Product-shuffle networks: toward reconciling shuffles and butterflies

Arnold L. Rosenberg

Department of Computer and Information Science, University of Massachusetts, Amherst, MA 01003, USA

Received 16 May 1989
Revised 21 May 1990

Abstract


We study product-shuffle (PS) networks, which are direct products of de Bruijn 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 de Bruijn 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;

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.

Keywords. Interconnection network, parallel architecture, network emulation, direct-product network, butterfly network, cube-connected cycles network, shuffle-exchange network, de Bruijn network.

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

* A portion of this research was supported by NSF Grant CCR-88-12567.
shuffle-exchange and de Bruijn 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 algorithm [6, 8, 11, 21] and its structural uniformity that simplifies programming [19]. The major shortcoming of the hypercube is its high valence. 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 [16]. These networks were constructed with a certain important genre of hypercube algorithm, called ascend–descend algorithms [16], in mind and so can emulate the hypercube with little or no slowdown on a 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 [3] of these large networks yield the smaller, shuffle-oriented bounded-degree “approximations” of the hypercube, most notably the shuffle-exchange [22] and de Bruijn [9, 20] 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 de Bruijn network has the computationally useful properties of being pancyclic\(^1\) [24], 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 [17] and [18], 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

---

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

\(^2\) 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 \( \leq N \).
shuffle-oriented networks are more powerful than butterfly-oriented ones) arises from studies in [3, 11] 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 (respectively, odd-order) butterfly and CCC network can be emulated with no time loss (respectively, with at most a factor of 2 time loss) by the smallest hypercube that is big enough to hold it [11];
   (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 $O(\log \log N)$ [3];
   (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 [12] which presents a computational framework wherein either of the butterfly or de Bruijn networks can emulate the other efficiently. In particular, these emulations yield an algorithm for sorting on the de Bruijn network that rivals the efficiency of the Reif-Valiant [18] 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 [12] to be preferred to the purely structure-oriented emulations of [3, 11], and vice versa?

Remark. The major argument in favor of the structure-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 [12] and [3] 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. 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 de Bruijn networks.

More precisely, an emulation of an $N$-processor architecture $\mathcal{G}$ by an $(M \leq N)$-processor architecture $\mathcal{H}$ is work preserving if $\mathcal{H}$ can emulate any sequence of $T$ computational steps of $\mathcal{G}$ in at most $c(N/M)T$ steps, for some fixed overhead constant $c$. 
The goals presented thus far can be satisfied by a variety of interconnection networks; indeed, the simplest 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 lies in our willingness to suffer a (very) modest amount of structural complication in return for a (very) considerable amount of algorithmic simplicity and efficiency. The following example should clarify our concerns. It is shown in [5] 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. PS networks overcome this deficiency, in that every "reasonable 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 [12], 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 (Theorem 3.7).
- PS networks admit simple and efficient load-balanced work-preserving emulations of both butterfly- and shuffle-oriented networks (Theorems 4.2 and 4.4).
- PS networks admit compact VLSI layouts (Theorem 4.7).

Additionally, the direct-product structure of PS networks guarantees efficient deterministic off-line permutation routing within the network [1]. Finally, the product structure allows PS networks to be used as an ingredient in achieving algorithmic enhancements to parallel architectures that would be prohibitively expensive to achieve in hardware [2]. 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 [12]), PS networks are strictly more

5 The parallel algorithm literature abounds with linear-algebraic and numerical algorithms that conform naturally to the structure of a mesh.
6 We say that the load is balanced in an emulation when each processor of the emulating architecture emulates the same number of processors of the emulated architecture.
Product-shuffle networks

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.

We have thus far used the term "shuffle-oriented networks" to refer ambiguously to the de Bruijn 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 de Bruijn 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 de Bruijn 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 $G$ is specified by a set $V_G$ of nodes and a multisubset $A_G \subseteq V_G \times V_G$ called arcs. One obtains an undirected graph $\tilde{G}$ from a directed graph $G$, by "symmetrizing" the set of arcs: one replaces each arc of $G$ with a pair of mated arcs having opposing directions. We refer to each mated pair of arcs as an edge of the graph $\tilde{G}$.

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

2.1.1. Notation and terminology

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

Note that we allow self-loops and parallel edges.
• The degree of node $v$ of graph $g$ is the number of edges of $g$ incident to (i.e., involving) $v$. The valence of $g$ is the maximum degree of any of its nodes.

2.1.2. The graphs of interest

Let $m,n$ be positive integers.

The order-$n$ de Bruijn 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 $\mathbb{Z}^n$. For all $\beta \in \mathbb{Z}_2$ and $x \in \mathbb{Z}_2^n$:

- in $\mathcal{D}_n$, each node of the form $\beta x$ is connected via a shuffle arc to node $x\beta$.
- Additionally:
  - in $\mathcal{D}_n$, each node of the form $\beta x$ is connected via a shuffle-exchange arc to node $x\beta$;
  - in $\mathcal{S}_n$, each node of the form $x\beta$ is connected via an exchange arc to node $x\beta$.

Figure 1 depicts the (directed version of the) order-3 de Bruijn 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 $V_n = \mathbb{Z}_n \times \mathbb{Z}_2^n$. We call $l$ the level of each node in $\{l\} \times \mathbb{Z}_2^n$. For each $l \in \mathbb{Z}_n$ and each $x = \beta_0 \beta_1 \cdots \beta_n \in \mathbb{Z}_2^n$:

- In both $\mathcal{B}_n$ and $\mathcal{C}_n$, each level-$l$ node $v = (l,x)$ is connected via a straight-edge with node $(l + 1) \text{mod } n, x$.
- Additionally:
  - in $\mathcal{B}_n$, node $v$ is connected via a cross-edge with node 
    $$\langle (l + 1) \text{mod } n, \beta_0 \beta_1 \cdots \beta_{l-1} \beta_{l+1} \cdots \beta_n \rangle;$$
  - in $\mathcal{C}_n$, node $v$ is connected via a level-edge with node 
    $$\langle l, \beta_0 \beta_1 \cdots \beta_{l-1} \beta_{l+1} \cdots \beta_{n-1} \rangle.$$
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 $\mathbb{Z}_2$ as the underlying alphabet of the graphs' node-sets. One can easily define arbitrary base-$d$ versions of the graphs\(^9\) and extend our results with only clerical changes. In [3], 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 $\mathcal{G}$ by another architecture $\mathcal{H}$, we assume that the PEs of $\mathcal{H}$ are sufficiently powerful to emulate the PEs of $\mathcal{G}$ step for step—so no delay is incurred because of computational steps. We restrict attention to emulations that honor a pulsed computation regimen: Architecture $\mathcal{H}$ alternates phases that emulate one computation step of architecture $\mathcal{G}$, with phases that emulate one communication step of architecture $\mathcal{G}$\(^10\). The slowdown incurred by an emulation arises from two sources. First, we allow emulations that require one PE of $\mathcal{H}$ to play the role of several PEs of $\mathcal{G}$; second, architecture $\mathcal{H}$ must emulate its interconnection graph communication steps that are tailored to the (possibly

\(^9\)For illustration, the base-$d$ order-$n$ de Bruijn graph has node-set $Z_d^n$ and edges connecting each node of the form $\delta x$, where $\delta \in Z_d$ and $x \in Z_d^{n-1}$, to all nodes of the form $y \in Z_d^n$.

\(^10\)This regimen of having $\mathcal{H}$ mimic the exact form of the computation by $\mathcal{G}$ motivates our using the term “emulate” rather than “simulate”.
very different) structure of the interconnection graph underlying architecture $G$. 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.

2.2.1. Graph embeddings

An embedding of the graph $G$ in the graph $H$ is specified by:

- a (possibly many-to-one) assignment $\alpha$ of the nodes of $G$ to the nodes of $H$:

$$\alpha : V_G \rightarrow V_H.$$ 

[A PE of $H$ must emulate all of the PEs of $G$ assigned to it via $\alpha$.]

- A routing $\varrho$ of each edge $(u, v)$ of $G$ along a path in $H$ connecting $\alpha(u)$ and $\alpha(v)$.\[11\] [H must emulate each communication along edge $(u, v)$ of $G$ by transmitting the “message” along the path $\varrho(u, v)$.]

2.2.2. Slowdown incurred by an emulation

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

- The load of the embedding is the maximum number of PEs of $H$ assigned to any one PE of $H$:

$$\text{load}(\alpha, \varrho) = \max_{u \in V_H} |\alpha^{-1}(u)|.$$ 

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

- The dilation of the embedding is the maximum amount that the routing $\varrho$ "stretches" any edge of $G$:

$$\text{dilation}(\alpha, \varrho) = \max_{(u, v) \in E_G} \text{Length}(\varrho(u, v)).$$ 

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

- The I/O-expansion of the embedding is the ratio of the valence of $H$ to the valence of $G$:

$$\text{I/O-expansion}(\alpha, \varrho) = \frac{\text{valence}(H)}{\text{valence}(G)}.$$ 

11 A length-$l$ path in $G$ from node $x \in V_H$ to node $y \in V_H$ is a sequence $\pi$ of $l+1$ nodes $x = z_0, z_1, \ldots, z_l = y$ such that, for each $0 \leq i < l$, $(z_i, z_{i+1}) \in E_G$. By abuse of notation, we write "$(z_i, z_{i+1}) \in \pi"."
[I/O-expansion 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 edge-cases of \( \mathcal{G} \) that \( \varphi \) routes over a single edge of \( \mathcal{H} \):

\[
\text{edge-congestion}(\varphi, \mathcal{G}) = \max_{e \in F_\mathcal{G}} |\{e' \in E_\mathcal{H} : e \in \varphi(e')\}|.
\]

[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-expansion seems 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 [3]. 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 [15]; its extension to the current framework appears in [12].

**Proposition 2.1** [12, 15]. 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 based on \( \mathcal{H} \) can emulate \( T \) steps of the architecture based on \( \mathcal{G} \) on a general computation in \( O(L + C + D)T \) steps.

Aside from 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 does, indeed, model the algorithmic situation that we want it to.

Proposition 2.1 points out the importance of balancing the load of an emulation (cf. footnote 6), i.e., of keeping the quantity

\[
\max_{u, v \in V_\mathcal{G}} |\|\alpha^{-1}(u)\| - |\alpha^{-1}(v)\| |
\]

bounded by a constant. Every emulation we present here has balanced load.

Since I/O-expansion 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 do not discuss it further.

Obviously, one strives to make emulations as "efficient" as possible. One can argue (cf. [12]) that one type of efficiency that is very desirable in emulations is work preservation: 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 \textit{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. All of the emulations we present in this paper are work preserving.

### 2.2.3. Two quasi-isometries

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

**Proposition 2.2.** For all \(n\):

(a) One can embed either of the shuffle-exchange graph \(\mathcal{S}_n\) or the de Bruijn graph \(\mathcal{G}_n\) in the other, with load 1, dilation 2, and edge-congestion 2.

(b) (i) One can embed the butterfly graph \(\mathcal{B}_n\) in the CCC graph \(\mathcal{C}_n\), with load 1, dilation 2, and edge-congestion 2.

(ii) [10] The CCC graph \(\mathcal{C}_n\) is a subgraph of the butterfly graph \(\mathcal{B}_n\); hence, one can embed \(\mathcal{C}_n\) in \(\mathcal{B}_n\) with load 1, dilation 1, and edge-congestion 1.

**Proof.** (Sketch). For each of the three embeddings of part (a) and part (b)(i), we employ the \textit{identity} assignment. Ignoring edges that are shared by the respective pairs of graphs in the embeddings (hence use the identity routing), we route

- edge \((\beta x, x\beta)\) of \(\mathcal{G}_n\) along the following length-2 path in \(\mathcal{S}_n\):
  \[
  \beta x \leftrightarrow x\beta \leftrightarrow x\beta;
  \]

- edge \((x\beta, x\beta)\) of \(\mathcal{S}_n\) along the following length-2 path in \(\mathcal{G}_n\):
  \[
  x\beta \leftrightarrow \beta x \leftrightarrow x\beta;
  \]

- edge \((\langle l, x \rangle, \langle (l+1) \mod n, x' \rangle)\) of \(\mathcal{B}_n\) along the following length-2 path in \(\mathcal{C}_n\):
  \[
  \langle l, x \rangle \leftrightarrow \langle l, x' \rangle \leftrightarrow \langle (l+1) \mod n, x' \rangle.
  \]

For the embedding of part (b)(ii), our assignment branches on the \textit{weight} of the string \(x\) in node \(\langle l, x \rangle\) of \(\mathcal{C}_n\), i.e., the number of 1s in \(x\). If \(x\) has \textit{even} weight, then node \(\langle l, x \rangle\) of \(\mathcal{C}_n\) is assigned to node \(\langle l, x \rangle\) of \(\mathcal{B}_n\); if \(x\) has \textit{odd} weight, then node \(\langle l, x \rangle\) of \(\mathcal{C}_n\) is assigned to node \(\langle (l+1) \mod n, x \rangle\) of \(\mathcal{B}_n\). Details are left to the reader. □

### 2.3. Structural characteristics of the graphs of interest

The \textit{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 de Bruijn graphs and hypercubes, although de Bruijn 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 $n$:

<table>
<thead>
<tr>
<th>Graph $\mathcal{D}_n$, $\mathcal{R}<em>n$, $\mathcal{P}</em>{m,n}$</th>
<th>Size $N=2^n$, $n2^n$, $2^{m+n}$</th>
<th>Valence 4, 4, 8</th>
<th>Diameter $\log N$, $2 \log N - 2 \log \log N$, $\log N$</th>
</tr>
</thead>
</table>

**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 $|x| = m$ 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 $|y| = n$ 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 embedded with unit load, edge-congestion, and dilation; an architecture based on PS graphs can emulate an architecture based on any of these subgraphs 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 $(v+1) \pmod{N}$.

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, $\mathcal{D}_n$ satisfies the following stronger property.

**Lemma 3.1** [24]. For all $n$, the de Bruijn graph $\mathcal{D}_n$ is pancyclic; that is, for all $1 \le k \le 2^n$, the $k$-node cycle $\mathcal{C}_k$ is a subgraph of $\mathcal{D}_n$.\(^\text{12}\)

PS graphs share this property, whose computational benefits are exploited in the emulations in [3] and in our Section 4.

**Theorem 3.2.** For all $m$, $n$ except for $m=n=1$, the PS graph $\mathcal{P}_{m,n}$ is pancyclic.

\(^{12}\)Since we allow self-loops and parallel edges, it makes sense to talk about cycles of lengths 1 and 2.
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 $R_c$ is a subgraph of $P_{m,n}$. Our algorithm assumes that the cycles promised by Lemma 3.1 can be produced algorithmically; cf. [24]. (Note that $P_{1,1}$ 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 $R_c$ is a subgraph of $P_{m,n}$, 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 $D_n$ that comprise $P_{m,n}$, together with a length-$b$ cycle from one additional copy of $D_n$ 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 $D_m$, 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 $D_n$, from the $2^m$ copies that comprise $P_{m,n}$; call the ordered copies $D_n^{(0)}, D_n^{(1)}, \ldots, D_n^{(d-1)}$.

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

Case 1: $b = 0$, so $d = a$ and $a > 1$. 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 $D_n^{(i)}$ of a Hamiltonian cycle $C$ of $D_n$. We start by selecting any two independent edges $(x, y)$ and $(u, v)$ of $D_n$,\footnote{"Independence" implies that $\{x, y\} \cap \{u, v\} = \emptyset$.} 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 $D_n^{(i)}$ of $D_n$. Assume that the nodes $x, y, u, v$ lie in (say, for definiteness) clockwise order around the cycle $C$ in $D_n$, 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 $P_{m,n}$.

(1) Trace the cycle $C_0$ in $D_n^{(0)}$ in clockwise order, from node $x_0$ to node $x_0$, leaving out the edge that connects the two nodes.
(2) Trace the following path to complete the cycle:

\[ x_0 \leftrightarrow x_1 \leftrightarrow Q_1 \leftrightarrow v_1 \leftrightarrow u_2 \leftrightarrow Q_2 \leftrightarrow x_2 \leftrightarrow x_3 \leftrightarrow Q_3 \leftrightarrow v_3 \leftrightarrow \cdots \]

\[ q_2 \leftrightarrow Q_{a-1} \leftrightarrow P_{a-1} \leftrightarrow s_{a-1} \leftrightarrow P_{a-1} \leftrightarrow l_{a-1} \leftrightarrow \cdots \]

\[ u_3 \leftrightarrow P_3 \leftrightarrow P_2 \leftrightarrow u_2 \leftrightarrow u_1 \leftrightarrow P_1 \leftrightarrow y_1 \leftrightarrow y_0 \]

where

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

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 Fig. 3.

Case 2: \( 0 < b < 2^n \), so \( d = a + 1 \) and \( 0 < a < 2^m \). 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 \( \mathcal{D}(a) \) 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 \( x \rightarrow y \rightarrow z \) 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 \((y, z)\).

**Proof.** The Claim is true by inspection when \( n = 2 \); when \( n \geq 2 \), it follows from standard facts about de Bruijn 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 \( \pi \) in \( \mathcal{G} \) whose removal does not disconnect \( \mathcal{G} \), one can construct an Eulerian cycle in \( \mathcal{G} \) which contains \( \pi \).

**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 0 or node 1.

Because of Fact 1, a 3-node path

\[ x \rightarrow y \rightarrow z \]

in \( \mathcal{D}_n \) results from a 3-edge path

\[ W \rightarrow X \rightarrow Y \rightarrow Z \]

in \( \mathcal{D}_{n-1} \). Since at most two of these edges can both be incident to either node 0 or node 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

\[ W \rightarrow X \rightarrow Y \]

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 \((y, z)\).

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 \( \mathcal{D}(a) \), 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-1 path

\[ r_{a-1} \leftrightarrow s_{a-1} \]

with the length-\((b+1)\) 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 length-\((b - 1)\) path within cycle \( B \) that connects nodes \( r_s \) and \( s_s \) in \( \mathcal{G}^{(a)} \) once edge \((r_a, s_a)\) is removed.

**Case 2(b):** \( b = 2 \). We proceed exactly as in Case 1, except that we alter the trajectory of the length-\(a2^n\) cycle of that case by replacing the length-1 path

\[
\begin{align*}
    r_{a-1} & \leftrightarrow s_{a-1} \\
\end{align*}
\]

with the length-3 path

\[
\begin{align*}
    r_{a-1} & \leftrightarrow r_a \leftrightarrow s_a \leftrightarrow s_{a-1} \\
\end{align*}
\]

where \( r, s \) are as in Case 1.

**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 copy \( \mathcal{G}^{(a-1)} \) of \( \mathcal{D}_n \). At that point we replace the length-3 path

\[
\begin{align*}
    q_{a-1} & \leftrightarrow r_{a-1} \leftrightarrow s_{a-1} \leftrightarrow t_{a-1} \\
\end{align*}
\]

from Case 1 with a length-4 path of one of the forms

\[
\begin{align*}
    q_{a-1} & \leftrightarrow s_{a-1} \leftrightarrow s_a \leftrightarrow t_a \leftrightarrow t_{a-1} \\
\end{align*}
\]

or

\[
\begin{align*}
    q_{a-1} & \leftrightarrow q_a \leftrightarrow s_a \leftrightarrow s_{a-1} \leftrightarrow t_{a-1} \\
\end{align*}
\]

within copies \( \mathcal{G}^{(a-1)} \) and \( \mathcal{H}^{(a)} \) of \( \mathcal{D}_n \). One verifies readily that one of these paths exists.

**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 has length \( e + 1 \)), 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

\[
\begin{align*}
    0 \leftrightarrow \vec{0} \leftrightarrow \vec{0}1, \\
\end{align*}
\]

the elision of node \( \vec{0} \) does not cut our cycle: it just shortens it, as desired.

This case analysis completes the proof. \( \square \)

Very little of the proof of Theorem 3.2 depends on properties that are peculiar to de Bruijn graphs. In fact, F. Annexstein and M. Baumslag [personal communication] have observed that an altered version of the proof will establish the following.

**Proposition 3.3.** 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 \( 2 \leq l \leq |\mathcal{G}| \), \( \mathcal{G} \) 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 $\mathcal{R}_m \times \mathcal{R}_n$.

One corollary of the main lower bound in [5] is that no butterfly- or shuffle-oriented graph $\mathcal{B}$ can emulate meshes with only constant slowdown (using our notion of emulation); it follows, of course, that $\mathcal{B}$ 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 Lemma 3.1.

**Corollary 3.4.** 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 $2^{n+1} - 1$ nodes comprise the set $\bigcup_{k=0}^{n} Z_2^k$ of binary strings of length $\leq n$ and whose edges connect each node $x$ of length $< n$ with nodes $x0$ and $x1$.

Complete binary trees are very useful computational structures, most obviously for broadcasting, but also for emulations [5]. Thus, the following obvious result points out one of the most useful properties of de Bruijn graphs; cf. [20].

**Lemma 3.5.** For all $n$, the de Bruijn graph $\mathcal{B}_n$ contains the complete binary tree $\mathcal{T}_{n-1}$ as a subgraph, rooted at node $\langle 01 \rangle$.

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

**Theorem 3.6.** For all $m, n$, the PS graph $\mathcal{P}_{m,n}$ contains the complete binary tree $\mathcal{T}_{m+n-2}$ as a subgraph.

**Proof.** We find an instance of $\mathcal{T}_{m+n-2}$ rooted at node $v_0 = \langle 01, 01 \rangle$ of $\mathcal{P}_{m,n}$, as follows. We first invoke Lemma 3.5 to find a copy of $\mathcal{T}_{n-1}$ in "copy $\langle 01 \rangle"$ of $\mathcal{B}_n$, rooted at node $v_0$ and having $2^{n-1}$ leaves of the form $\langle 01, x \rangle$ for some $x \in Z_2^{n-1}$. We then invoke Lemma 3.5 once for each "copy" of $\mathcal{B}_m$ that is "connected" to one of these leaves, to find a copy of $\mathcal{T}_{m-1}$ rooted at each of these leaves. □

To place Theorem 3.6 in perspective, the efficient embedding of complete binary trees in butterfly graphs presented in [5] 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 $\mathcal{M}_T_{m,n}$ ($m$ 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 $T_{m}(n)$ along each column, using the column-nodes as leaves of the tree,
- erecting a copy of the complete binary tree $T_{n}(m)$ along each row, using the row-nodes as leaves of the tree.

Parallel architectures based on meshes of trees have been shown to be quite powerful computationally [14]. One can prove, using Lemma 3.5, that PS graphs contain at least moderate-size meshes of trees as subgraphs.

**Theorem 3.1.** For all $m$, $n$, and for all $p$, $q$ such that $2^{p} \leq m/2$, $2^{q} \leq n/2$, the PS graph $P_{m,n}$ contains the mesh of trees $M_{2^{p},2^{q}}$ as a subgraph.

**Proof.** Let us denote by $x_{i}$, $1 \leq i \leq 2^{p}$, the $i$th leaf of $T_{p}$ and by $y_{j}$, $1 \leq j \leq 2^{q}$, the $j$th leaf of $T_{q}$. By Lemma 3.5, $P_{m,n}$ contains the product graph $T_{p} \times T_{q}$ as a subgraph. Consequently, $P_{m,n}$ contains as a subgraph every tree of the form $\{x_{i}\} \times T_{q}$, as well as every tree of the form $T_{p} \times \{y_{j}\}$. The union of all these subgraphs is $M_{2^{p},2^{q}}$. $\blacksquare$

4. PS vs. butterfly vs. de Bruijn 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 de Bruijn 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 de Bruijn 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 de Bruijn networks

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

**Lemma 4.1.** For all $m$ and $n$, one can embed the de Bruijn graph $D_{m+n}$ in the PS graph $P_{m,n}$, with load 1, dilation 2, and edge-congestion 2.

**Proof.** Each node $x$ of $D_{m+n}$ 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:

$$\alpha(x) = (x)_{m}, [x]_{n}$$
for all $x \in \mathbb{Z}_2^{m+n}$. Each edge of $\mathcal{D}_{m+n}$ has the form $(\beta w, w\delta)$ for $\beta \in \mathbb{Z}_2$, $w \in \mathbb{Z}_2^{m+n-1}$, and $\delta \in \{\beta, \beta\}$. Let us write each $w \in \mathbb{Z}_2^{m+n-1}$ in the form $w = uyv$, with $u \in \mathbb{Z}_2^{m-1}$, $y \in \mathbb{Z}_2$, and $v \in \mathbb{Z}_2^{n-1}$. Then the routing function $\rho$ of the embedding realizes the edge

$$(\beta w, w\delta) = (\beta uyv, uyv\delta)$$

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

$$a(\beta w) = a(\beta uyv) = \langle \beta u, vy \rangle \leftrightarrow \langle uy, yv \rangle \leftrightarrow \langle uy, v\delta \rangle = a(uyv\delta) = a(w\delta).$$

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 $\delta$, while the second edge identifies the edge being emulated, up to the identity of $\beta$. $\square$

**Theorem 4.2.** For all $n$ and all $p$ and $q$ with $p + q \leq n$, one can embed the de Bruijn graph $\mathcal{D}_n$ in the PS graph $\mathcal{P}_{p,q}$, with load $2^{n-p-q}$, dilation 4, and edge-congestion $2^{n-p-q}$. This leads to a work-preserving emulation of $\mathcal{D}_n$ by $\mathcal{P}_{p,q}$.

**Proof.** First, 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.

Next, we 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 $(x,y)$ 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. $\square$

### 4.2. PS networks emulating butterfly networks

**Lemma 4.3** [3]. For all $n$, one can embed the butterfly graph $\mathcal{B}_n$ in the PS graph $\mathcal{P}_{(n),n}$, with load 1, dilation 2, and edge-congestion 2.

**Proof (Sketch).** We sketch the proof from [3], which has recently been put in a much more general context in [4]. By the pancyclicity of de Bruijn graphs (Lemma 3.1), it suffices to embed $\mathcal{B}_n$ in the product graph $\mathcal{P}_{n} \times \mathcal{D}_n$, with unit load, dilation 2, and edge-congestion 2.

We label the nodes of $\mathcal{B}_n$ with strings from $\mathbb{Z}_2^n$ via the following inductive procedure that is implicit in [3]; cf. Fig. 4.

1. Label node $(0, \overline{0})$ of $\mathcal{B}_1$ with the string $\overline{0}$.
2. If level-$l$ node $v$ ($l \in \mathbb{Z}_n$) is labeled with string $L(v)$, then label the straight-edge (respectively, the cross-edge) neighbor of node $v$ on level $(l+1)\mod n$ with the shuffle (respectively, the shuffle-exchange) of $L(v)$.
Now, isolate any two consecutive levels of the labeled $B_n$, together with the $2^{n+1}$ edges that connect the levels; cf. Fig. 5. Produce the $2^n$-node graph $G_n$ from the isolated levels by identifying like-labeled nodes and eliminating self-loops. Our labeling procedure guarantees that:

**Claim.** For any two consecutive levels of $B_n$, the graph $G_n$ is isomorphic to $D_n$.

The result is now direct: To embed $B_n$ in $B_n \times D_n$:
- Assign level-$l$ node $v$ of $B_n$ to node $L(v)$ of copy $l$ of $D_n$, where $L(v) \in Z_2^n$ is the label assigned to node $v$ by the indicated procedure.
- Route edge $\langle (l,x),(l',y) \rangle$ of $B_n$ within $B_n \times D_n$ via the length-2 path:
  \[ \langle l, L(\langle l,x \rangle) \rangle \leftrightarrow \langle l, L(\langle l',y \rangle) \rangle \leftrightarrow \langle l', L(\langle l',y \rangle) \rangle. \]

Thus, we first route within a copy of $D_n$ and then between copies.

Fig. 5. Two consecutive levels of $B(3)$ with the shuffle-oriented labelling; “columns” are permuted to help visualize the identification.
The described embedding clearly has unit load and dilation 2. The edge-congestion of the embedding is only 2 because each edge connecting levels \( l \) and \( l' \) in \( B_n \) is routed first within the level-\( l \) copy of \( D_n \) and only then between the level-\( l \) and level-\( l' \) copies of \( D_n \); hence, the only edges that engender edge-congestion are pairs of straight-edges and cross-edges in \( B_n \) that share an endpoint. □

**Theorem 4.4.** For all \( n \) and all \( p \) and \( q \) with \( q \leq \min(n, 2^p) \), one can embed the butterfly graph \( B_n \) in the PS graph \( P_{p,q} \) with load \( L = 2^{n-q} \cdot \max(1, 2\lceil n2^{-p} - 1 \rceil) \), edge-congestion \( 4L \), and dilation 2. This leads to a work-preserving emulation of \( B_n \) by \( P_{p,q} \).

**Proof.** Our embedding proceeds in three stages.

Stage 1. We embed \( B_n \) in a graph \( G_{n,q} \) which is defined implicitly via the surjective assignment and routing mappings \((\alpha_1, \rho_1)\) defined as follows. For each node \( \langle l, x \rangle \) of \( B_n \),

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

The mapping \( \rho_1 \) routes each edge of \( B_n \) naively, via an edge of \( G_{n,q} \). The embedding thus defined has load \( L_1 = 2^{n-q} \), edge-congestion \( 2L_1 \), and unit dilation. In order to verify these costs and to see what the next stage of our composite embedding needs to accomplish, consider what the graph \( G_{n,q} \) looks like. From one vantage point, one obtains \( G_{n,q} \) from \( B_n \) by removing all cross-edges except those having both endpoints in levels \( 0, 1, \ldots, q \); from another vantage point, one constructs \( G_{n,q} \) from the nodes of \( B_n \) by inserting edges that:

- make the induced subgraph of \( G_{n,q} \) on levels \( 0, 1, \ldots, q \) (isomorphic to) the graph obtained from \( G_{q+1} \) by removing the edges connecting level \( q \) to level \( 0 \);
- make the induced subgraph of \( G_{n,q} \) on levels \( q, q + 1, \ldots, n - 1, 0 \) (isomorphic to) \( 2^q \) node-disjoint length-(\( n-q \)) paths, each 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 \( x \in \mathbb{Z}_2^q \).

The only subtlety involving the costs of this embedding is that the edges within levels \( q, q + 1, \ldots, n - 1, 0 \) of \( G_{n,q} \) are congested twice as much as the edges within levels \( 0, 1, \ldots, q \), because within the higher-numbered levels, cross-edges share routing paths with straight-edges.

Stage 2. Now we compare the values of \( n \) and \( p \). If \( n \leq 2^p \), then we do nothing. If \( n > 2^p \), then we embed \( G_{n,q} \) in \( G_{2^p,q} \), as follows.

1. “Fold” the length-(\( n - 2^p \)) “dangling paths” that start at level \( 2^p \) of \( G_{n,q} \) and proceed to level \( 0 \), into the top-\( 2^p \) levels of the graph.
2. Eliminate all levels of \( G_{n,q} \) from level \( 2^p + 1 \) through level \( n - 1 \).
3. Identify levels \( 0 \) and \( 2^p \) of the resulting “pruned” graph.

\(^{14} \) Recall that \((x)_q\) is the length-\( q \) prefix of the string \( x \).
The "folding" of "dangling paths" is accomplished via the assignment function $a_2$ defined by:

$$a_2((2^p + k, x)) = a_2((n - k - 1, x)) = (k \mod 2^p, x)$$

for all $x \in Z_q$ and all

$$0 \leq k \leq \frac{n - 2^p - ((n - 2^p) \mod 2)}{2}.$$

Once again, we employ the naive (edge-to-edge) routing to complete the specification of Stage 2 of our embedding. One verifies easily that this embedding has load $L_2 = \max(1, 2^{n - 2^p - 1})$, edge-congestion $2L_2$, and unit dilation.

Stage 3. Finally, we embed the host graph $\mathcal{G}_{m,q}$ from Stage 2, where $m = \min(n, 2^p)$, into $\mathcal{P}_{p,q}$. This embedding can be specified indirectly, by invoking the proof (rather than statement) of Lemma 4.3. In that proof, $\mathcal{B}_n$ is embedded in $\mathcal{R}_n \times \mathcal{D}_n$ with unit load and with dilation and edge-congestion 2. Precisely the same reasoning embeds $\mathcal{G}_{m,q}$ in $\mathcal{R}_m \times \mathcal{D}_q$ with the same costs. Our embedding of $\mathcal{G}_{m,q}$ in $\mathcal{P}_{p,q}$ is completed by noting that $\mathcal{R}_m \times \mathcal{D}_q$ is a subgraph of $\mathcal{P}_{p,q}$ (Lemma 3.1).

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 de Bruijn graphs, in the sense of the following result. Note how much stronger the result is for butterfly graphs than for de Bruijn graphs, both in terms of quantification and dilation.

**Theorem 4.5.** (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 de Bruijn graph $\mathcal{D}_{m+n}$ must have dilation $\Omega(\log \min(m, n))$.

**Proof.** Let $M = 2^{\min(m, n)}$. By Corollary 3.4, $\mathcal{P}_{m,n}$ contains the $M \times M$ mesh $\mathcal{M}_{M,M}$ as a subgraph. It is proved in [5] 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 de Bruijn graph must have dilation $\Omega(\log \log M)$. □

The lower bounds of Theorem 4.5 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 de Bruijn 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 layouts of either of the other two graphs. We refer the reader to [7,23] for background on the formal framework and techniques of analysis for VLSI layouts.

We begin with the layout requirements of de Bruijn and butterfly graphs.

**Theorem 4.6.** (a) [13] The de Bruijn graph $D_{m+n}$ 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) [23] Any VLSI layout of the de Bruijn graph $D_{m+n}$ consumes area

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

(c) [23] The area-minimal VLSI layouts of the butterfly graph $B_n$, 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 4.7.** The PS graph $P_{m,n}$ admits a VLSI layout of area

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

**Proof.** We use the layouts guaranteed by Theorem 4.6(a) and by the following lemma to obtain a VLSI layout of $P_{m,n}$ with the advertised area.

**Lemma 4.8** [7]. The de Bruijn graph $D_n$ 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 $D_n$ from Theorem 4.6(a), aligned so that, for each node $v$ of
Product-shuffle networks

5. Concluding remarks

Permutation routing. Proposition 2.3 indicates that PS networks can match the efficiency of de Bruijn 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 [12], 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 be an interesting challenge to adduce underlying principles that explain the answers (along the lines of the algebraic development in [3]).

Acknowledgement

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

References