathcal{P}(m,n)$ contains the $M \times M$ mesh $\mathcal{M}(M,M)$ as a subgraph. It is proved in [4] that any embedding of $\mathcal{M}(M,M)$ in *any* butterfly graph must have dilation $\Omega(\log M)$. It is also proved there that any embedding of $\mathcal{M}(M,M)$ in a like-sized deBruijn graph must have dilation $\Omega(\log M)$. □

The lower bounds of Theorem 4.3 grow faster than any constant, thus justifying our assertion about the power of PS graphs; however, each of these lower bounds is smaller than the best known corresponding upper bound. We do not know at this point whether to believe that the upper bounds can be lowered or that the lower bounds can be raised.

4.4. Area-Efficient VLSI Layouts of the Networks

The additional power of PS graphs over both butterfly and deBruijn graphs comes at modest cost. First, and obviously, PS graphs are 8-valent while their competitors are 4-valent. Less obviously, PS graphs admit VLSI layouts which are only modestly more consumptive of area than the most efficient layout of either of the other two graphs. We refer the reader to [6, 21] for background on the formal framework and techniques of analysis for VLSI layouts.

We begin with the layout requirements of deBruijn and butterfly graphs.

Theorem 4.4. (a) [11] *The deBruijn graph $\mathcal{D}(m+n)$ admits a VLSI layout in a "box" of dimensions*

$$O\left(\frac{2^{m+n}}{m+n}\right) \times O\left(\frac{2^{m+n}}{m+n}\right).$$

(b) [21] *Any VLSI layout of the deBruijn graph $\mathcal{D}(m+n)$ consumes area*

$$\Omega\left(\frac{4^{m+n}}{(m+n)^2}\right).$$

(c) [21] *The area-minimal VLSI layouts of the butterfly graph $\mathcal{B}(n)$, which has $N = n2^n$ nodes, consume area*

$$\Theta\left(\frac{N^2}{\log^2 N}\right) = \Theta\left(\frac{4^{n+\lambda(n)}}{n^2}\right).$$

Now we turn to the layout area of PS graphs.

Theorem 4.5. *The PS graph $\mathcal{P}(m,n)$ admits a VLSI layout of area*

$$O\left(\frac{4^{m+n}}{mn}\right).$$

Proof. We use the layouts guaranteed by Theorem 4.4(a) and by the following Lemma to obtain a VLSI layout of $\mathcal{P}(m,n)$ with the advertised area.

Lemma 4.3. [6] *The deBruijn graph $\mathcal{D}(n)$ admits a* collinear *VLSI layout in a "box" of dimensions*

$$O\left(\frac{2^n}{n}\right) \times O(2^n),$$

i.e., a layout in which all nodes are laid out in a line.

Assume, with no loss of generality, that $m \leq n$. Stack 2^m copies of the area-efficient layout of $\mathcal{D}(n)$ from part (a) of Lemma 4.3, aligned so that, for each node v of $\mathcal{D}(n)$, all copies of v are lined up in the same vertical track. For each node v of $\mathcal{D}(n)$, allocate $2^m/m$ new vertical tracks, and use these tracks to route a collinear layout of $\mathcal{D}(m)$ via the layout of Lemma 4.3(b), using the copies of v as nodes. Easily supplied details turn this schematic description into a layout of $\mathcal{P}(m,n)$, whose area satisfies the bound of the Theorem. □

5. Concluding Remarks

Permutation Routing. Proposition 2.3 indicates that PS networks can match the efficiency of deBruijn networks and exceed the efficiency of butterfly networks on single point-to-point communications and on single-source broadcasts. Recent work [1] shows PS networks to be competitive with the other two networks also on (deterministic, off-line) permutation routing.

Proposition 5.1. [1] *There is a deterministic algorithm that routes any permutation on the PS network $\mathcal{P}(m,n)$ in time*

$$\leq 2(m+n) + 2\min(m,n) - 3.$$

Further Challenges. Among the unresolved problems in the study of hypercube-derivative networks, the most inviting seek definitive answers to the questions of how efficiently the hypercube and its derivatives (including PS graphs) can emulate one another. Although certain of these questions have been resolved within the more comprehensive framework of [10], there are practical, as well as intellectual, reasons to determine whether or not our simpler framework yields the same answers to these questions. Even after all of these individual questions have been answered, it will still be an interesting challenge to adduce underlying principles that explain the answers (along the lines of the algebraic development in [2]).

Acknowledgments. It is a pleasure to thank Fred Annexstein, Marc Baumslag, Lenny Heath, Tom Leighton, Seth Malitz, and the anonymous referees for helpful comments and suggestions.

6. References

1. F. Annexstein and M. Baumslag (1989): A unified approach to global permutation routing on parallel networks. Typescript, Univ. Massachusetts; submitted for publication.

2. F. Annexstein, M. Baumslag, A.L. Rosenberg (1987): Group action graphs and parallel architectures. *SIAM J. Comput.*, to appear.

3. M. Baumslag and A.L. Rosenberg (1989): Processor-time tradeoffs for algebraically specified interconnection networks (tentative title). In preparation, Univ. Massachusetts.

4. S.N. Bhatt, F.R.K. Chung, J.-W. Hong, F.T. Leighton, A.L. Rosenberg (1988): Optimal simulations by butterfly networks. Tech. Rpt. 88-55, Univ. Massachusetts; see a preliminary version in *20th ACM Symp. on Theory of Computing,* 192-204.

5. S.N. Bhatt, F.R.K. Chung, F.T. Leighton, A.L. Rosenberg (1988): Efficient embeddings of trees in hypercubes. Tech. Rpt., Univ. Massachusetts; submitted for publication. See also, Optimal simulations of tree machines. *27th IEEE Symp. on Foundations of Computer Science* (1986) 274-282.

6. S.N. Bhatt and F.T. Leighton (1984): A framework for solving VLSI graph layout problems. *J. Comp. Syst. Sci. 28*, 300-343.

7. R.M. Chamberlain (1988): Gray codes, Fast Fourier Transforms and hypercubes. *Parallel Computing 6*, 225-233.

8. N.G. DeBruijn (1946): A combinatorial problem. *Proc. Akademe Van Weteschappen 49*, Part 2, 758-764.

9. D.S. Greenberg, L.S. Heath and A.L. Rosenberg (1989): Optimal embeddings of butterfly-like graphs in the hypercube. *Math. Syst. Th.*, to appear.

10. R. Koch, F.T. Leighton, B. Maggs, S. Rao, A.L. Rosenberg (1989): Work-preserving emulations of fixed-connection networks. *21st ACM Symp. on Theory of Computing*, 227-240.

11. F.T. Leighton (1983): *Complexity Issues in VLSI: Optimal Layouts for the Shuffle-Exchange Graph and Other Networks.* MIT Press, Cambridge, MA.

12. F.T. Leighton (1984): Parallel computation using meshes of trees. *1983 Workshop on Graph-Theoretic Concepts in Computer Science*, Trauner Verlag, Linz, pp. 200-218.

13. F.T. Leighton, B. Maggs, S. Rao (1988): Universal packet routing algorithms. *29th IEEE Symp. on Foundations of Computer Science*, 256-269.

14. D.K. Pradhan and M.R. Samatham (1988): The deBruijn multiprocessor network: A versatile parallel processing and sorting network for VLSI. *IEEE Trans. Comp. 38*, 567-581.

15. F.P. Preparata and J.E. Vuillemin (1981): The cube-connected cycles: a versatile network for parallel computation. *C. ACM 24*, 300-309.

16. A.G. Ranade (1987): How to emulate shared memory. *28th IEEE Symp. on Foundations of Computer Science*, 185-194.

17. J.H. Reif and L.G. Valiant (1987): A logarithmic time sort for linear graphs. *J. ACM 34*, 60-76.

18. Y. Saad and M.H. Schultz (1988): Topological properties of hypercubes. *IEEE Trans. Comp. 37*, 867-872.

19. C. Stanfill (1987): Communications architecture in the Connection Machine system. Tech. Rpt. HA87-3, Thinking Machines Corp.

20. H. Stone (1971): Parallel processing with the perfect shuffle. *IEEE Trans. Comp., C-20,* 153-161.

21. C.D. Thompson (1980): *A complexity theory for VLSI.* Ph.D. Thesis, CMU.

22. M. Yoeli (1962): Binary ring sequences. *Amer. Math. Monthly 69,* 852-855.

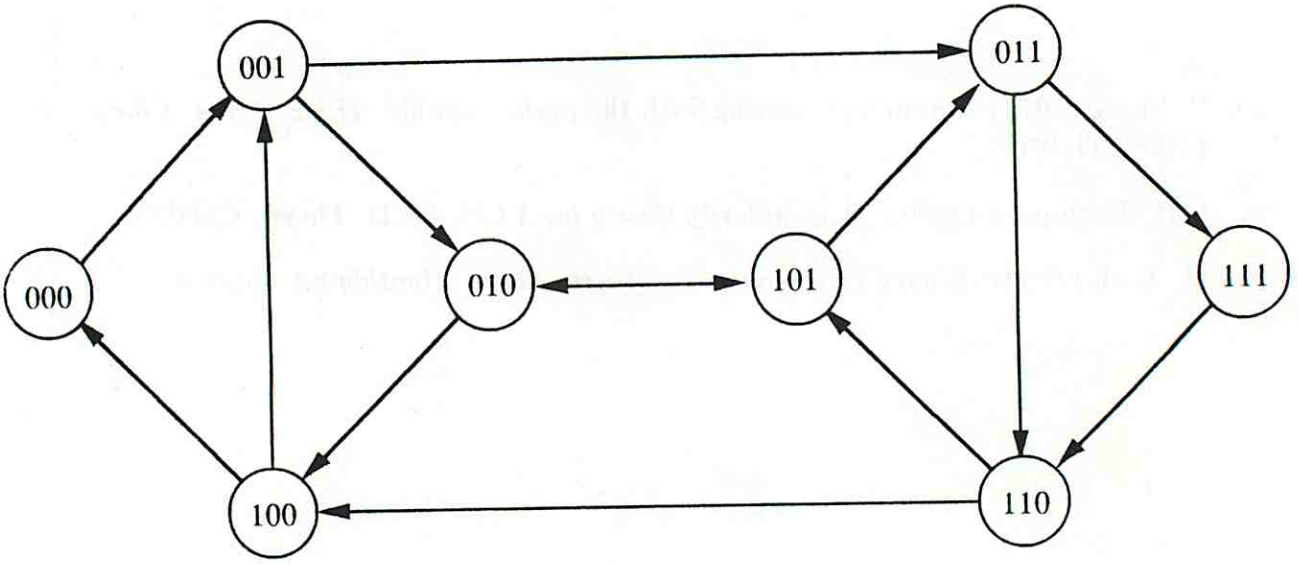

Figure 1. The (directed version of the) order-3 deBruijn graph $\mathcal{D}(3)$.

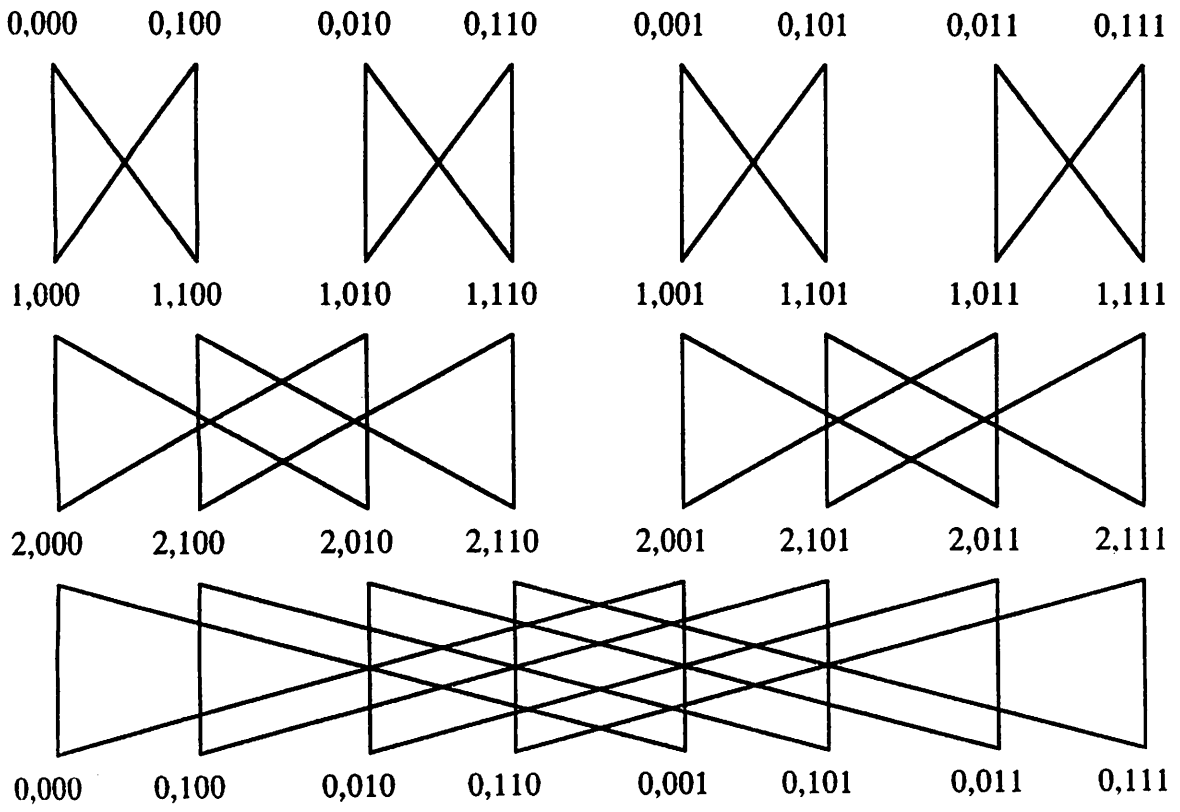

Figure 2. The order-3 butterfly graph $\mathcal{B}(3)$ with level 0 replicated to aid visualization.

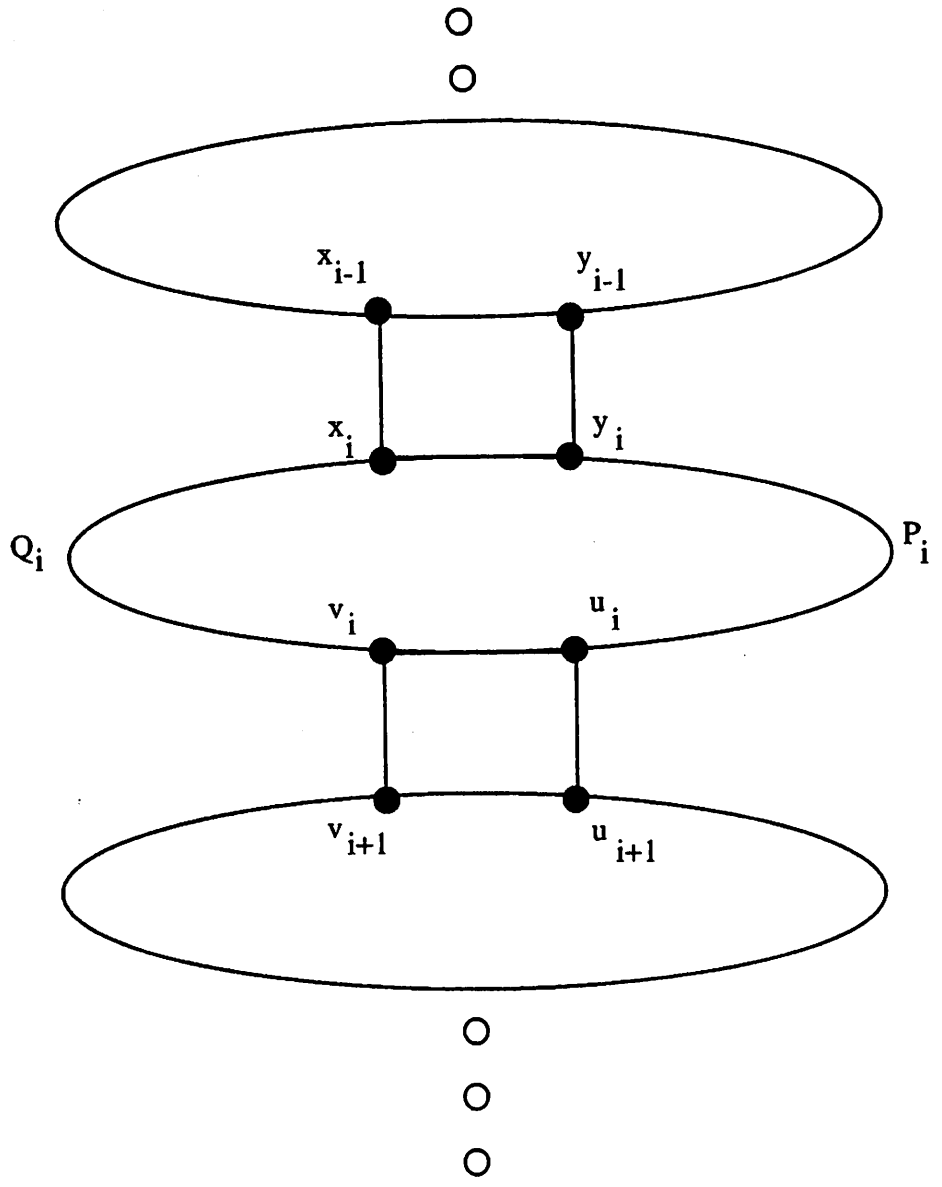

Figure 3. A schematic view of the cycle constructed in Theorem 3.1.

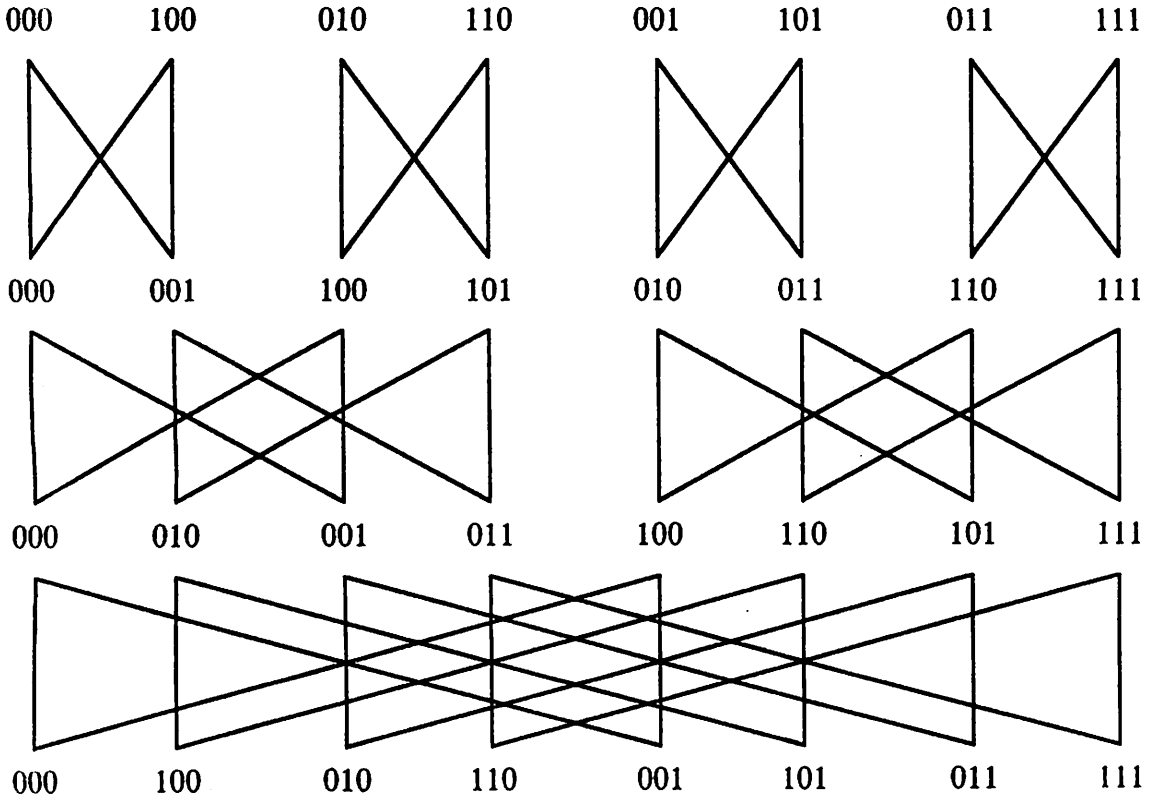

Figure 4. $\mathcal{B}(3)$ with the shuffle-oriented labelling of Lemma 4.2; level 0 is replicated to aid visualization.

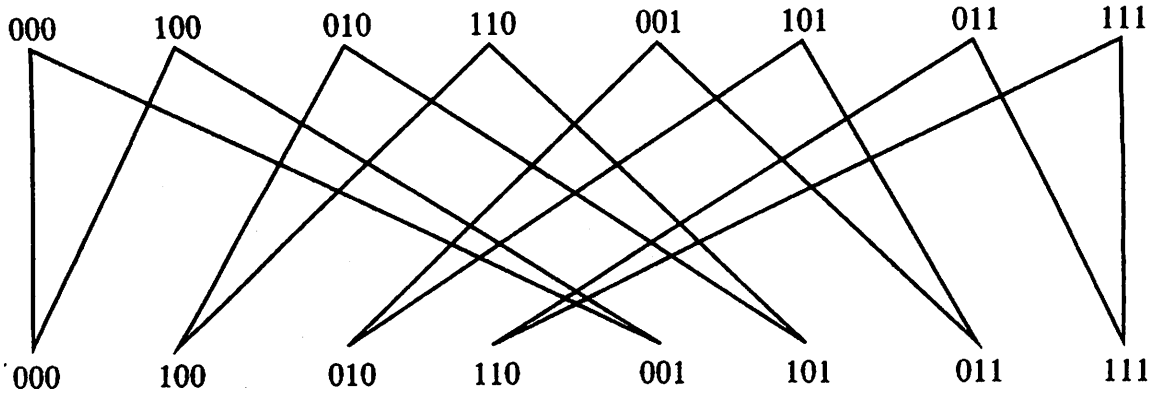

Figure 5. Two consecutive levels of $\mathcal{B}(3)$ with the shuffle-oriented labelling; "columns" are permuted to help visualize the identification.