Using Emulations to Construct High-Performance Virtual Parallel Architectures

Bojana Obrenić, Martin C. Herbordt,
Arnold L. Rosenberg, Charles C. Weems,
Fred S. Annexstein, and Marc Baumslag
Computer and Information Science Department
University of Massachusetts

COINS Technical Report 91-40
May 1991

Using Emulations to Construct High-Performance Virtual Parallel Architectures[1]

Bojana Obrenić
Martin C. Herbordt
Arnold L. Rosenberg
Charles C. Weems
Fred S. Annexstein
Marc Baumslag

Department of Computer and Information Science
University of Massachusetts
Amherst, Massachusetts

Acknowledgment of Support: The research of F. S. Annexstein and M. Baumslag was supported in part by NSF Grant CCR-88-12567. The research of B. Obrenić, and A. L. Rosenberg was supported in part by NSF Grants CCR-88-12567 and CCR-90-13184. The research of M. C. Herbordt and C. C. Weems was supported in part by DARPA Contracts DACA76-86-C-0015 and DACA76-89-0016, monitored by the U.S. Army Engineer Topographic Laboratory.

Present Affiliations and Addresses:
F. S. Annexstein is with the Department of Computer Science, University of Cincinatti, Cincinatti, OH 45221.
M. Baumslag is with the Department of Computer Science, City University of New York, New York, NY 10036.
M. C. Herbordt, B. Obrenić, A. L. Rosenberg, and *C. C. Weems* are with the Department of Computer and Information Science, University of Massachusetts, Amherst, MA 01003.

[1]A preliminary version of this paper was presented at the *3rd IEEE Symp. on Frontiers of Massively Parallel Computation*, Oct. 8-10, 1990, in College Park, Md.

Abstract

We use techniques and results from the theory of network emulations to endow a processor array with (apparent) architectural enhancements solely by algorithmic means, with no enhancements to the array's hardware. The specific enhancement we study endows an N-processor bit-serial processor array $\mathcal{A}$ with a "meta-instruction" GAUGE k, which (logically) reconfigures $\mathcal{A}$ into an N/k-processor virtual machine $\mathcal{B}_k$ that has

- a datapath and memory bus whose (apparent) width is k bits, as opposed to $\mathcal{A}$'s 1-bit width;
- an instruction set that operates on k-bit words, in contrast to $\mathcal{A}$'s instruction set, which operates on 1-bit words.

The emulation techniques we present can be implemented efficiently even if the bit-serial array operates in SIMD mode, with very restricted masking capabilities. We describe at an algorithmic level how to implement our technique—including datapath conversion ("corner-turning") and the creation of the bit-parallel instruction sets—on SIMD bit-serial processor arrays of any network topology. We describe detailed implementations of our technique for SIMD bit-serial processor arrays based on the hypercube, the de Bruijn network, and the mesh with reconfigurable buses.

1 Overview

1.1 Motivation

Recent years have seen significant advances in the theory of network emulations; cf. [1], [4], [5], [9], [13], [18]. We now have a battery of nontrivial techniques that allow one interconnection network to emulate another efficiently, even when the emulated and emulating networks differ dramatically in size and topology. Particularly significant for our concerns here is the fact that emulation techniques in all of the cited papers, save [13], operate on an instruction-by-instruction basis, thereby enabling emulations even within strict algorithmic regimens. We implement some of these emulations within the regimen of SIMD operation with very restricted masking.

One potential application that motivated the development of the techniques in the cited sources is that of allowing a processor array to change the (apparent) size and/or topology of its underlying interconnection network repeatedly, in the course of a computation, whenever such a change would enhance algorithmic efficiency. This facility can be viewed as an algorithmic analogue of the philosophy underlying hardware-reconfigurable architectures such as the Blue CHiP [22].

The just-noted parallel between the (algorithmic) study of emulations and the (architectural) study of hardware-reconfigurable architectures has inspired the research we report on here.

This paper is devoted to developing, in considerable detail—down to the level of specifying pseudo-code for all tasks—an algorithmic methodology that allows a processor array to exhibit *multigauge behavior*, i.e., to change the (apparent) width of its datapath and memory bus.

We view the contributions of this research as three in number.

- We introduce algorithmic techniques that achieve a certain type of architectural enhancement.
- We demonstrate, by achieving this algorithmic goal within a very restricted architectural environment—bit-serial processor arrays with very simple processors, operating in a strict SIMD regimen with single-bit masking—that this restricted architectural environment has unexpected computational robustness.
- We demonstrate that our software enhancements can be achieved cleanly, by constructing a virtual machine on top of the native instruction set and below the application level.

1.2 Multigauge Processor Arrays

A processor array is said to exhibit *multigauge behavior* if it can dynamically change its *gauge size*, i.e., the (apparent) width of its datapath and memory bus. We present a strategy for achieving multigauge behavior in processor arrays, solely via algorithmic emulations, with no hardware enhancements. In order to emphasize the versatility of our strategy and to choose a testbed where our techniques are most likely to be of significance, we assume henceforth that the processor arrays we endow with multigauge behavior are *bit-serial* (i.e., have unit gauge size) and that they operate in a *SIMD* regimen. Figure 1 depicts schematically how a gauge-1 (i.e., bit-serial) processor would access (the bits of) a 3-bit word $b_0b_1b_2$, in contrast to how a gauge-3 processor would access the same word.

Our choice of multigauge behavior as a goal was influenced by a number of studies in the literature that demonstrate its value [23], [17]. In particular, it is shown in [17] that multigauge behavior can improve computational performance in certain applications, by allowing a single machine to match its gauge size dynamically to the natural width of the data type it is operating on.

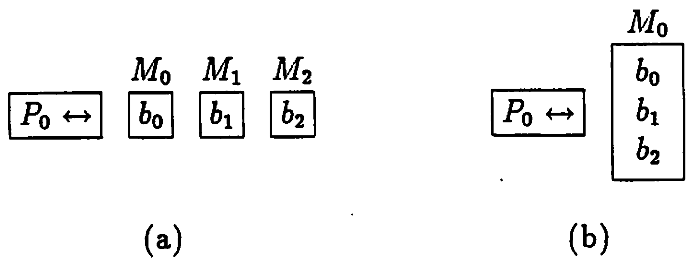


Figure 1: (a) *A gauge-1 (bit-serial) processor accessing the 3-bit word* $b_0b_1b_2$. (b) *A gauge-3 processor accessing the same word.*

Techniques for achieving multigauge behavior by hardware enhancements have been studied in [17], [24], [12], [2]. These sources comment on three problems associated with hardware-enabled multigauging: the costs in hardware, the difficulty of enabling a large repertoire of gauge sizes, and the extent to which the required early commitment to hardware multigauging interferes with subsequent design decisions. Algorithm-enabled multigauging encounters none of these problems, offering great flexibility at modest operating cost; indeed, emulation-enabled multigauge behavior provides—at least in principle—virtually *any conceivable gauge size*[2] at an operational cost that is a slowly growing function of the present gauge size (cf. Section 7).

We specify the change of gauge size to a level of detail that renders the change an ordinary computation step, with the selection of gauge size as easy as parameter passing. In fact, one can view the programming environment our approach builds on the host N-processor architecture, as having a software implemented *meta-instruction*

$$\mathsf{GAUGE}\ k.$$

This meta-instruction has a two-pronged effect on the host $\mathcal{A}$ by reconfiguring $\mathcal{A}$ to an N/k-processor virtual machine $\mathcal{B}_k$. First, it changes the (apparent) width of the datapath and memory bus from 1 bit to k bits. Second, it (apparently) transforms the instruction set from one that operates on 1-bit words to one that operates on k-bit words. This approach stresses and amplifies the known [23] *conceptual* advantage of multigauge machines. Such machines are both fine grain and coarse grain; indeed, in our solution, they are *multi-grain*. They exhibit this multi-granularity with respect to a single memory, even dynamically with respect to the same computation. Our approach allows an *instance* of a bit-serial processor array $\mathcal{A}$ to act as though it were a *family* $\{\mathcal{B}_k\}$ of bit-parallel processor arrays that differ in gauge size, but whose instruction sets enjoy *uniform* operational semantics across all

[2]In practice, there are implementational and operational advantages when the repertoire of available gauge sizes is restricted to the set of integer divisors of the number of processors.

gauge sizes. And, this *flexibility* is accompanied by operational performance that is superior to that achieved by having the physical bit-serial host $\mathcal{A}$ perform k-bit operations in the straightforward bit-serial way.

The remainder of the paper is organized as follows. Section 2 describes details of the architectural framework of our study. Section 3 presents our emulation strategy in a topology-independent fashion. The next three sections describe and analyze our emulations on three different genres of array: the hypercube network [25], the deBruijn network [20] (or, equivalently, the shuffle-exchange network [26] [21]), and the extended coterie network, which is a mesh enhanced with a highly reconfigurable bus that is an abstract version of the University of Massachusetts CAAPP architecture [28]. The final section summarizes the accomplishments of the paper.

2 Bit-Serial and Bit-Parallel Architectures

The architectures we study are *processor arrays*; i.e., each comprises a set of identical *processing elements* (PEs), each having its private *memory module*, interconnected via a network that allows them to intercommunicate, by passing data items. As is customary, we adopt an abstract view of a processor array in which the array is represented as an *undirected graph*; the *nodes* of the graph represent the array's PEs (with their associated memory modules) while the *arcs* of the graph represent the array's point-to-point inter-PE communication links.

We make two substantive assumptions about the way the processor arrays we study operate: First, we assume that the arrays observe a *pulsed* mode of computation in which *computation* steps and *communication* steps may alternate at each pulse. Second, we assume that the arrays compute within a *SIMD* regimen, so that all PEs execute the same instruction at each pulse. Both communication and computation steps are initiated by the central controller of the SIMD machine. Note that the SIMD regimen complicates our implementation of multigauge behavior, in that we must specify the behavior of the individual PEs solely in terms of the behavior of the SIMD controller. To elaborate on what happens in these steps, consider a SIMD array $\mathcal{B}_k$ with gauge size k. During a computation step, each PE of $\mathcal{B}_k$ refers to its own memory for a k-bit operand and/or performs an arithmetic or logical operation on k-bit operand(s), all PEs executing the same instruction. During a communication step, each PE of $\mathcal{B}_k$ receives/sends a single k-bit word through one of its I/O ports from/to a PE that is adjacent in the network-graph.

A processor array is said to be

- *bit-serial* if it has gauge size $k = 1$

- *bit-parallel* if it has gauge size $k > 1$
- *multigauge* if it can assume more than one gauge size during the course of a computation.

The instruction sets and interconnection topologies of real processor arrays vary rather widely. We now summarize the basic characteristics that we assume of both the physical and logical PEs and networks we consider. Our ability to describe both physical and logical architectures simultaneously derives from our enforcing uniform operational semantics on all processor arrays; hence, we merely view the physical host array $\mathcal{A}$ as the $k = 1$ instance of the family of logical arrays $\{\mathcal{B}_k\}$.

Instruction Repertoire. Each PE in a processor array of gauge size k is capable of the following operations on words of width k:

- algebraic addition
- logical operations[3] (both bitwise and accumulative)
- numerical comparison
- circular shifts

We assume that each bit-serial PE has the capabilities of a full adder.

Memory. Since all PEs in a processor array are identical, their memory modules have the same capacity (in bits). We implement the meta-instruction GAUGE k

- only for values of k that do not exceed the common capacity of the host array's memory modules
- in a manner that ensures the uniform capacity of the memory modules of the resulting bit-parallel array's PEs.

Implementing the emulations that achieve multigauge behavior may tie up some small (constant) amount of memory within each bit-parallel PE; therefore, the cumulative memory in a bit-parallel array $\mathcal{B}_k$ may be slightly less than in the physical host array $\mathcal{A}$.

Our algorithms and their implementations can accommodate array-memory that is addressed either directly or indirectly. *Directly* addressed memory is characterized by an address field in each instruction, which names the location being accessed; *indirectly* addressed

[3]We allow both *bitwise* operations that map $\langle x_1, \ldots, x_n \rangle$ and $\langle y_1, \ldots, y_n \rangle$ to $\langle x_1 \circ y_1, \ldots, x_n \circ y_n \rangle$, and *accumulative* operations that map $\langle x_1, x_2, \ldots, x_n \rangle$ to $x_1 \circ x_2 \circ \cdots \circ x_n$.

memory uses the contents of a designated address register associated with each PE as the address of the accessed location. We consider also one type of *implicitly* addressed memory, as epitomized by the presence of a *shift register* in each PE. Throughout our study, the addressing mode(s) of the bit-serial host array are inherited by any emulated bit-parallel guest array.

As part of its memory, each PE has a *processor index register* (PIR) which contains the PE's unique processor index. Bits of the PIRs are addressed like memory bits, except that they are accessed in *read-only* mode.

Data Transfer. Data transfer takes place between PEs that are adjacent in the communication network. Each PE has an *input* port and an *output* port for each incident arc. The ports have the same width as, and are addressed and accessed very much like memory locations. A communication instruction transfers the contents of an output port into the input port of the PE at the other end of the incident arc, at the end of the communication step. No arc can have both of its output ports active at the same time; only one output port of each PE can be active at one communication step.

Data I/O. We do *not* model or emulate the I/O subsystems of logical arrays, relying instead on the bit-serial host array to load and unload the physical memory. Specifically, during physical I/O, each bit-parallel data item is handled by one bit-serial PE.

The major consequence of this decision is that every change in gauge size must begin and end with the operation of *datapath conversion*. This operation transforms the contents of all memory modules to provide the functionality of the *corner-turning* circuitry of machines that accomplish multigauging via hardware support. When one goes from bit-serial to bit-parallel processing, the operation builds the words for the bit-parallel PEs. When one returns from bit-parallel to bit-serial processing, the operation returns data to its bit-serial format, for further processing or for output.

Control. At each step, the SIMD controller issues the same instruction to every PE. The instructions specify the operations to be performed and the address(es) of the operand(s). Each PE has an *activity bit register* (AB). This register can be addressed and accessed like a memory location; its role in a PE is to *enable* or *disable* the PE for the execution of the instruction.

We have now specified enough details of the architecture of the bit-serial host array $\mathcal{A}$ to describe our emulation algorithms in detail; we have now specified enough details of the architectures of the bit-parallel guest arrays $\mathcal{B}_k$ to know what needs to be emulated.

3 The Emulation Strategy

3.1 The High-Level Strategy

Henceforth, for mnemonic emphasis, we denote the given physical ***host*** bit-serial SIMD processor array $\mathcal{A}$ by $\mathcal{H}$, and we denote its underlying graph by H.[4]

Say that we want to have the host array $\mathcal{H}$ behave as though it were one of the arrays $\mathcal{B}_k$, for definiteness, say the gauge-k bit-parallel *guest* SIMD array $\mathcal{G}$ (whose underlying graph is denoted G). Our strategy for achieving this objective is to *reconfigure* the network H underlying $\mathcal{H}$ *logically* into disjoint isomorphic *aggregates* of k PEs each. Each aggregate is viewed as a separate k-PE processor array which acts as a logical *macro-PE* of the width-k array $\mathcal{G}$. To emphasize the fact that $\mathcal{G}$ and its underlying graph G are *logical* entities, we speak of them as a *macro-array* and a *macro-graph*. We support the logical reconfiguration of $\mathcal{H}$ into a macro-array of macro-PEs by a set of procedures, comprising *macro-instructions*, to be executed by the SIMD controller of $\mathcal{H}$. Computation macro-instructions effect k-bit operations within each aggregate, while communication macro-instructions transfer data among aggregates in a way that honors the structure of the macro-graph G. We strive for a strong semantic correspondence between each basic instruction executed on the host array $\mathcal{H}$ and its matching macro-instruction interpreted over the macro-graph and the new gauge size, and we strive to implement the macro-instruction capabilities of $\mathcal{G}$ efficiently on top of the physical instruction set.

The scenario just depicted presents certain challenges relating to the question of how to partition the host array $\mathcal{H}$.

- The partition must allow the SIMD controller to "address" all macro-PEs efficiently.
- The partition must allow the PEs within each macro-PE to "cooperate" efficiently.

We address these challenges by exploiting the structure of the graph H in determining the partition that converts H logically into a "fat" version of the macro-graph G.

3.2 The Detailed Strategy

The strategy whose implementation achieves the efficient emulations we seek has three major ingredients which we describe and discuss in this subsection.

[4]Throughout the paper, we use uppercase script letters to denote processor arrays and the corresponding uppercase italic letters to denote the graphs underlying the arrays.

Ingredient 1: Emulating Direct-Product Arrays

The first ingredient we discuss contributes to our solution by efficiently ensuring structural uniformity of all graphs.

The *direct product* of arrays $\mathcal{A}$ and $\mathcal{B}$, denoted $\mathcal{A} \times \mathcal{B}$ is defined as follows. The nodes of $\mathcal{A} \times \mathcal{B}$ are all ordered pairs (u, v), where u is a node of $\mathcal{A}$ and v is a node of $\mathcal{B}$. There is an arc in $\mathcal{A} \times \mathcal{B}$ connecting nodes (u, v) and (u', v') just when either

$$u = u', \text{ and } v \text{ is connected to } v' \text{ in } \mathcal{B}$$

or

$$v = v', \text{ and } u \text{ is connected to } u' \text{ in } \mathcal{A}.$$

We apply the notion of direct-product array to the problem at hand, in the following way. Say we are presented with an N-PE host array $\mathcal{H}$. For each integer k for which we want to implement the instruction GAUGE k on array $\mathcal{H}$, we choose a single k-PE array $\mathcal{K}$ to play the role of our aggregate-array, and we choose a specific $\lfloor N/k \rfloor$-PE array $\mathcal{G}$ to play the role of our macro-array.[5] We then begin our implementation by having array $\mathcal{H}$ emulate the direct-product array $\mathcal{K} \times \mathcal{G}$:

```
H emulates  K   ×   G
            |   |   |   ----  acts as the macro-network
            |   |   --  ----  yields identical copies of "factors"
            |   --  --  ----  each copy acts as the k-bit macro-PE
```

This portion of our strategy automatically solves several aspects of the implementation problem. Note first that $\mathcal{K} \times \mathcal{G}$ can be viewed as array $\mathcal{G}$ with "fat" nodes, each of which is a copy of array $\mathcal{K}$. Because of the consistency of this world view, all macro-PEs in a direct-product solution have the same node-set. *This completely solves the "addressing" problem for the SIMD macro-controller* (i.e., the SIMD controller of $\mathcal{H}$ acting as a controller for the macro-array $\mathcal{G}$); specifically, the controller uniformly addresses the ith PE of the jth macro-PE as PE (i, j) of the (emulated) direct product array. Another consequence of the world view is that, if copies $\mathcal{K}_1$ and $\mathcal{K}_2$ of macro-PE $\mathcal{K}$ are "adjacent" to one another within the macro-array $\mathcal{G}$, then every node of $\mathcal{K}_1$ is adjacent to its like-named analogue in $\mathcal{K}_2$ within the emulated array $\mathcal{K} \times \mathcal{G}$. *This renders trivial the problem of implementing data transfers within the (emulated) direct-product array;* specifically, a word-transfer between macro-PEs

[5]In all of our examples, we consider only values of k that divide N. While these values of k admit more efficient solutions, there is no conceptual mandate to insist on this restriction.

h and j is effected via a parallel bit-transfer between PEs (i,h) and (i,j) of the (emulated) direct product array.

We lend more detail to this description.

We let each copy of $\mathcal{K}$ perform the tasks of a macro-PE operating on k-bit operands in $\mathcal{G}$, and we use the copies of the graph G that underlies $\mathcal{G}$ to realize the connections of the macro-graph. Each copy of the macro-PE $\mathcal{K}$ operates as a k-bit ALU, and each bit-serial PE in $\mathcal{K}$ is responsible for one bit-wide slice. In order to assign slices to PEs, we specify an *ordering assignment* that numbers the PEs in the macro-PE $\mathcal{K}$, from 0 to $k-1$. We implement the macro-array $\mathcal{G}$ on $\mathcal{H}$ so that the ith bit of every k-bit operand in $\mathcal{G}$ resides in the ith PE of the corresponding macro-PE, while all the bits have the same memory, port, or register address.

We effect the transfer of a k-bit operand between two adjacent macro-PEs, call them $\mathcal{K}_1$ and $\mathcal{K}_2$, by (emulating) k parallel single-bit transfers in $\mathcal{G}$: for all $0 \leq i \leq k-1$ in parallel, the ith PE in array $\mathcal{K}_1$ transfers a single bit to the ith PE in array $\mathcal{K}_2$. For each data transfer between macro-PEs in the macro-array, the implementation must identify the actual port(s) of the physical, bit-serial PEs that effect the transfer. For a good emulation, the assignment of physical ports to macro-ports should be uniform across macro-PEs.

Implementing memory accesses is immediate within our framework, since all bits of a memory word within a macro-PE have the same bit-serial address; this common address becomes the macro-address of the word. By concentrating on *direct* memory access in the remainder of our study, we allow $\mathcal{G}$ to inherit either direct or direct-plus-indirect memory access from $\mathcal{H}$.

To implement control in the (logical) direct-product array, we must provide the equivalent of loading the activity bit AB depending on some bit(s) of the processor index. The most natural solution retains the bit-serial processor indices, while endowing them with a different interpretation. Each binary processor index is divided into two fields—one for the macro-PE index in the macro-array (MIR) and one for the node within the macro-PE (AIR). In this way, a (physical) bit-group in the PIR can characterize either individual like-named PEs within all macro-PEs (for arithmetic/logical operations) or whole macro-PEs as units (for data transfer). The detailed implementation of this strategy depends on the specific structure of the network underlying the array $\mathcal{H}$.

Ingredient 2: Permutation Routing

A *(partial) permutation route* in a processor array $\mathcal{A}$ is a set of inter-PE messages for which each PE node of $\mathcal{A}$ is the source of (at most) one message and the destination of (at most) one message. The term "permutation route" reflects the fact that the source-destination pairs form a (partial) *permutation* of the PEs of $\mathcal{A}$.

(Partial) permutation routes afford us an efficiently implemented vehicle for achieving a variety of internal data transfers that our strategy requires. Note that we neither require nor consider the routing of arbitrary permutations in our study; rather, we restrict attention to three specific types of permutations, all of which yield to efficient solutions within our computational framework.

In the following descriptions, the reader should keep in mind that the contribution of our study lies as much in the implementation of the required routings as in the representation of these problems in terms of permutation routing. The major implementational challenge is to orchestrate these routes so that

- items to be moved at each route can be accessed efficiently *within local memories, within a SIMD regimen*;
- control of the routes can be specified within a SIMD regimen

and to accomplish these feats without increasing the number of permutations that must be routed. These features of the routing procedures are best appreciated within the sections devoted to specific array topologies.

Emulation Routing. An emulation of a guest processor array $\mathcal{G}$ by a host processor array $\mathcal{H}$ entails two mappings. First, each PE p of $\mathcal{G}$ is assigned to a PE $\alpha(p)$ of $\mathcal{H}$ that will perform its role. Second, each communication link of $\mathcal{G}$ is assigned a routing-path in $\mathcal{H}$, along which the link's messages will be sent; specifically, the communication link from PE p to PE q in $\mathcal{G}$ is assigned a routing-path in $\mathcal{H}$ from PE $\alpha(p)$ to PE $\alpha(q)$. Let us now "color" the links of array $\mathcal{G}$ so that all links entering the same PE receive distinct colors and all links leaving the same PE receive distinct colors.[6] If we interpret each link (p,q) of $\mathcal{G}$ as a partial mapping of the PEs of $\mathcal{G}$ that maps PE p to PE q, then each set of links of $\mathcal{G}$ that receive the same color specifies a (partial) permutation of the PEs of $\mathcal{G}$. Performing such a coloring of the links of $\mathcal{G}$ and such an interpretation of the coloring thus reduces the problem of emulating one communication step of array $\mathcal{G}$ to the problem of performing d (partial) permutation routes in $\mathcal{H}$, where d is the larger of the maximum indegree of any PE of $\mathcal{G}$ and the maximum outdegree of any PE of $\mathcal{G}$.

Word Shifting. Shifting k-bit words within a macro-PE each of whose constituent PEs contains one bit of the word is transparently a permutation route within the macro-PE $\mathcal{K}$. The route is total when the shift is circular; it is partial otherwise.

Datapath Conversion. In our implementation of the meta-instruction GAUGE k, the conversion of the datapath to width k takes place, in parallel, within all macro-PEs. Assume

[6]Efficient algorithms for such coloring abound; cf. [7], [8], [27].

that the PEs in each macro-PE have been ordered from 0 to $k-1$ in some uniform way. The conversion procedure operates on the words

$$b_{i,0}b_{i,1}\cdots b_{i,k-1}$$

in the memory of the ith PE in each macro-PE; it transposes the contents of the memories so that the memory of the ith PE ends up containing the vectors

$$b_{0,i}, b_{1,i}, \ldots, b_{k-1,i}$$

of ith bits of all words in the collective memory of the macro-PE. A simple example of this operation appears in Figure 2, which depicts the transition between three 3-bit words stored in bit-serial fashion and the same words stored in bit-parallel fashion, within a 3-PE macro-PE.

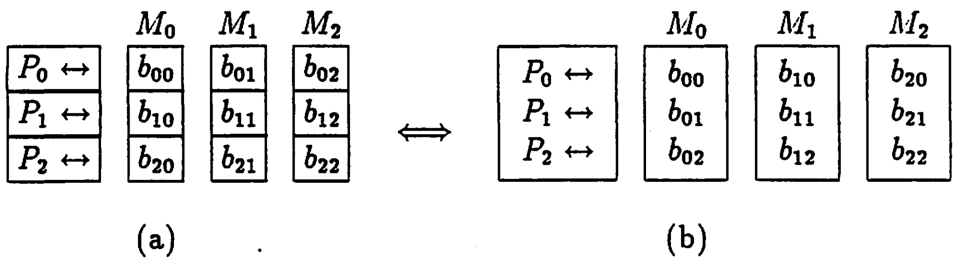

M_0 M_1 M_2
$P_0 \leftrightarrow$ b_{00} b_{01} b_{02}
$P_1 \leftrightarrow$ b_{10} b_{11} b_{12}
$P_2 \leftrightarrow$ b_{20} b_{21} b_{22}
$\Longleftrightarrow$
M_0 M_1 M_2
$P_0 \leftrightarrow$ b_{00} b_{10} b_{20}
$P_1 \leftrightarrow$ b_{01} b_{11} b_{21}
$P_2 \leftrightarrow$ b_{02} b_{12} b_{22}
(a)
(b)

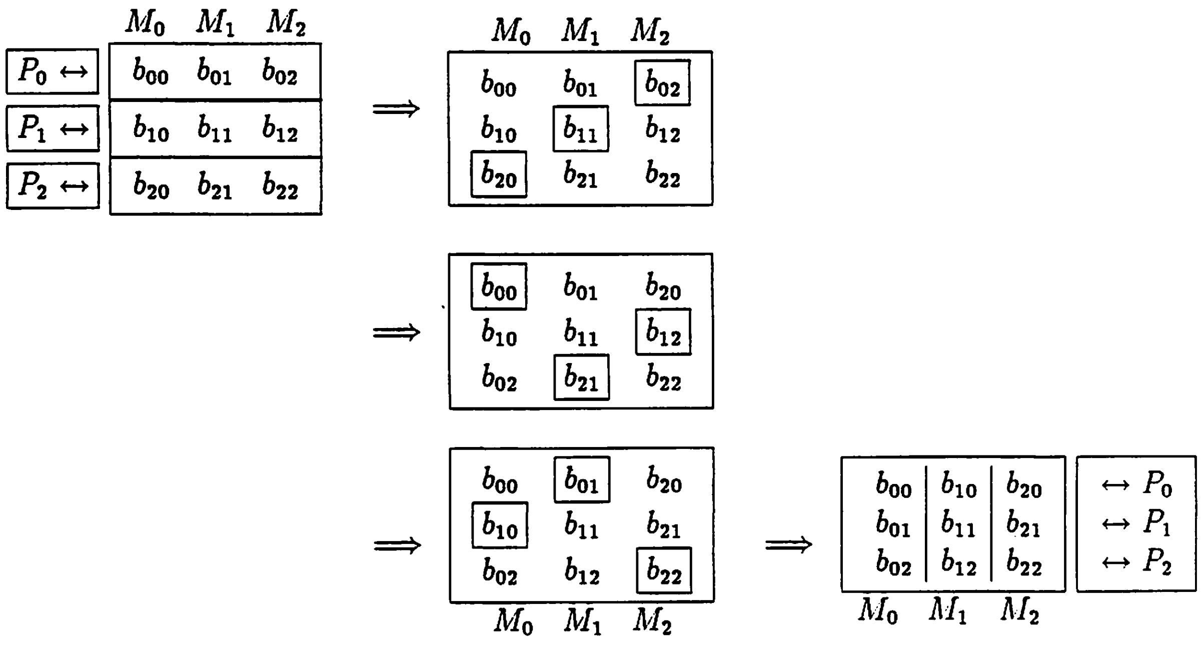
Figure 2: (a) *Three 3-bit words stored in bit-serial (gauge-1) fashion.* (b) *The same words stored in bit-parallel (gauge-3) fashion.*

We illustrate in Figure 3 how datapath conversion from unit gauge size to gauge size k can be effected via a sequence of k permutation routes, when PEs access their memories directly. We soon see that the illustrated procedure can be modified to accommodate other modes of addressing (and, of course, even values of k).

It is important to note that in all three situations where our strategy employs permutation routes, the permutations to be routed come from a fixed set of permutations that are

- known *a priori*
- amenable to SIMD implementation.

In a large variety of networks—including the ones we discuss in later sections, such routing can be performed in time only slightly exceeding the diameter (i.e., the maximum inter-node distance) of the network underlying the array in which the routing takes place. When placing



M_0 M_1 M_2
$P_0 \leftrightarrow$
$P_1 \leftrightarrow$
$P_2 \leftrightarrow$
b_{00} b_{01} b_{02}
b_{10} b_{11} b_{12}
b_{20} b_{21} b_{22}
$\Longrightarrow$
$\boxed{b_{02}}$ $\boxed{b_{11}}$ $\boxed{b_{20}}$
$\boxed{b_{00}}$ $\boxed{b_{12}}$ $\boxed{b_{21}}$
$\boxed{b_{01}}$ $\boxed{b_{10}}$ $\boxed{b_{22}}$
$\leftrightarrow P_0$
$\leftrightarrow P_1$
$\leftrightarrow P_2$

Figure 3: *One way to accomplish datapath conversion as a sequence of permutation routes. Individually boxed bit-positions represent items included in the permutation routed at that stage.*

these timing estimates in perspective, one must recall that the routing for both word-shifting and datapath conversion takes place in the macro-PEs, each of which is a k-PE array; only the emulation routing takes place in the N-PE host array.

Before proceeding further, we present a topology-independent algorithm for datapath conversion, which is the most complex operation that we are *guaranteed* to need to implement when achieving multigauging via emulations. (Depending on the topology of the network underlying array $\mathcal{H}$, the other ingredient algorithms, such as emulating a direct-product array, could be quite complex also.) We carry our discussion of datapath conversion to this level of detail for several reasons, the first of which is the combination of the importance of the operation to our strategy and the complexity of the operation. Our detailed look at the operation also allows us to focus on the differences of the three modes of memory access that we consider; we conclude that, absent knowledge of the topology of the network underlying our host array, the SIMD regimen we are assuming favors shift-register memory. Finally, our detailed specification of datapath conversion affords us a vehicle for introducing the (hopefully) transparent pseudo-programming language we use to specify all of our implementations.

The instructions in our language are high level, in that their implementations may take different numbers of basic instructions on different physical host architectures. Although these numbers may not be equal across our instruction set, even for a specific host architecture $\mathcal{H}$, the variations across instructions are within small constant factors which are easily derived from our pseudo-code (once $\mathcal{H}$ is specified). For illustration, an instruction such as:

```
if (PIR[$i$] = 0) then from M[$j$] output ($port$)
```

may translate into a sequence of low-level instructions to

1. load the ALU from the ith bit of the PIR
2. load the AB according to this bit
3. load the ALU from the memory bit M[j]
4. transfer the ALU bit to the port named *port*
5. clear the AB

When assessing the cost of a macro-instruction, we estimate the number of basic instructions it is likely to take on a conventional host array architecture; we estimate the cost in terms of the host array's instructions only, ignoring the cost of the controller program.[7]

[7]The costs of the controller program are small, residing only in the time for procedure calls and some small amount of space that does not depend on k.

We now present the main assumptions and notational conventions that persist throughout our procedure specifications.

Notation and Conventions.

- We assume that the only data type in the N-PE host array $\mathcal{H}$ is *bit*, while the central (SIMD) controller's instructions operate on *integer* data.
- We assume that the repertoire of interesting gauge sizes—hence, also the numbers of nodes in a macro-PE is always a power of 2. Clerical changes obviate this assumption, which is motivated by a desire for notational and clerical simplicity.
- We establish the following notational conventions.
  - We write macro-instruction names in uppercase, host-array instruction names in lowercase, and procedure names in capitalized form.
  - Z_2 denotes the set $\{0, 1\}$, and $\bar{\beta} = 1 - \beta$ for $\beta \in Z_2$.
  - Z_2^n denotes the set of length-n binary strings, the length-0 string being the null string; $\text{lgth}(x)$ denotes the length of string $x \in Z_2^n$, (which is n); β^n, where $\beta \in Z_2$, denotes the string in Z_2^n that consists of n occurrences of β.
  - We write $\log n$ for $\log_2 n$, and we write $\ell(k)$ for $\log k$, where k is the sought gauge size.

Our topology-independent k-bit datapath-conversion procedure assumes a given (but arbitrary) k-node interconnection network K_k. For purposes of specifying and analyzing the procedure, we posit the existence of a SIMD protocol that routes an arbitrary permutation π within K_k within $R(k)$ steps and of shift-register access to local memory. The local memory of the ith PE contains the bit vector $b_{i,0}b_{i,1} \cdots b_{i,k-1}$; the ith PE can access only the bit at position $b_{i,0}$. The shift instruction left transforms the contents of the memories so that the memory of the ith PE ends up containing the vector $b_{i,k-1}b_{i,0}b_{i,1} \cdots b_{i,k-2}$. Analogously, the shift instruction right produces the vector $b_{i,1}b_{i,2} \cdots b_{i,k-1}b_0$ in the memory of the ith PE.

Note. We present and analyze our conversion procedure within the context of converting a single set of k k-bit words from bit-serial to bit-parallel format within the macro-PEs built upon the network K_k. A "real" application of this procedure would likely convert the entire memory of all macro-PEs. All of these memories would be converted in parallel, but the conversion within each macro-PE would proceed serially through chunks of k k-bit words each.

```
macro-instruction GAUGE ($k$) is {
  -- network-independent datapath conversion
  LeftShift               -- shift by $i+1$ positions; $i$th PE now accesses bit $b_{i,k-i-1}$
  for $j := 0$ to $k-1$ do {   -- all bits in the memory
    route $\pi(j,k)$          -- permutation sending the accessible bit to destination PE
    right }               -- shift; the next bit is ready for routing
  RightShift }            -- by $i+1$ positions; to compensate for the initial LeftShift
```

The permutation $\pi(j,k)$ on the set $\{a_0, a_1, \ldots, a_{k-1}\}$ is given by:

$$\pi(j,k)(a_i) = \begin{cases} a_{j-i-1} & \text{if } i < j \\ a_{k-1-i+j} & \text{if } i \geq j \end{cases}$$

The left circular shift (the right shift being analogous), which makes the element $b_{i,k-i-1}$ in the memory of ith PE accessible, is specified by:

```
procedure LeftShift is {
  -- left shift in $i$th PE by $i+1$ positions
  left                         -- shift by 1 position
  for $m := 0$ to $\ell(k) - 1$ do {   -- for bits in PIR
    if (PIR[right] = 1) then {   -- for each 1
      for $j := 1$ to $2^m$ do {     -- as many shifts
        left }}}}
```

The operation LeftShift as specified, with memory behaving as a shift register, performs $k + O(\log k)$ bit-serial computation instructions per conversion (k memory accesses and $O(\log k)$ adjustments to activity bits); the constant hiding in the big-O is estimated to be under 5. The total datapath-conversion cost is, then, $2k + O(\log k) + kR(k)$. Absent details about the topology of the network, the term $kR(k)$ cannot be improved upon.

Were we to implement the same algorithm for LeftShift on memory with *direct* addressing and without shift-register capabilities, the time for datapath conversion would rise to $2k^2 + O(\log k) + kR(k)$; the squaring of the linear term reflects the fact that shifting a word by one position now requires k memory accesses. Implementing the algorithm on memory with *indirect* addressing and without shift-register capabilities would replace physical shifting by an appropriate reference; thus, the algorithm would operate in time $O(kR(k)\log k)$ when address arithmetic is bit-serial (so that incrementing the address register takes time proportional to the length of the register) and $O(kR(k))$ when address arithmetic handles words. The only addressing mechanism whose advantages outweigh those of the shifting capability is the *address index register*, which allows its contents to be *added* to the value of the address field to obtain the actual address. Such a mechanism reduces the time for datapath conversion to $O(kR(k))$.

Ingredient 3: Emulating Complete Binary Trees

The ability to efficiently emulate the *complete binary tree* $\mathcal{T}_h$ *of height* $h = \lceil \ell(k) \rceil$ plays a significant computational role in our emulation strategy.

The tree $\mathcal{T}_h$ has $2^{h+1} - 1$ nodes which are conventionally identified with the set of all $2^{h+1} - 1$ binary strings x of length $\leq h$. These nodes are conventionally partitioned into *levels* by their lengths: the *root* of $\mathcal{T}_h$ is the unique node at level 0, and the *leaves* of $\mathcal{T}_h$ are all 2^h binary strings of length h. The *arcs* of $\mathcal{T}_h$ connect every nonleaf node x to its *left child* $x0$ and its *right child* $x1$.

We use complete binary trees in two major roles, first as *valuation trees*, and second, as (logical) networks for implementing the *parallel-prefix operator* [14]. We discuss each of these roles in turn.

Valuation Trees. A variety of useful *binary associative* operations can be computed on sequences efficiently, by emulating complete binary trees level by level, using them as *valuation trees*. To wit, let $*$ denote such an operation, let $\vec{x} = (x_0, x_1, \ldots, x_{k-1})$ be a k-tuple of elements in the domain of $*$, and let $h = \lceil \ell(k) \rceil$. The following algorithm will compute the "product"

$$\xi = x_0 * x_1 * \cdots * x_{k-1}$$

in h parallel emulation steps. Load $\vec{x}$ into the leaves of $\mathcal{T}_h$, from "left to right," as indicated in Figure 4, and (logically, via emulation) sweep up $\mathcal{T}_h$ level by level, computing at each node the $*$-product of the data in its children, as indicated in Figure 5. After h emulation steps, the value ξ will reside in the PE assigned to emulate the root of $\mathcal{T}_h$.

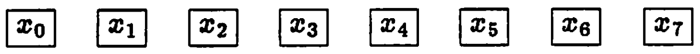


Figure 4: *The 8-element data-vector $\vec{x}$ loaded in the leaves of $\mathcal{T}_3$, from "left to right."*

Examples of operations that one might wish to implement using valuation trees are

- Find the sum (resp., product, minimum, maximum) of k integers; in these cases, $* = +$ (resp., $* = \times, \min, \max$).
- Find the **and** (resp., **or**) of k boolean values; in this case, $* = \wedge$ (resp., $* = \vee$).

$$\begin{array}{llllllll}
\text{STEP 0} & \boxed{x_0} & \boxed{x_1} & \boxed{x_2} & \boxed{x_3} & \boxed{x_4} & \boxed{x_5} & \boxed{x_6} \quad \boxed{x_7} \\
\text{STEP 1} & \boxed{x_0 * x_1} & \boxed{x_2 * x_3} & \boxed{x_4 * x_5} & \boxed{x_6 * x_7} \\
\text{STEP 2} & \boxed{x_0 * x_1 * x_2 * x_3} & \boxed{x_4 * x_5 * x_6 * x_7} \\
\text{STEP 3} & \boxed{x_0 * x_1 * x_2 * x_3 * x_4 * x_5 * x_6 * x_7}
\end{array}$$

Figure 5: *The 3-step computation of the product ξ from the vector $\vec{x}$. Each step can be viewed as occurring at the next higher level of the valuation tree.*

In general, if computing $*$ on a pair of data takes c steps, then computing the $*$-product of a length-k sequence takes $c \log k$ steps.

The Parallel-Prefix Operator. The *parallel-prefix operator* [14] (see also the scan operator of [11], [6]) generalizes the operation of evaluating the product ξ of the elements of the vector $\vec{x}$, by simultaneously evaluating the products of all prefixes of $\vec{x}$. In other words, we again start with a binary associative operation $*$ and a vector $\vec{x} = (x_0, x_1, \ldots, x_{k-1})$ of elements in the domain of $*$, but now we compute the k-tuple of products

$$\textbf{parallel-prefix}(\vec{x}; *) = \begin{pmatrix} x_0 \\ x_0 * x_1 \\ \vdots \\ x_0 * x_1 * \cdots * x_{k-1} \end{pmatrix}.$$

The ability to compute the parallel-prefix operator efficiently enables the efficient parallel computation of myriad useful functions, including:

- carry-lookahead addition (by having $*$ transmit information about the propagation and generation of carries)
- broadcast (by having $*$ transmit information from left argument to right argument)
- selection operations, such as max, min, etc. (by having $*$ perform pairwise comparisons, retaining winners and disposing of losers).

The ability to emulate complete binary trees efficiently, level by level, endows one with the ability to compute the parallel-prefix operator efficiently. We sketch a justification of this assertion.

The Algorithm

1. Start with the x_i in the leaves of $\mathcal{T}_h$, in left-to-right order, as indicated in Figure 4.
2. In h steps, sweep up $\mathcal{T}_h$, using it as a valuation tree, as in Figure 5. Each node of the tree retains the partial product computed by the subtree of which it is the root.
3. In two steps, do the following sibling transfers from lefthand nodes of $\mathcal{T}_h$ through parent nodes to righthand nodes:
   (a) Each righthand nonleaf node of $\mathcal{T}_h$ *replaces* its result by its lefthand sibling's result.
   (b) Each righthand leaf node of $\mathcal{T}_h$ *combines* its lefthand sibling's result with its result.

   The result of the sibling transfer is illustrated in Figure 6.
4. Level by level, in $h - 1$ steps, each righthand node passes its current value down.
   (a) A nonleaf child in $\mathcal{T}_h$ *replaces* its result by the new result.
   (b) A leaf child in $\mathcal{T}_h$ *combines* the new result with its result.

   The contents of $\mathcal{T}_h$ after one iteration of this subprocedure are illustrated in Figure 7

After completing step 4, the leaves of $\mathcal{T}_n$ contain the desired result, as illustrated in Figure 8.

D. Summary

This completes our survey of the ingredients of our strategy for achieving multigauge behavior via emulations. We encapsulate the strategy: We enable a bit-serial N-PE processor

Level 0	$\boxed{x_0 * x_1 * x_2 * x_3 * x_4 * x_5 * x_6 * x_7}$							
Level 1	$\boxed{x_0 * x_1 * x_2 * x_3}$	$\boxed{x_0 * x_1 * x_2 * x_3}$						
Level 2	$\boxed{x_0 * x_1}$	$\boxed{x_0 * x_1}$	$\boxed{x_4 * x_5}$	$\boxed{x_4 * x_5}$				
Level 3	$\boxed{x_0}$	$\boxed{x_0 * x_1}$	$\boxed{x_2}$	$\boxed{x_2 * x_3}$	$\boxed{x_4}$	$\boxed{x_4 * x_5}$	$\boxed{x_6}$	$\boxed{x_6 * x_7}$

Figure 6: *The contents of the tree after the sibling transfer.*

Level 0 $\boxed{x_0 * x_1 * x_2 * x_3 * x_4 * x_5 * x_6 * x_7}$

Level 1 $\boxed{x_0 * x_1 * x_2 * x_3}$ $\boxed{x_0 * x_1 * x_2 * x_3}$

Level 2 $\boxed{x_0 * x_1 * x_2 * x_3}$ $\boxed{x_0 * x_1 * x_2 * x_3}$ $\boxed{x_0 * x_1 * x_2 * x_3}$ $\boxed{x_0 * x_1 * x_2 * x_3}$

Level 3 $\boxed{x_0}$ $\boxed{x_0 * x_1}$ $\boxed{x_0 * x_1 * x_2}$ $\boxed{x_0 * x_1 * x_2 * x_3}$

$\boxed{x_4}$ $\boxed{x_4 * x_5}$ $\boxed{x_4 * x_5 * x_6}$ $\boxed{x_4 * x_5 * x_6 * x_7}$

Figure 7: *The contents of the tree after one iteration of passing down values.*

Leaf 0 $\boxed{x_0}$

Leaf 1 $\boxed{x_0 * x_1}$

Leaf 2 $\boxed{x_0 * x_1 * x_2}$

Leaf 3 $\boxed{x_0 * x_1 * x_2 * x_3}$

Leaf 4 $\boxed{x_0 * x_1 * x_2 * x_3 * x_4}$

Leaf 5 $\boxed{x_0 * x_1 * x_2 * x_3 * x_4 * x_5}$

Leaf 6 $\boxed{x_0 * x_1 * x_2 * x_3 * x_4 * x_5 * x_6}$

Leaf 7 $\boxed{x_0 * x_1 * x_2 * x_3 * x_4 * x_5 * x_6 * x_7}$

Figure 8: *The final result of the parallel-prefix algorithm.*

array $\mathcal{H}$ to behave as though it were a bit-parallel $\lfloor N/k \rfloor$-PE processor array $\mathcal{G}$ of gauge size k, as follows.

1. We have $\mathcal{H}$ emulate a direct-product array $\mathcal{K} \times \mathcal{G}$, where $\mathcal{K}$ is a k-PE processor array.
2. We use (partial) permutation routing on $\mathcal{H}$ to implement communication paths in $\mathcal{K} \times \mathcal{G}$.
3. We have $\mathcal{H}$ emulate (partial) permutation routing on $\mathcal{K}$ to implement datapath conversion (corner-turning) and shift operations on k-bit words.
4. We have $\mathcal{H}$ emulate height-$\ell(k)$ (complete binary) valuation trees to compute *accumulative* logical and arithmetic operations.
5. We have $\mathcal{H}$ emulate height-$\ell(k)$ complete binary trees to compute the parallel-prefix of appropriate operations, thereby implementing k-bit-parallel operations such as carry-lookahead arithmetic.
6. We have the PEs of $\mathcal{K}$ do bitwise logic, one bit per PE.

In the next three sections, we add detail to our emulation strategy, by describing its implementation on three different array topologies, hypercubes, de Bruijn networks, and meshes with reconfigurable buses. The reader will note that our aggregation strategy (i.e., the strategy we use to choose the topologies of the macro-PE array $\mathcal{K}$ and the macro-array $\mathcal{G}$) with these exemplary topologies cleaves to a principle we have not yet enunciated:

> **The Principle of Self-Similarity.** Whenever possible, choose the macro-PE (array) $\mathcal{K}$ and the macro-array $\mathcal{G}$ to be smaller versions of the array $\mathcal{H}$. This principle places our multigauge architectures among the type A architectures of [23].

In accord with this principle, we (logically) decompose a hypercube into a hypercube of hypercubes, a de Bruijn network into a de Bruijn network of de Bruijn networks, and a mesh into a mesh of meshes. This type of decomposition is very convenient when it can be achieved, because *it allows one to retain network-dependent algorithmic strategies across gauge sizes.* Such a decomposition is usually possible with popular network topologies, because these topologies tend to comprise families of like-structured but varying size instances. The one practical detractor from this decomposition principle is that it restricts the range of gauge sizes that one can aspire to. Thus, we always choose a gauge size that is a power of 2 when we implement our strategy on a hypercubes or de Bruijn networks, and we always choose a gauge size that is a perfect square when we implement our strategy on meshes.

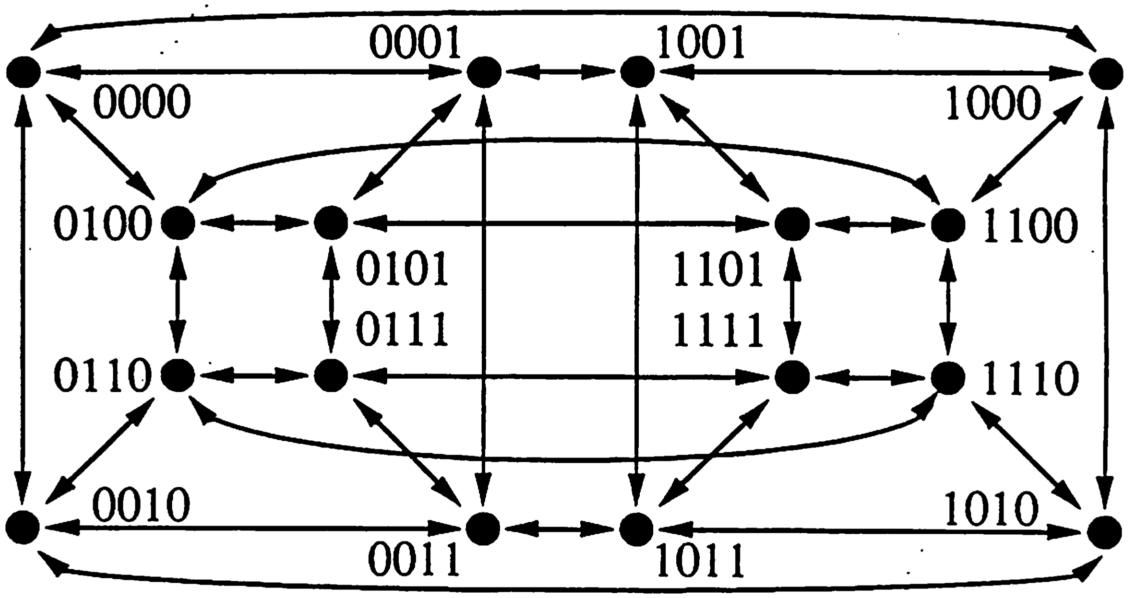


Figure 9: *The 4-dimensional hypercube graph Q_4.*

4 Hypercube Networks

The *n-dimensional (boolean) hypercube* Q_n is a graph whose node-set is Z_2^n. The arcs of Q_n connect every pair of nodes u and v that differ in just one bit-position, i.e., node u has the form $u = x\beta y$ and node v has the form $v = x\bar{\beta}y$, for some $\beta \in Z_2$ and $xy \in Z_2^{n-1}$. See Figure 9.

In the architecture $\mathcal{Q}_n$ built on the graph Q_n, we linearly order PEs according to the numerical ordering on their names: the ith PE of $\mathcal{Q}_n$ resides at the node x of Q_n which is the binary representation of integer i. Each PE in $\mathcal{Q}_n$ has n input ports and n output ports: the port that leads node-PE $x\beta y$ to node-PE $x\bar{\beta}y$, where $x \in Z_2^{n-m}$, $\beta \in Z_2$, and $y \in Z_2^{m-1}$, is called the mth port of node-PE $x\beta y$.

4.1 Algorithmic Issues

A. Emulating a Direct-Product Network

We sketch the algorithmic basis for the emulation by an arbitrary $(N = 2^n)$-PE hypercube array $\mathcal{Q}_n$ of a direct-product hypercube array $\mathcal{Q}_{\ell(k)} \times \mathcal{Q}_{n-\ell(k)}$. Within the context of our study:

- $\mathcal{Q}_n$ is the host array $\mathcal{H}$;
- $\mathcal{Q}_{\ell(k)}$ is the macro-PE array $\mathcal{K}$, and $\mathcal{Q}_{n-\ell(k)}$ is the macro-array $\mathcal{G}$.

This emulation is, conceptually, a very simple one, because the hypercube graph Q_n is (isomorphic to) the direct-product hypercube graph $Q_{\ell(k)} \times Q_{n-\ell(k)}$.

As we discussed in Section 3, an emulation has two component mappings. First we must assign each PE of the guest array to the PE of the host array that will emulate it; then we must assign routing paths in the host array to effect communications along the communication links of the guest array. Because our emulation builds on an embedding of the graph $Q_{\ell(k)} \times Q_{n-\ell(k)}$ in the graph Q_n, we describe the emulation in graph-theoretic terms; cf. [1].

We **assign** each node (x, y) of the guest direct-product hypercube graph $Q_{\ell(k)} \times Q_{n-\ell(k)}$ to the node of the host hypercube graph Q_n whose string-name is the concatenation xy of the component string-names of the guest node. We thereby implicitly assign the PEs of the guest array $\mathcal{Q}_{\ell(k)} \times \mathcal{Q}_{n-\ell(k)}$ to PEs of the host array $\mathcal{Q}_n$.

We **route** the links of the guest array $\mathcal{Q}_{\ell(k)} \times \mathcal{Q}_{n-\ell(k)}$ within the host array $\mathcal{Q}_n$ by routing the arcs of the graph $Q_{\ell(k)} \times Q_{n-\ell(k)}$ within the graph Q_n, as follows.

- We route the arc
$$(x_0\xi x_1,\ y) \longrightarrow (x_0\bar{\xi}x_1,\ y)$$
($\xi \in Z_2$, $x_0x_1 \in Z_2^{\ell(k)-1}$) within copy y of $Q_{\ell(k)}$ along the following unit-length path (i.e., arc) in Q_n.
$$x_0\xi x_1 y \longrightarrow x_0\bar{\xi}x_1 y.$$

- We route the arc
$$(x,\ y_0\eta y_1) \longrightarrow (x,\ y_0\bar{\eta}y_1)$$
($\eta \in Z_2$, $y_0y_1 \in Z_2^{n-\ell(k)-1}$) between copies $y_0\eta y_1$ and $y_0\bar{\eta}y_1$ of $Q_{\ell(k)}$ along the following unit-length path in Q_n.
$$xy_0\eta y_1 \longrightarrow xy_0\bar{\eta}y_1.$$

This emulation maps arcs of the guest graph to arcs of the host graph; therefore, the emulation incurs no cost.

B. (Partial) Permutation-Routing

In subsection 4.2, we indicate how the permutation-routes needed for our emulations can be done efficiently within a SIMD regimen, with very simple masking.

C. Emulating a Complete Binary Tree (level by level)

Our level-by-level emulation of the complete binary tree $\mathcal{T}_{\ell(k)}$ by the hypercube array $\mathcal{Q}_{\ell(k)}$ exploits the string-names of the cube's nodes to effect the emulation with no slowdown.

We **assign** PEs of $\mathcal{T}_{\ell(k)}$ to PEs of $\mathcal{Q}_{\ell(k)}$ via the following many-to-one map. Each PE x of $\mathcal{T}_{\ell(k)}$ is assigned to PE $x1^{\ell(k)-\mathrm{lgth}(x)}$ of $\mathcal{Q}_{\ell(k)}$; in particular, each PE at level h of $\mathcal{T}_{\ell(k)}$ is assigned to a PE of $\mathcal{Q}_{\ell(k)}$ whose name ends with a string of $\ell(k) - h$ 1s.

We **route** links of $\mathcal{T}_{\ell(k)}$ within $\mathcal{Q}_{\ell(k)}$ via shortest paths. Because of the many-one nature of the assignment, this emulation is not merely a graph embedding.

- We route the link $(x, x0)$ of $\mathcal{T}_{\ell(k)}$ via the unit-length path
$$(x1^{\ell(k)-\mathrm{lgth}(x)} \longrightarrow x01^{\ell(k)-\mathrm{lgth}(x)-1})$$
within $\mathcal{Q}_{\ell(k)}$.
- We route the link $(x, x1)$ of $\mathcal{T}_{\ell(k)}$ as a "null" link within $\mathcal{Q}_{\ell(k)}$, because tree-PEs x and $x1$ are assigned to the same PE $x1^{\ell(k)-\mathrm{lgth}(x)}$ of $\mathcal{Q}_{\ell(k)}$.
- We route the predecessor link $(x\beta, x)$ of $\mathcal{T}_{\ell(k)}$, $\beta \in Z_2$, via the "inverse" of the routing path for the link $(x, x\beta)$.

We chose our PE-assignment with the parallel-prefix algorithm in mind. We decrease the time to emulate each communication step of a righthand PE by employing an assignment that obviates moving data when following a rightward link.

Our emulation maps links of $\mathcal{T}_{\ell(k)}$ to paths of length 0 or 1 in $\mathcal{Q}_{\ell(k)}$; therefore, the emulation incurs no slowdown.

4.2 Implementation Issues

Computation, Communication, and Control. Because $\mathcal{Q}_n$ is isomorphic to the direct-product graph $\mathcal{Q}_{\ell(k)} \times \mathcal{Q}_{n-\ell(k)}$, data transfers in the macro-array $\mathcal{G}$ are emulated with no slowdown. We let the PIR of each bit-serial PE be the concatenation of the AIR and the MIR, so that

$$\mathrm{AIR} = \mathrm{PIR}[n - \ell(k) \ldots n - 1] \ ; \ \mathrm{MIR} = \mathrm{PIR}[0 \ldots n - \ell(k) - 1].$$

Letting $g = 2^{n-\ell(k)}$, we order the macro-PEs in the macro-array $\mathcal{G}$ so that the ith macro-PE consists exactly of those PEs of $\mathcal{Q}_n$ whose numbers (in $\mathcal{Q}_n$) are in the set $A^{(i)} = \{a \mid a \equiv i (\mathrm{mod}\ g)\}$.

The communication macro-instruction that outputs a k-bit word to macro-port j in $\mathcal{G}$ is:

```
macro-instruction OUTPUT ($j$: Port) is {
  - - bit-parallel output from macro-port $j$ in $\mathcal{G}$
  output ($j$) }
```

Within a copy of the macro-PE $\mathcal{K}$, we order PEs so that the ith bit position in all macro-PE is occupied exactly by those PEs of $\mathcal{Q}_n$ whose numbers (in $\mathcal{Q}_n$) are in the set $A_i = \{a \mid i = \lfloor a/g \rfloor\}$.

The macro-PE primitive instruction AgOutput, which outputs a bit to port j in $\mathcal{K}$ is specified as follows:

```
procedure AgOutput ($j$: Port) is {
  - - aggregate data transfer in macro-PE $\mathcal{K}$ from port $j$
  output ($n - \ell(k) + j$) }
```

Both output and input data transfer operations (which are done analogously) in $\mathcal{G}$ and $\mathcal{K}$ are performed in unit time, with no slowdown compared to *bit-serial* data transfers in $\mathcal{H} = \mathcal{Q}_n$. For perspective and contrast, we remark that if an $(n - \ell(k))$-dimensional subcube of $\mathcal{Q}_n$ were used to perform the desired computation bit-serially on k-wide data, then the cost of an I/O operation would be $O(k)$, which is dramatically more costly than in our emulation.

The parallel-prefix computations that perform arithmetic operations within each macro-PE $\mathcal{K}$ require exactly the PEs on level h of the emulated tree $\mathcal{T}_{\ell(k)}$ to be active at stage $\ell(k) - h$ of the ascending phase, or at stage h of the descending phase of the algorithm. This emulation incurs no slowdown because of the PE assignment we use. The cost of an arithmetic operation in our emulation is thus $O(\ell(k)) = O(\log k)$, while the cost of such an operation in bit-serial processing of the same data by $\mathcal{Q}_n$ is $O(k)$. The constant hidden in the first (emulation) expression must be somewhat larger than in the second (bit-serial) expression, because the parallel-prefix computation operates on 2-bit data. This causes the emulated arithmetic operations to be slower than the bit-serial ones for a few small gauge sizes. Compensating for this slight slowdown are two types of significant speedup achieved by the emulation approach.

The less important, though still significant, speedup is observed when executing *computational* instructions. Arithmetic operations are accelerated from linear to logarithmic time per operation in all gauge sizes except the initial few. Logical operations experience an even larger speedup, from linear to constant time per operation.

The more important speedup is experienced with *communication* instructions, which are accelerated from linear to constant time per operation. This speedup strongly and favorably affects the overall performance of the emulated machine—for all gauge sizes—to an extent that is dictated by the frequency of communication steps in any particular computation. It is worth emphasizing that while many parallel algorithms excel at accelerating computation, they get bogged down when computations require much communication. In contrast, the more parallel the computation is, and the more inter-process communication it requires, the *greater* the advantage of our emulation approach.

Datapath Conversion. Our datapath conversion algorithm for the macro-PEs $\mathcal{K}$ is very similar to the matrix transposition algorithm MTADEA of [19], except that we provide a detailed memory layout, augmented with a precise description of the local bit-wide data movements under the SIMD regimen. In particular, we interleave block exchanges on a bit level, so that our overhead is only one bit per bit-serial PE (the bit variable save). Our macro-instruction GAUGE k performs exactly $k \log k$ bit-serial data transfers and $O(k \log k)$ bit-serial computation instructions, with the constant in the big-O estimated to be under 20.

```
macro-instruction GAUGE ($k$) is {
  -- datapath conversion in $\mathcal{Q}_{\ell(k)}$
  for $i$ := 0 to $\ell(k) - 1$ do {          -- dimensions
    for $m$ := 1 to $k/2^{i+1}$ do {        -- blocks
      high := $(2m-1)2^i$                 -- block start
      low  := $2(m-1)2^i$                 -- block start
      for $j$ := 1 to $2^i$ do {            -- block bits
        if (AIR[$i$]= 0) then {           -- exchange
          from M[high] AgOutput ($i$) }
        if (AIR[$i$]= 1) then {
          save := M[low]
          AgInput ($i$) to M[low]
          from save AgOutput ($i$) }
        if (AIR[$i$]= 0) then {
          AgInput ($i$) to M[high] }
        low := low + 1                  -- next bit
        high := high +1 }}}}
```

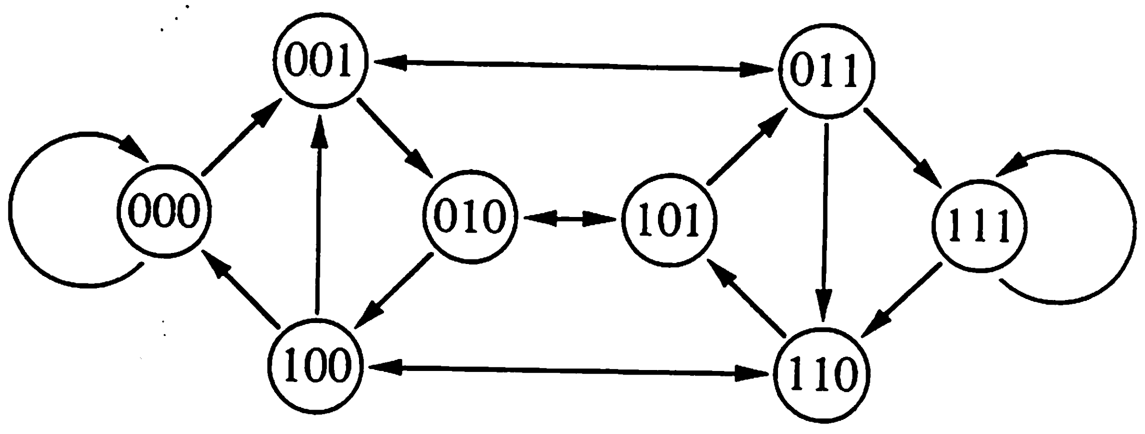


Figure 10: *The order-3 de Bruijn graph D_3.*

5 de Bruijn Networks

The *(base-2) order-n de Bruijn graph* D_n has node-set Z_2^n. The arcs of D_n connect

- each node βx, where $\beta \in Z_2$ and $x \in Z_2^{n-1}$, to nodes $x\beta$ (a *shuffle* arc) and $x\bar{\beta}$ (a *shuffle-exchange* arc),
- each node $x\beta$, where $\beta \in Z_2$ and $x \in Z_2^{n-1}$, to nodes βx (an *unshuffle* arc) and $\bar{\beta}x$ (an *unshuffle-exchange* arc).

See Figure 10.

In the architecture $\mathcal{D}_n$ built on the graph D_n, we linearly order PEs according to the numerical ordering of their names: the ith PE of $\mathcal{D}_n$ resides at the node x of D_n which is the binary representation of integer i. Each PE in $\mathcal{D}_n$ has four input ports and four output ports. For $\beta_0, \beta_1 \in Z_2$ and $x \in Z_2^{n-2}$, the ports of node-PE $\beta_0 x \beta_1$ are named as follows:

from node	to node	port name
$\beta_0 x \beta_1$	$x\beta_1\beta_0$	SHUFFLE
	$x\beta_1\bar{\beta}_0$	SHUFFLE-EXCHANGE
	$\beta_1\beta_0 x$	UNSHUFFLE
	$\bar{\beta}_1\beta_0 x$	UNSHUFFLE-EXCHANGE

5.1 Algorithmic Issues

A. Emulating a Direct-Product Network

We sketch the algorithmic basis of an efficient emulation by an arbitrary ($N = 2^n$)-PE de Bruijn array $\mathcal{D}_n$ of the direct-product de Bruijn array $\mathcal{D}_{\ell(k)} \times \mathcal{D}_{n-\ell(k)}$. In the context of our study:

- The array $\mathcal{D}_n$ is the physical host array $\mathcal{H}$;
- the array $\mathcal{D}_{\ell(k)}$ is the macro-PE $\mathcal{K}$, while the array $\mathcal{D}_{n-\ell(k)}$ is the macro-array $\mathcal{G}$.

Our emulation here is materially more complicated than the corresponding emulation for hypercubes, because de Bruijn networks do not enjoy a direct-product structure. However, in common with that emulation, our emulation here operates within the graph-theoretic framework of embedding the direct product of de Bruijn graphs $D_{\ell(k)} \times D_{n-\ell(k)}$ in the de Bruijn graph D_n. Our embedding derives from [1].

We **assign** nodes of the graph $D_{\ell(k)} \times D_{n-\ell(k)}$ to nodes of the graph D_n by concatenating node-names: each node $(x, y) \in Z_2^{\ell(k)} \times Z_2^{n-\ell(k)}$ of the direct-product graph is assigned to node $xy \in Z_2^n$ of D_n.

We **route** arcs of the direct-product graph in a way that exploits the ability of de Bruijn graphs to "rewrite" node-names by traversing the appropriate paths. For the sake of brevity, we are a bit sketchy in our discussion of the emulation-routing, in two respects:

1. We describe how to implement only the "positive" moves, SHUFFLE and SHUFFLE-EXCHANGE of de Bruijn graphs, leaving to the reader the task of implementing the "negative" moves, UNSHUFFLE and UNSHUFFLE-EXCHANGE, which are implemented by straightforward analogy with their "positive" counterparts.
2. We assume that the sought gauge size is so small that $\ell(k) \leq n - \ell(k)$. This assumption manifests itself in our choosing (in Figures 11 and 12) to rewrite the first-coordinate string of node (x, y) of the product array rather than the second-coordinate string. Of course, when the sought gauge size is so large as to reverse the assumed inequality, one should merely interchange the roles of the first- and second-coordinate strings.

- We route the arc
$$((\beta x, y),\ \ (x\gamma, y))$$
($x \in Z_2^{\ell(k)-1}$, $y \in Z_2^{n-\ell(k)}$, $\beta, \gamma \in Z_2$) within copy y of $D_{\ell(k)}$ via the length-($2\ell(k)$) path from node βxy to node $x\gamma y$ in D_n that is depicted in Figure 11.

$$
\begin{array}{rll}
\beta x y & = \beta \xi_1 \xi_2 \cdots \xi_{\ell(k)-2} \xi_{\ell(k)-1} y & \\
& \longrightarrow \xi_1 \xi_2 \cdots \xi_{\ell(k)-2} \xi_{\ell(k)-1} y \xi_1 & \text{shuffle(-exchange)} \\
& \longrightarrow \xi_2 \cdots \xi_{\ell(k)-2} \xi_{\ell(k)-1} y \xi_1 \xi_2 & \text{shuffle(-exchange)} \\
& \longrightarrow \xi_3 \cdots \xi_{\ell(k)-2} \xi_{\ell(k)-1} y \xi_1 \xi_2 \xi_3 & \text{shuffle(-exchange)} \\
& \vdots & \\
& \longrightarrow \xi_{\ell(k)-1} y \xi_1 \xi_2 \cdots \xi_{\ell(k)-2} \xi_{\ell(k)-1} & \text{shuffle(-exchange)} \\
& \longrightarrow y \xi_1 \xi_2 \cdots \xi_{\ell(k)-1} \gamma & \text{shuffle(-exchange)} \\
& \longrightarrow \gamma y \xi_1 \xi_2 \cdots \xi_{\ell(k)-2} \xi_{\ell(k)-1} & \text{unshuffle} \\
& \longrightarrow \xi_{\ell(k)-1} \gamma y \xi_1 \xi_2 \cdots \xi_{\ell(k)-2} & \text{unshuffle} \\
& \vdots & \\
& \longrightarrow \xi_1 \xi_2 \cdots \xi_{\ell(k)-2} \xi_{\ell(k)-1} \gamma y & \text{unshuffle} \\
& = x \gamma y &
\end{array}
$$

Figure 11: *The length-$2\ell(k)$ path from node $\beta x y$ to node $x \gamma y$ in D_n; $x = \xi_1 \xi_2 \cdots \xi_{\ell(k)-1}$.*

- We route the arc
$$((x, \beta y),\ \ (x, y\gamma))$$
(where $\beta, \gamma \in Z_y$, $x \in Z_2^{\ell(k)}$, $y \in Z_2^{n-\ell(k)-1}$) between copies βy and $y\gamma$ of $D_{\ell(k)}$ via the length-$(2\ell(k)+1)$ path from node $x\beta y$ to node $x y \gamma$ in D_n specified in Figure 12.

It is shown in [1] how to *orchestrate* the traversals of the link-routing paths of this emulation in $\mathcal{D}_n$ so that only $O(1)$ messages (i.e., a constant number, independent of n and k) contend for any single link at one time. This orchestration (which we soon see is consistent with the SIMD regimen) assures us that the slowdown incurred by this emulation is at worst proportional to $\ell(k)$, with a small constant of proportionality.

B. (Partial) Permutation-Routing

In subsection 5.2, we indicate how the permutation-routes needed for our emulations can be done efficiently within a SIMD regimen, with very simple masking.

C. Emulating a Complete Binary Tree (level by level)

Our level-by-level emulation of $\mathcal{T}_{\ell(k)}$ by $\mathcal{D}_{\ell(k)}$ derives from [21] (wherein the technique is used on the shuffle-exchange network, a close relative of the de Bruijn network).

$$\begin{array}{rll}
x\beta y & = & \xi_1\xi_2\cdots\xi_{\ell(k)-1}\xi_{\ell(k)}\beta y \\
 & \longrightarrow & \xi_2\cdots\xi_{\ell(k)-1}\xi_{\ell(k)}\beta y\gamma \qquad \text{shuffle(-exchange)} \\
 & \longrightarrow & \xi_3\cdots\xi_{\ell(k)-1}\xi_{\ell(k)}\beta y\gamma\xi_1 \qquad \text{shuffle(-exchange)} \\
 & \vdots & \\
 & \longrightarrow & \beta y\gamma\xi_1\xi_2\cdots\xi_{\ell(k)-1} \qquad \text{shuffle(-exchange)} \\
 & \longrightarrow & y\gamma\xi_1\xi_2\cdots\xi_{\ell(k)-1}\xi_{\ell(k)} \qquad \text{shuffle(-exchange)} \\
 & \longrightarrow & \xi_{\ell(k)}y\gamma\xi_1\xi_2\cdots\xi_{\ell(k)-1} \qquad \text{unshuffle} \\
 & \longrightarrow & \xi_{\ell(k)-1}\xi_{\ell(k)}y\gamma\xi_1\xi_2\cdots\xi_{\ell(k)-2} \qquad \text{unshuffle} \\
 & \vdots & \\
 & \longrightarrow & \xi_1\xi_2\cdots\xi_{\ell(k)-1}\xi_{\ell(k)}y\gamma \qquad \text{unshuffle} \\
 & = & xy\gamma
\end{array}$$

Figure 12: *The length-$(2\ell(k)+1)$ path from node $x\beta y$ to node $xy\gamma$ in D_n; $x = \xi_1\xi_2\cdots\xi_{\ell(k)}$.*

We **assign** each node x of $\mathcal{T}_{\ell(k)}$ to node $0^{\ell(k)-\text{lgth}(x)}x$ of $\mathcal{D}_{\ell(k)}$.

We **route** arcs of $\mathcal{T}_{\ell(k)}$ via shortest paths in $\mathcal{D}_{\ell(k)}$.

- We route arc $(x, x0)$ of $\mathcal{T}_{\ell(k)}$ via the shuffle arc
$$(0^{n-\text{lgth}(x)}x, 0^{n-\text{lgth}(x)-1}x0)$$
in $\mathcal{D}_{\ell(k)}$.
- We route arc $(x, x1)$ of $\mathcal{T}_{\ell(k)}$ via the shuffle-exchange arc
$$(0^{n-\text{lgth}(x)}x, 0^{n-\text{lgth}(x)-1}x1)$$
in $\mathcal{D}_{\ell(k)}$.
- We route the predecessor arc $(x\beta, x)$ of $\mathcal{T}_{\ell(k)}$ via the "inverse" of the arc of $\mathcal{D}_{\ell(k)}$ that is used to route arc $(x, x\beta)$.

Since each arc of $\mathcal{T}_{\ell(k)}$ is routed along a single arc of $\mathcal{D}_{\ell(k)}$, this emulation incurs no overhead.

5.2 Implementation Issues

Computation, Communication, and Control. For all $k \leq 2^n$, the direct-product array $\mathcal{D}_{\ell(k)} \times \mathcal{D}_{n-\ell(k)}$ can be emulated by the order-n de Bruijn array $\mathcal{D}_n$ with slowdown $O(\min(n-\ell(k), \ell(k)))$; cf. Section 5.1.A. Because we expect to have $\ell(k) \leq n - \ell(k)$ in general, so that the macro-PE $\mathcal{K}$ is smaller than the macro-array $\mathcal{G}$, and because the principles of our emulation translate easily when the reverse inequality holds, we let the PIR of each bit-serial PE be the concatenation of the AIR and the MIR, so that

$$\text{AIR} = \text{PIR}[n - \ell(k) \ldots n-1] \; ; \; \text{MIR} = \text{PIR}[0 \ldots n - \ell(k) - 1].$$

Letting $g = 2^{n-\ell(k)}$, we order the copies of macro-PE $\mathcal{K}$ in the macro-array $\mathcal{G}$ so that macro-PE i consists exactly of those PEs of $\mathcal{D}_n$ whose numbers (in $\mathcal{D}_n$) are in the set $A^{(i)} = \{a \mid i = a \equiv i (\text{mod } g)\}$.

The communication macro-instruction that outputs a k-bit word to a macro-port SHUFFLE in $\mathcal{G}$ (other macro-ports are handled analogously) follows the routing function of the emulation in Section 5.1.A; cf. Figure 12 (with $\beta = \gamma$). From PE $x\beta y$, where $x \in Z_2^{\ell(k)}$, $\beta \in Z_2$, and $y \in Z_2^{n-\ell(k)-1}$, the routing makes $\ell(k)+1$ shuffle or shuffle-exchange hops to node $y\beta x$; from this node, $\ell(k)$ unshuffle hops take one to the target node $xy\beta$. During each of the first $\ell(k)+1$ hops, the SIMD regimen forces us to spend one transfer cycle for the communications through SHUFFLE ports and another one for the communications through SHUFFLE-EXCHANGE ports. Two more cycles per hop are spent on "memorizing" the most significant processor index bit of the previous node in the sequence, to make it equal the least significant bit of the next node. PEs with 0 in the most significant bit of the PIR send in the first two cycles, and PEs with 1 in this bit send in the last two cycles. Four bits of memory are used to buffer bits as they are input during one 4-cycle hop and output during the next. OUTPUT(SHUFFLE) performs $5\ell(k)+4$ bit-serial data transfers and $O(\ell(k)) = O(\log k)$ bit-serial computation instructions, where the constant in the big-O is estimated to be under 10.

```
macro-instruction OUTPUT (SHUFFLE) is {
  -- bit-parallel output from SHUFFLE in $\mathcal{D}_{n-\ell(k)}$
  if (PIR[$n-\ell(k)-1$] = PIR[$n-1$]) then {
    store to s[0] }                    -- to SHUFF
  if (PIR[$n-\ell(k)-1$] $\neq$ PIR[$n-1$]) then {
    store to e[0] }                    -- to EXCH
  for $i$ := 1 to $\ell(k)+1$ do {          -- toward $y\beta x$
    $j$ := $i$(mod 2)                      -- counters
    $k$ := ($j$ + 1)(mod 2)                -- (alternating)
  if (PIR[$n-1$] = 0) then {          -- 1st 2 cycles
    output (SHUFF)  from s[$k$]       -- out
    output (EXCH) from e[$k$] }       -- out
  if (PIR[0] = 0) then {              -- in
    if (PIR[$n-1$]=0) then {          -- classify
      input (UNSHUFF) to s[$j$] }     -- to shuffle
    if (PIR[$n-1$]=1) then {
      input (UNSHUFF) to e[$j$] }}    -- to exchange
  if (PIR[0] = 1) then {              -- in
    if (PIR[$n-1$]=0) then {          -- classify
      input (UN-EXCH) to s[$j$] }     -- to shuffle
    if (PIR[$n-1$]=1) then {
      input (UN-EXCH) to e[$j$] }}    -- to exchange
  if (PIR[$n-1$] = 1) then {          -- 2nd 2 cycles
    output (SHUFF)  from s[$k$]       -- out
    output (EXCH) from e[$k$] }       -- out
  if (PIR[0] = 0) then {              -- in
    if (PIR[$n-1$]=1) then {          -- classify
      input (UN-EXCH) to s[$j$] }     -- to shuffle
    if (PIR[$n-1$]=0) then {
      input (UN-EXCH) to e[$j$] }}    -- to exchange
  if (PIR[0] = 1) then {              -- in
    if (PIR[$n-1$]=1) then {          -- classify
      input (UNSHUFF) to s[$j$] }     -- to shuffle
    if (PIR[$n-1$]=0) then {
      input (UNSHUFF) to e[$j$] }}}-- to exchange
if (PIR[$\ell(k)$] $\neq$ PIR[0]) then {   -- at $y\beta x$; $\beta$?
  s[$j$] = e[$j$] }                   -- via exchange
for $i$ := 1 to $\ell(k)$ do {        -- toward $xy\beta$
  from s[$j$] output (UNSHUFF)
  input (SHUFF) to s[$j$] }
load from s[$j$] }                    -- macro-input
```

In contrast to the situation with hypercube arrays, the $O(\log k)$ cost of a communication step in a multigauge de Bruijn array cannot be compared to the communication cost in the bit-serial processing of the same data, as *it is not clear that bit-serial processing is even possible.* To wit, whereas $Q_{n-\ell(k)}$ is a subgraph of Q_n for any $\ell(k) \leq n$, there is no known way to identify a copy of $D_{n-\ell(k)}$ within D_n in any nontrivial case.

Within a macro-PE $\mathcal{K}$, we order PEs so that the ith bit position in all macro-PEs is occupied exactly by those PEs of $\mathcal{D}_n$ whose numbers (in $\mathcal{D}_n$) are in the set $A_i = \{a \mid i = \lfloor a/g \rfloor\}$.

The macro-PE data-transfer primitive AgOutput requires one bit in the PIR of the PE at hop $\ell(k)$ to depend on a bit in the PIR of the first PE in the sequence; cf. Figure 11 (with $\beta = \gamma$). This precludes memorizing this bit on the way; so, two complete rounds of $2\ell(k)$ cycles (one for each value of the bit) must be performed in sequence. The number of bit-serial data transfers is $6\ell(k)$, while the number of computation instructions is $O(\ell(k)) = O(\log k)$, with the constant in the big-O estimated to be under 10.

```
procedure AgOutput (SHUFFLE) is {
  -- macro-PE data transfer from SHUFFLE in $\mathcal{D}_{\ell(k)}$
  for $\beta$ := 0 to 1 do {                -- 2 rounds
    if (PIR[$n-1$] = $\beta$) then {          -- the round
      if (PIR[$n-2$] = PIR[$n-1$]) then {
        output (SHUFFLE)}                 -- $\beta xy$; start
      if (PIR[$n-2$] $\neq$ PIR[$n-1$]) then {
        output (SH-EXCHANGE) }}           -- $\beta xy$; start
    for $i$ := 0 to $\ell(k) - 2$ do {          -- toward $yx\beta$
      if (PIR[$n-2$] = PIR[$n-1$]) then {
        output (SHUFFLE) }
      if (PIR[$n-2$] $\neq$ PIR[$n-1$]) then {
        output (SH-EXCHANGE) }}
    if (PIR[$n-1$] = $\beta$) then {          -- arrival; $\beta$?
      output (SHUFFLE) }
    if (PIR[$n-1$] $\neq$ $\beta$) then {
      output (SH-EXCHANGE) }
    for $i$ := 1 to $\ell(k)$ do {              -- toward $x\beta y$
      output (UNSHUFFLE) }
    if (PIR[$n-m$] = $\beta$) then {          -- $x\beta y$; round?
      store to save }}
  load from save }                        -- AgInput
```

As described in Section 5.1.C, the parallel-prefix computations that implement arithmetic operations within each macro-PE $\mathcal{K}$ are performed by emulating a complete binary tree; our emulation is inspired by the corresponding emulation in the perfect shuffle network [21]. In this emulation, the PEs of $\mathcal{K}$ play the following roles: each nonleaf PE x of the tree communicates with its left child $x0$ through its SHUFFLE port and with its right child $x1$ through its SHUFFLE-EXCHANGE port; each nonroot PE $x\beta$ of the tree communicates with its parent through its UNSHUFFLE port if $\beta = 0$ and through its UNSHUFFLE-EXCHANGE port if $\beta = 1$. The emulation takes time $O(\log k)$ within $\mathcal{K}$; but, recall, $\mathcal{K}$ is itself implemented via the emulation by $\mathcal{D}_n$ of the direct-product array $\mathcal{D}_{\ell(k)} \times \mathcal{D}_{n-\ell(k)}$, and this emulation takes time $O(\log k)$ to emulate each communication step of $\mathcal{K}$. It follows that each k-bit arithmetic operation takes $O(\log^2 k)$ steps by $\mathcal{D}_n$.

Datapath Conversion. Finally, datapath conversion in $\mathcal{K}$ is performed by the macro-instruction GAUGE k. Our pseudo-code is designed to reveal the similarity between this procedure and its analogue for the hypercube (in global communication and data layout). Communication is slower here by a factor of $O(\log k)$, reflecting the emulation overhead for macro-PE data transfers. So, the "corner-turning" procedure takes $O(k \log^2 k)$ steps per k-word block of k-bit words. To the best of our knowledge, the following algorithm is original.[8]

```
macro-instruction GAUGE ($k$) is {
    -- datapath conversion in $\mathcal{D}_{\ell(k)}$
    for $i$ := 0 to $\ell(k) - 1$ do {          -- dimensions
        for $j$ := 1 to $k/2^{i+1}$ do {        -- blocks
            high := $(2j-1)2^i$                 -- block start
            low  := $2(j-1)2^i$                 -- block start
            for $j$ := 1 to $2^i$ do {          -- block bits
                if (AIR[$i$]= 0) then {         -- exchange
                    from M[high] AgOutput (NEXT) }
                if (AIR[$i$]= 1) then {
                    save := M[low]
                    store to M[low]
                    from save AgOutput (PREVIOUS) }
                if (AIR[$i$]= 0) then {
                    store to M[high] }
                low := low + 1                  -- next bit
                high := high +1 }}
        AgOutput (UNSHUFFLE) }}
```

[8]For the purposes of this algorithm, we view node $x1$ as the "next" node after node $x0$ (although it actually is two links away), whence the words "NEXT" and "PREVIOUS" in the procedure.

6 Extended Coterie Networks

The final family of networks we study differs from the others we have studied in two fundamental respects. Firstly, each structure in the family is the union of a graph and a hypergraph [3]. Secondly, each structure is dynamic in the sense that its hypergraph component may change at each communication step of a computation. We represent this dynamic nature by parameterizing each structure with a time-index as well as a size-index.

The *time-t* $n \times n$ *Extended Coterie Network graph* (*ECN graph*, for short) $C_n^{(t)}$ $(t = 0, 1, \ldots)$ has node-set Z_n^2. Each node (i, j) of $C_n^{(t)}$ is incident to arcs leading to and from the (at most four) nodes (i', j') of $C_n^{(t)}$ for which $|i - i'| + |j - j'| = 1$. Thus, the graph component of $C_n^{(t)}$ is a directed $n \times n$ mesh.[9] Additionally, each node of $C_n^{(t)}$ is incident to precisely one *coterie-hyperedge*: the coterie-hyperedge incident to node (i, j) of $C_n^{(t)}$ is a subset S of Z_2^n that

- contains node (i, j)
- is *connected* in the sense that the induced subgraph of the $n \times n$ mesh on the set S is a connected graph.

Note that at each time t, the coterie-hyperedges partition the node-set Z_n^2 of $C_n^{(t)}$. The coterie-hyperedges of $C_n^{(t)}$ do not depend in any way on the coterie-hyperedges of $C_n^{(t-1)}$. The coterie-hyperedges of $C_n^{(0)}$ are singleton sets of nodes. See Figure 13.

In the Extended Coterie (EC) processor array $C_n^{(t)}$ built on the ECN graph $C_n^{(t)}$, we order PEs according to the row-major ordering on their node-names: node (i, j) is the $(in + j)$th PE in the ordering. Each PE in $C_n^{(t)}$ has input and output ports that link it to its "North, South, East, and West" neighbors, i.e., to PEs $(i \pm 1, j)$ and $(i, j \pm 1)$; of course, PEs on the "edges" of the mesh lack some of these neighbors. Additionally, each PE has an input and output port connecting it to precisely one *coterie*, i.e., a (possibly irregularly shaped) bus: PEs (i, j) and (k, l) of the EC array $C_n^{(t)}$ are connected at time t to the same coterie (bus) just when nodes (i, j) and (k, l) of the ECN graph $C_n^{(t)}$ are incident to the same coterie-hyperedge. A coterie is a bus in the sense that, at any time, a single incident PE can "talk" while all other incident PEs "listen."[10]

[9]There is a natural "torus" variant of the ECN graph, in which each node (i, j) has precisely four neighbors: $((i \pm 1) \bmod n, j)$ and $(i, (j \pm 1) \bmod n)$. Since our emulation strategy requires the present "mesh" variant of the graph eventually (cf. Section 6.1), we choose to consider only the "mesh" variant throughout.

[10]The actual coterie network has the additional feature that if multiple PEs "talk," the "listeners" receive the bitwise logical OR of those messages. However, we have not yet found a use for this feature in the present context.

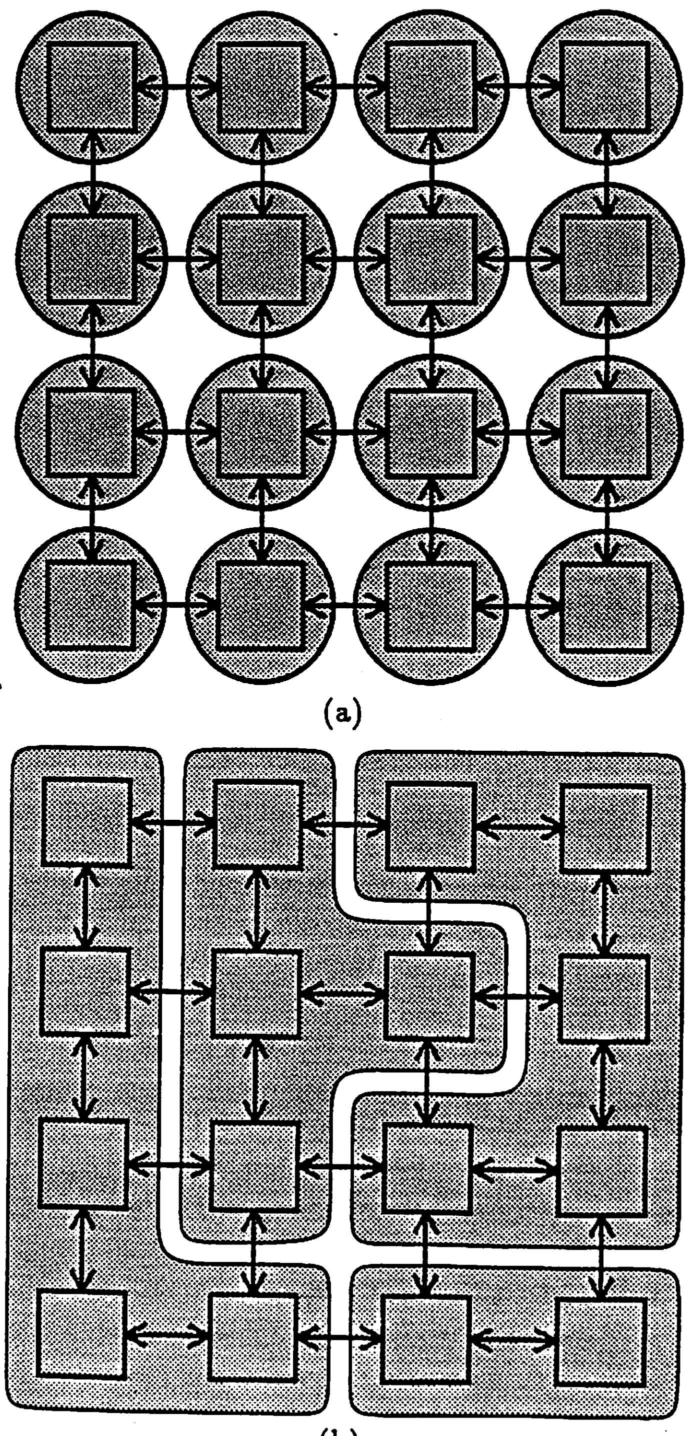


Figure 13: *The* 4 × 4 *ECN graph; shaded areas denote the coterie hyperedges:* (a) *when each coterie is a singleton;* (b) *with "arbitrary" coteries.*

6.1 Algorithmic Issues

Our assessment of the time for communicating in an EC array is consistent with our practice in previous sections, except as regards the time for coterie (bus) communications—which does not arise with the other networks we study. In the interests of getting a practical assessment of communication time in an EC array, we sacrifice scalability, by viewing the side-dimension n of $\mathcal{C}_n^{(t)}$ as a fixed integer (say, 512, as in the implemented version of the network). At the cost of the scalability of our results, we assess only one step for broadcasting a message along a coterie. We discuss the rationale behind our cost assessment in Section 6.2. A scalable delay model would emerge from the following reasoning. Since switches are necessary to implement the ability of a PE to join or leave coteries at every time step, the delay of a message along a coterie is proportional to the number of switches the message traverses; therefore, although the constant of proportionality is quite small in practice (a conservative estimate would be 1/50th the transit time of a PE), an honest scalable delay model would be *linear* in the diameter of the coterie.

A. Emulating a Direct-Product Network

Although the $n \times n$ mesh that underlies the ECN graph is the direct product of two copies of the length-n linear array, the aggregation that arises from this observation is computationally inferior to an aggregation that has the ECN array $\mathcal{C}_n^{(t)}$ emulate the direct product $\mathcal{C}_{\sqrt{k}}^{(t)} \times \mathcal{C}_{n/\sqrt{k}}^{(t)}$. In short, the latter aggregation affords us data transfer times that are both symmetric in the row and column directions and smaller in the worse of these directions than they would be with the former aggregation (which would mandate "long skinny" aggregates). Moreover, in accord with our Principle of Self-Similarity, the latter aggregation allows the direct use of $\sqrt{k} \times \sqrt{k}$ versions of EC array algorithms on the macro-PEs of the gauge-k logical EC array. Finally, the latter aggregation allows us to exploit the observed fact that many computational problems map more naturally onto square mesh-like arrays than onto "long skinny" ones. For simplicity of notation, we assume henceforth that the desired gauge size k is a perfect square.

We now sketch the algorithmic basis for the emulation by an arbitrary $(N = n^2)$-PE EC array $\mathcal{C}_n^{(t)}$ of a direct-product EC array $\mathcal{C}_{\sqrt{k}}^{(t)} \times \mathcal{C}_{n/\sqrt{k}}^{(t)}$. Summarizing the preceding discussion, within the context of our study, at each time t,

- the $n \times n$ EC array $\mathcal{C}_n^{(t)}$ is the host array $\mathcal{H}$;
- the sought gauge size k is a perfect square;
- the $\sqrt{k} \times \sqrt{k}$ EC array $\mathcal{C}_{\sqrt{k}}^{(t)}$ is the macro-PE array $\mathcal{K}$;
- the $n/\sqrt{k} \times n/\sqrt{k}$ EC array $\mathcal{C}_{n/\sqrt{k}}^{(t)}$ is the macro-array $\mathcal{G}$.

We **assign** each node $((a,b),(c,d))$ of the guest direct-product array $C^{(t)}_{\sqrt{k}} \times C^{(t)}_{n/\sqrt{k}}$, to node $(c\sqrt{k}+a, d\sqrt{k}+b)$ of the host array $C^{(t)}_n$.

We **emulate** the links and coteries of the guest array $C^{(t)}_{\sqrt{k}} \times C^{(t)}_{n/\sqrt{k}}$ within the host array $C^{(t)}_n$ as follows.

- We route mesh-links of the forms

  $$\begin{array}{rcl} ((a,b),(c,d)) & \longrightarrow & ((a\pm 1,b),(c,d)) \\ ((a,b),(c,d)) & \longrightarrow & ((a,b\pm 1),(c,d)) \end{array}$$

  within copy (c,d) of macro-PE $C^{(t)}_{\sqrt{k}}$ along the unit-length paths (i.e., arcs) in the host array $C^{(t)}_n$:

  $$\begin{array}{rcl} (c\sqrt{k}+a, d\sqrt{k}+b) & \longrightarrow & (c\sqrt{k}+a\pm 1, d\sqrt{k}+b) \\ (c\sqrt{k}+a, d\sqrt{k}+b) & \longrightarrow & (c\sqrt{k}+a, d\sqrt{k}+b\pm 1) \end{array}$$

- We route mesh-links of the forms

  $$\begin{array}{rcl} ((a,b),(c,d)) & \longrightarrow & ((a,b),(c\pm 1,d)) \\ ((a,b),(c,d)) & \longrightarrow & ((a,b),(c,d\pm 1)) \end{array}$$

  between macro-PEs (c,d) and either $(c\pm 1,d)$ or $(c,d\pm 1)$, respectively, within macro-array $C^{(t)}_{n/\sqrt{k}}$, along the following length-$\sqrt{k}$ paths in $C^{(t)}_n$:

  $$(c\sqrt{k}+a, d\sqrt{k}+b) \longrightarrow (c\sqrt{k}+a\pm 1, d\sqrt{k}+b) \longrightarrow \cdots \longrightarrow (c\sqrt{k}+a\pm\sqrt{k}, d\sqrt{k}+b)$$
  $$(c\sqrt{k}+a, d\sqrt{k}+b) \longrightarrow (c\sqrt{k}+a, d\sqrt{k}+b\pm 1) \longrightarrow \cdots \longrightarrow (c\sqrt{k}+a, d\sqrt{k}+b\pm\sqrt{k})$$

  Each link of $C^{(t)}_n$ that appears in one of the length-$\sqrt{k}$ paths is used to emulate $\sqrt{k}$ different paths in the macro-array. Pipelining allows one to avoid much of the congestion suggested by this fact. Specifically, one can ensure that, at every *host* communication step ($\sqrt{k}$ of which are required to emulate each *guest* communication step), each host link is devoted to exactly one of $\sqrt{k}$ guest links routed over it.

Emulating the coteries of the guest direct-product array is conceptually as easy as emulating the mesh links: by dint of our assignment of PEs of $C^{(t)}_{\sqrt{k}} \times C^{(t)}_{n/\sqrt{k}}$ to PEs of $C^{(t)}_n$, each coterie within a macro-PE and each coterie within the macro-array is a coterie within the host array.

- Each coterie $S^{(t)}_{\sqrt{k}} \subseteq Z^2_{\sqrt{k}}$ within macro-PE (a,b) of $C^{(t)}_{\sqrt{k}}$ is the coterie

  $$S^{(t)}_1 = \{((a\sqrt{k}+u),(b\sqrt{k}+v)) \mid (u,v) \in S^{(t)}_{\sqrt{k}}\}$$

  in the host array.

- Each coterie $S^{(t)}_{n/\sqrt{k}} \subseteq Z^2_{n/\sqrt{k}}$ within the macro-array $C^{(t)}_{n/\sqrt{k}}$ is the coterie

  $$S^{(t)}_2 = \{((a\sqrt{k}+u),(b\sqrt{k}+v)) \mid (a,b) \in S^{(t)}_{n/\sqrt{k}} \wedge (u,v) \in Z^2_{\sqrt{k}}\}$$

  which contains all nodes of each macro-PE incident to $S^{(t)}_{n/\sqrt{k}}$.

In particular, our PE-assignment ensures that the nodes of $S^{(t)}_1$ and $S^{(t)}_2$ are connected in $C^{(t)}_n$ just when the nodes of $S^{(t)}_{\sqrt{k}}$ and $S^{(t)}_{n/\sqrt{k}}$ are connected in $C^{(t)}_{\sqrt{k}}$ and $C^{(t)}_{n/\sqrt{k}}$, respectively; the host coteries are, therefore, well defined.

B. Routing (Partial) Permutations

In Section 6.2 we indicate how the permutation routes needed for our emulations can be done efficiently within the SIMD regimen, with very simple masking.

C. Emulating a Complete Binary Tree (level-by-level)

Within our delay model of unit transfer time along a coterie, level-by-level emulation of the complete binary tree $\mathcal{T}_{\ell(k)}$ by the EC array $C^{(t)}_{\sqrt{k}}$ can be performed with no slowdown. We simplify our description (without compromising conceptual generality) by assuming that the sought gauge size k is simultaneously a perfect square and a power of 2, i.e., has the form 2^{2p} for some p.

We **assign** PEs of $\mathcal{T}_{\ell(k)}$ to PEs of $C^{(t)}_{\sqrt{k}}$ via the following many-to-one map. For any binary string x, let $\Lambda(x)$ denote the integer represented in binary by x.

- For $p < l \leq \ell(k)$, each level-l node $x \in Z^l_2$ of $\mathcal{T}_{\ell(k)}$ is assigned to node (a,b) of $C^{(t)}_{\sqrt{k}}$, where $(a,b) \in Z^2_{\sqrt{k}}$ is the unique pair such that $a\sqrt{k}+b = \Lambda(x1^{\ell(k)-l})$.

- For $0 \leq l \leq p$, each level-l node $x \in Z^l_2$ of $\mathcal{T}_{\ell(k)}$ is assigned to node $(\Lambda(x1^{p-l}), \sqrt{k}-1)$ of $C^{(t)}_{\sqrt{k}}$.

We observe that our assignment maps all level-p nodes of $\mathcal{T}_{\ell(k)}$ to nodes in column $(\sqrt{k}-1)$ of $C^{(t)}_{\sqrt{k}}$; it maps every node $x \in Z^p_2$ to row $\Lambda(x)$. Every node of $\mathcal{T}_{\ell(k)}$ in a level $l < p$ is mapped to the same node of $\mathcal{C}^{(t)}_{\sqrt{k}}$ to which its right child is mapped. In much the same way, the height-p subtree $\mathcal{T}_x$ of $\mathcal{T}_{\ell(k)}$ rooted at level-p node x is mapped to row $\Lambda(x)$ of $\mathcal{C}^{(t)}_{\sqrt{k}}$. Consequently, for all length-$(0 \leq l < \ell(k))$ string/nodes x, nodes x and $x0$ of $\mathcal{T}_{\ell(k)}$ both belong either to the same row or to the same column of $\mathcal{C}^{(t)}_{\sqrt{k}}$; no other level-$l$ or level-$(l+1)$ nodes of $\mathcal{T}_{\ell(k)}$ are mapped to the (shortest) path, call it P_x, that connects the images of x and $x0$.

We **emulate** the links of $\mathcal{T}_{\ell(k)}$ within $\mathcal{C}^{(t)}_{\sqrt{k}}$ as follows.

- We route the link $(x \rightarrow x0)$ via the coterie P_x.
- We route the link $(x \rightarrow x1)$ as a "null" link within $\mathcal{C}^{(t)}_{\sqrt{k}}$, because tree-PEs x and $x1$ are both assigned to the same PE of $\mathcal{C}^{(t)}_{\sqrt{k}}$.
- We route the predecessor link $(x\beta \rightarrow x)$ of $\mathcal{T}_{\ell(k)}$, $\beta \in Z_2$, via the reversal of the routing path for the link $(x \rightarrow x\beta)$.

Our emulation maps links of $\mathcal{T}_{\ell(k)}$ to paths of length 0 or 1 in $\mathcal{C}^{(t)}_{\sqrt{k}}$; therefore, the emulation incurs no slowdown.

6.2 Implementation Issues

We now apply the techniques developed for the ECN array to a real processor array, the Content Addressable Array Parallel Processor (CAAPP). We describe a specific implementation of the EC network: the components (mesh-links and coteries) have different overhead properties, hence are discussed separately. First we sketch the target architecture and present the programming model.

Basic CAAPP Architecture. The CAAPP is an SIMD array that, for the purposes of our study, differs from the model architecture of Section 2 mainly in its memory hierarchy: Each PE of the CAAPP has 320 bits of on-chip cache (accessible in a single cycle) and 32K bits of off-chip memory that can be loaded into cache memory roughly at the rate of one bit every three cycles. All instructions, unless otherwise specified, require a single cycle (time-step).

The CAAPP has two communication networks, the nearest-neighbor mesh interconnection network with wraparound, and a reconfigurable mesh (also with wraparound) called the coterie network. These correspond to the mesh-links and coteries of the ECN array, respectively. When using the mesh network, PEs perform the following operation in a single instruction:

Given a specified memory location M and a direction D (north, south, east, or west), they read data from M in the neighboring PE in the D direction and deposit the contents in location M in local memory.

In conformance with our assumptions about pure SIMD behavior, all PEs must read (or write) the same memory location from (or to) the same direction.

In order to use the coterie network (simplified slightly for this paper), each PE controls a set of four switches, N, S, E, and W, enabling the creation of electrically isolated groups of PEs that share a common bus. The switches control access in the north, south, east, and west directions, respectively. These isolated groups of processors form coteries. For example, when mesh-neighbor PEs close switches between them (e.g. a PE closes its W switch, its west neighbor closes its E switch) to share a circuit, those PEs are then members of the same coterie. The network is used to transfer data as follows: (1) PEs write the specified datum to the coterie-link; (2) the array controller issues an instruction for the network to propagate the data; (3) PEs get the data from the coterie network by reading from their coterie links.

The coterie network switches are set by loading the corresponding bits of the *mesh control register* (MR) in each PE. Because each PE views the MR as local storage, coterie configurations can be loaded from memory; they can also be set as a result of local data-dependent calculations. One particular way of using the coterie network is to set the switches so that columns and rows are isolated. It may also be useful to then divide the row and column "buses" into segments. The coterie network can thus emulate the mesh with reconfigurable buses [16] and the polymorphic torus [15].

The Cost of Coterie Network Operations. The operations that read and write on the coterie links (steps (1) and (3) of the data transfer paradigm) each take one cycle. The "propagate" operation (step (2) of the paradigm) also takes one cycle in the context of this paper. This last time assessment warrants a more detailed description of the workings of the coterie network.

Once values have been written to the coterie network (i.e., step (1) of the paradigm has been executed), the controller issues a command for the signal to propagate. At the end of the instruction cycle, the signals will have propagated through a certain number of PEs in all directions from the originating PEs. The actual function of number of PEs traversed per unit time depends on the characteristics of the device, but is not of critical importance to multigauge emulation. It has been experimentally determined to be 50 PEs per instruction cycle. Since the maximum gauge k can be assumed to be not larger than a few hundred, and since the shapes of the macro-PEs are rectangles, the coterie for the macro PE will be less than 50 in diameter and thus only a single propagation step is required.

Implementation Notation. In order to show the effect of our technique on a real system, we will use the actual language in which the algorithms are written on the CAAPP, and from which timings have been directly computed. The CAAPP PE instructions are indicated by brackets "[ ]" and are embedded in the C language programs of the controller. The CAAPP PE instructions in the sample code below use the following notation:

!!	all PEs execute the instruction unconditionally
A!	only PEs with register A set execute the instruction
A,X	are one bit registers; X is also the coterie network read/write port.
X-PC	precharge the coterie network (to compensate for the unequal times to charge and discharge the buses)
MR	a four-bit register containing the coterie switch settings; can be loaded from memory in a single instruction.

The Use of Precomputed Masks. In the implementations discussed below, the decision whether or not a PE executes an issued instruction often depends on its position within the macro-PE. To make these determinations during the execution of the macro-PE instruction is not practical: operations on the position representation require $O(\log k)$ bit-serial instructions. Instead, we use *precomputed* masks: we assume that the PEs to be involved in each instruction can be selected by loading the A register with a precomputed mask that resides in cache memory. Since for any gauge-k, only $k + 6\log k + 5$ bit-masks are needed in our implementation, this strategy consumes only moderate time and space. In particular:

- all masks for a given gauge size less than, say, 128 fit easily into PE cache memory;
- all masks for all gauge sizes likely to be used, say 20 different sizes, fit into a small fraction of PE main memory;
- loading the PE cache with the necessary masks (for a given gauge size as part of the GAUGE k instruction) takes only a small fraction of the cost of corner turning: 128 bits can be loaded in less than 400 cycles, while corner turning requires $O(k^2)$ cycles (as is shown below).

A. Macro-PE to Macro-PE Data Transfer

The Mesh Network. The primitive operation of the nearest-neighbor mesh network is a transfer (for all PEs at once) of data from location M in memory to that same location in the neighboring PE in a specified direction. The corresponding macro-array instruction is identical except that it transfers k-bit macro-words among macro-PEs, instead of single bits. The macro-array instruction

```
macro-instruction MeshOutput
        (Direction: Port, MemLoc: MemoryAddress) is {
[ MemLoc := Direction( MemLoc ) !! ]; }
```

is emulated by the following host array procedure:

```
procedure AgMeshOutput
        (Direction: Port, MemLoc: MemoryAddress) is {
for ($i = 0$; $i < \sqrt{k}$; $i$++)
        [ MemLoc := Direction( MemLoc ) !! ]; }
```

This procedure uses $\sqrt{k}$ nearest-neighbor moves, each requiring one time-step. The equivalent gauge-1 procedure to move k-bit words a distance of one PE uses k nearest neighbor moves and k time-steps.

The Coterie Network. Since the actual communication mechanism in the coterie network is primitive—there is only one Read/Write port—the implementation of the macro-array coterie network consists of emulating two primitive operations:

- setting the macro-array $\mathcal{C}^{(t)}_{n/\sqrt{k}}$ switches, and
- the actual data transfer among macro-PEs in the same coteries.

The first of these operations requires setting the appropriate switches in the underlying host array $\mathcal{C}^{(t)}_{n}$: the second requires orchestrating the use of the communication links (time-multiplexing) when multiple PEs in a copy of macro-PE $\mathcal{C}^{(t)}_{\sqrt{k}}$ are sending data.

We present first the mapping of the switch settings of $\mathcal{C}^{(t)}_{n/\sqrt{k}}$ onto $\mathcal{C}^{(t)}_{n}$. A macro-PE switch is emulated entirely by switches of the k host-array PEs that comprise the macro-PE. We call a switch *internal* (resp., *external*) if it connects PEs that belong to the same (resp., distinct) macro-PEs. At each step during the operation of the macro-array coterie network, all internal switches are closed, so that each macro-PE will act as a unit. The external switches in a given direction (e.g., toward the macro-PE in the east direction) are set according to the setting of the corresponding (logical) macro-PE switch. For example, if the E switch of the macro-PE is set, then the E switches of all $\sqrt{k}$ PEs on the east side of the macro-PE will also be set.

When programming the physical coterie network, one sets switches by writing to the corresponding bit in the MR register. This 4-bit register is loaded from memory in a single instruction cycle. The macro-array instruction loads the macro-PE equivalent of the MR with the low-order 4 bits from a specified memory location starting at, say, location MemLoc. The emulation proceeds by first creating coteries corresponding to each macro-PE (closed internal, open external switches), and then having PEs 0 through 3 successively broadcast their MemLoc values. Each PE in the macro-PE then sets its switches according to the value of the input signal and its position within the macro-PE. The following masks are used:

IsolateMacroPEMask:	contains the initial switch settings
MacroPEMask($0, \ldots, k-1$):	specifies selection of PEs $0, \ldots, k-1$ within each macro-PE
SwitchMask(N, E, W, S):	indicates which PEs participate in the setting of the switches in each direction.

The macro-array instruction

```
macro-intruction LoadMeshRegister
        (MemLoc: MemoryAddress) is {
[ MR := MemLoc !! ]; }                          -- Load macro-PE mesh registers
```

is emulated by the host array procedure:

```
procedure AgLoadMeshRegister
        (MemLoc: MemoryAddress) is {
[ MR := IsolateMacroPEMask !! ];                --make coteries of macro-PEs
for (i = 0; i < NumberOfSwitches; i++) {
        [ A := MacroPEMask(i) !! ];             --select PE i
        [ X-PC := MemLoc A! ];                  --send macro-switch setting
        [ PROPAGATE ];                          --let signal propagate
        [ A := X !! ];                          --input broadcast signal
        [ MRCopy(i) := SwitchMask(i) A! ]; }    --if set, close switch
[ MR := MRCopy !! ]; }                          --load mesh registers
```

The procedure executes five instructions per switch plus two cycles of overhead for a total of 22.

We now examine how the actual data transfer is carried out. Broadcast of k-length words (from location MemLocSend to location MemLocReceive) by macro-PEs must be done bit-by-bit, as all k PEs in each macro-PE are members of the same coterie. The locations of the broadcasting and receiving macro-PEs are stored in variables BroadcastMask and ReceiveMask, respectively. The macro-array instruction

```
macro-instruction CoterieOutput
        (MemLocSend: InputAddress, MemLocReceive: OutputAddress) is {
[ A := BroadcastMask !! ];                      --select sending macro-PEs
[ X-PC := MemLocSend A! ];                      --broadcast data
[ PROPAGATE ];                                  --propagate
[ A := ReceiveMask !! ];                        --select receiving macro-PEs
[ MemLocReceive := X A! ]; }                    --get data
```

is emulated by the host array procedure

```
procedure AgCoterieOutput
        (MemLocSend: InputAddress, MemLocReceive: OutputAddress) is {
for ($i = 0$; $i < k$; $i$++) {
        [ A := BroadcastMask !! ];              --select sending macro-PEs
        [ A := A AND Macro-PE-Mask($i$) !! ];   --select PE i for ith bit
        [ X-PC := MemLocSend A! ];              --broadcast a bit
        [ PROPAGATE ];                          --let signal propagate
        [ A := ReceiveMask !! ];                --select receiving macro-PEs
        [ A := A AND Macro-PE-Mask($i$) !! ];   --select PE i for ith bit
        [ MemLocReceive := X A! ]; }}           --input from network.
```

This procedure executes $7k$ instructions to transfer a k-bit word.

The coterie network can also be used to emulate the mesh with reconfigurable buses network: since the communication in that network takes place in only one dimension at a time, each macro-PE can be partitioned into $\sqrt{k}$ coteries corresponding to $\sqrt{k} \times 1$ paths of PEs. Since each of these strips can transmit independently, only $\sqrt{k}$ broadcast cycles (rather than k) are required.

Datapath Conversion. If one uses the permutation-routing techniques of Section 3.2 to implement datapath conversion on the CAAPP, the resulting algorithm consumes time $O(\sqrt{k}k^2)$ to perform the gauge size k conversion. However, one can exploit the broadcast capability of the coterie network to derive a competing algorithm that consumes $O(k^2)$ steps. The broadcast-based procedure begins by loading the appropriate masks from main memory into cache. The first PE instruction isolates each macro-PE as a coterie by appropriately setting the coterie network switches. Then sequentially, but parallel in the sense that the same operation is being executed within each coterie, each bit of each PE is broadcast to its destination.

```
AgGauge $k$ is {
LoadMultigaugeMasks(k);
[ MR := IsolateMacro-PEMask !! ];
for ($i = 0$; $i < k$; $i$++)
        for ($j = 0$; $j > k$; $j$++) {
                [ A := Macro-PEMask($i$) !! ];
                [ X-PC := MemorySend($j$) A! ];
                [ PROPAGATE ];
                [ A := Macro-PEMask($j$) !! ];
                [ MemoryReceive($i$) := X A! ]; } }
```

This procedure requires five instructions per bit: load the sender mask, put the output bit into the transfer register, execute the broadcast, load the receiver mask, and get the bit from the transfer register. Using this method, datapath conversion takes $5k^2 + 50$ machine cycles per k bits of memory.

Parallel-Prefix. The parallel-prefix operator can be implemented on the CAAPP by applying the standard two-phase algorithm to the tree network emulation presented in Section 6.1.C. It is possible, however, to modify that algorithm to take advantage of the broadcast data transfer capability of the coterie network, thereby cutting in half the number of communication steps. The following algorithm is a simplified and (slightly) more efficient version of the one developed for the mesh with reconfigurable buses (see e.g. [16]).

The broadcast-based algorithm collapses two phases of the standard algorithms for computing the parallel-prefix on a complete binary tree. Specifically, as data passes up the tree, level by level, being combined with sibling data at each node, we broadcast the partial result computed at each node to *all descendants of the right-hand sibling node.* Through this mechanism, the second phase of the standard algorithm (wherein partial results are passed down the tree) is obviated.

This procedure is implemented by creating coteries during each phase $i = 0, \ldots, \log k - 1$ (during which level i of the tree is emulated). At phase i, each coterie contains: one PE at level $\log k - i - 1$, both of its child-PEs, and all of the descendants of its right child. These coteries are constructed as follows. Recall that our emulation of the binary tree by the coterie network has the same coterie PE emulate each tree node and the node's right child. The link from the left child to the parent therefore also contains the right child. Further, for the emulation of the lower $\log \sqrt{k}$ levels of the tree, all descendants of the right child are also part of that coterie. The emulation of the upper $\log \sqrt{k}$ levels of the tree is not quite so direct, but is still straightforward.

The following operations take place during each phase i: $j = 2^{\log k - i - 1}$ coteries are formed, each consisting of $2^i + 1$ tree-nodes. One PE in each coterie broadcasts its data; the rest of the tree-nodes receive that data and combine it with their own. Using row-major indexing within the macro-PEs, the PEs within the jth coterie during iteration i can be enumerated as follows: the sender PE is computed by taking i 1s and adding $j * 2^{i+1}$. The receivers are the 2^i PEs numbered consecutively from the sender. As in the previous procedures, much of the computation occurs off-line: for each of the *logk* iterations, there resides in memory masks for the coterie switches (PPSwitchMask) and the sending and receiving PEs (PPSendMask,PPReceiveMask).

```
procedure AgParallelPrefix
        (Data: MemoryAddress, DataSize: WordLength) is {
for ($i = 0$; $i < \log k$; $i$++) {
        [ MR := PPSwitchMask($i$) !! ];              --Create Coteries
        for ($j = 0$; $j <$ DataSize; $j$++) {
                [ A := PPSendMask($i$) !! ];         --left sibling
                [ X-PC := Data($i$) A! ];
                [ PROPAGATE ];
                [ A := PPReceiveMask($i$) !! ];      --right sibling and descendants
                [ Temp($i$) := X A! ]; }
        [ Combine(Data,Temp,j,*,A!) ]; } }           --macro to perform * operation
```

This procedure requires $\log k$ iterations during which a mask is loaded (one instruction), j bits transfered using the coterie network ($5j$ instructions), and the data combined with the current value ($2j$ instructions). The total is $\log k(7j + 1)$.

7 Conclusion

We have presented, and illustrated on three examples, a strategy for emulating a family $\{\mathcal{B}_k\}$ of k-bit-parallel SIMD processor arrays on its bit-serial instance $\mathcal{H} = \mathcal{A} = \mathcal{B}_1$. Our technique requires emulating a direct-product graph by $\mathcal{H}$, and implementing ALU-computations within its aggregates. Our goal has been a collection of consistent virtual machine instruction sets, indexed by the gauge size. The *flexibility* and *conceptual clarity* of the result are accompanied by significant *performance advantages*. We believe that the most appropriate method of assessing these advantages is to compare the cost of the various macro-instructions we have implemented to the cost of achieving the same instruction functionality within a purely bit-serial computation regimen. While an exact assessment would require details of network topology and specifics of the node-architecture, the following big-O assessment indicates that our emulation approach outperforms its bit-serial alternative when processing k-bit data as follows.

The following compares the time required for various k-bit operations when implemented on a bit-serial hypercube, a bit-serial de Bruijn array, and a bit serial extended coterie array, both using our emulation algorithms and using only straightforward software implementation, without emulations. We consider three classes of operations.

1. Operations that are cheap ($O(1)$ circuitry) if multigauge behavior is implemented in hardware [24] (e.g.: communication, bit-wise logic, memory reference)

	Bit-serial Computation	*Emulated Bit-Parallel Computation*
Hypercube	$O(k)$ steps per operation	$O(1)$ steps per operation
De Bruijn	$O(k)$ steps per operation	$O(\log k)$ steps per communication $O(1)$ steps per other operation
ECN	$O(k)$ steps per operation	$O(\sqrt{k})$ steps per communication $O(1)$ steps per other operation

2. Operations that are moderate in cost ($O(k)$ circuitry) if multigauge behavior is implemented in hardware [24] (e.g.: arithmetic)

	Bit-serial Computation	*Emulated Bit-Parallel Computation*
Hypercube	$O(k)$ steps per operation	$O(\log k)$ steps per operation
De Bruijn	$O(k)$ steps per operation	$O(\log^2 k)$ steps per operation
ECN	$O(k)$ steps per operation	$O(\log k)$ steps per operation

3. Operations whose hardware complexity is substantial and detail-dependent [24] (e.g.: shifting and multiplication)

Bit-serial Computation	*Emulated Bit-Parallel Computation*
$O(k^{O(1)})$ steps per operation	$O(\log^{O(1)} k)$ steps per operation

Note that operations in this class typically would not even be present in the native instruction set of a bit-serial machine, but would be welcome in a bit-parallel one. Our emulations achieve such operations at modest cost.

Of course, the above assessment ignores the cost of datapath conversion, which is incurred each time a new gauge size is selected. As we noted earlier, in our implementation of macro-instruction GAUGE k in Section 3.2, when one changes from bit-serial mode to gauge size k, this overhead cost is never greater than the cost of k deterministic off-line permutation routes within the emulated k-PE macro-PEs (multiplied by any overhead incurred when emulating these macro-PEs); indeed the particular form of the needed permutations often allows one to route these permutations even more efficiently than general ones.

Because the problem of achieving multigauge behavior has been a vehicle for illustrating a general philosophy of trying to achieve architectural enhancements algorithmically rather than in hardware, we have taken no pains to optimize our multigauge virtual machines. Were we to undertake such optimization, one path we would explore is estimating the instruction mix of our bit-parallel machines and considering the possibility of emulating *totally parallel* adders ([10], Section 4.7) in our macro-PEs, rather than carry-lookahead adders. Emulating such adders would increase both the overhead of gauge conversion and the cost of bit-parallel operations such as comparison, but in the presence of the appropriate instruction mix, this might be a cost-effective tradeoff.

References

[1] F.S. Annexstein, M. Baumslag, A.L. Rosenberg (1990): Group action graphs and parallel architectures. *SIAM J. Comput. 19*, 544-569.

[2] R. Barman, M. Bolotski, D. Camporese, J.J. Little (1990): Silt: The bit-parallel approach. *10th Intl. Conf. on Pattern Recognition*, Vol. II, 332-336.

[3] C. Berge (1973): *Graphs and Hypergraphs.* North-Holland, Amsterdam.

[4] S.N. Bhatt, F.R.K. Chung, J.-W. Hong, F.T. Leighton, B. Obrenić, A.L. Rosenberg, E.J. Schwabe (1991): Optimal emulations by butterfly-like networks. *J. ACM*, to appear.

[5] S.N. Bhatt, F.R.K. Chung, F.T. Leighton, A.L. Rosenberg (1991): Efficient embeddings of trees in hypercubes. *SIAM J. Comput.*, to appear.

[6] G.E. Blelloch (1989): Scans as primitive parallel operations. *IEEE Trans. Comp. 38*, 1526-1538.

[7] H.N. Gabow, T. Nishizeki, O. Kariv, D. Leven, O. Terada (1986): Algorithms for edge-coloring graphs. Typescript, Univ. Colorado.

[8] A.V. Goldberg and S.A. Plotkin (1987): Efficient parallel algorithms for $(\Delta+1)$-coloring and maximal independent set problems. Typescript, MIT.

[9] D.S. Greenberg, L.S. Heath and A.L. Rosenberg (1990): Optimal embeddings of butterfly-like graphs in the hypercube. *Math. Syst. Th. 23*, 61-77.

[10] K. Hwang (1979): *Computer Arithmetic: Principles, Architecture, and Design.* John Wiley and Sons, New York.

[11] K.E. Iverson (1962): *A Programming Language.* John Wiley and Sons, New York.

[12] S.I. Kartashev and S.P. Kartashev (1979): A multicomputer system with dynamic architecture. *IEEE Trans. Comp., C-28*, 704-721.

[13] R. Koch, F.T. Leighton, B. Maggs, S. Rao, A.L. Rosenberg, E.J Schwabe (1990): Work-preserving emulations of fixed-connection networks. Submitted for publication; see also, *21st ACM Symp. on Theory of Computing*, 227-240.

[14] R.E. Ladner and M.J. Fischer (1980): Parallel prefix computation. *J. ACM 27*, 831-838.

[15] H. Li and M. Maresca (1989): Polymorphic-torus network. *IEEE Trans. Comp. 38*, 1345-1351.

[16] R. Miller, V.K. Prasanna-Kumar, D. Reisis, Q.F. Stout (1988): Meshes with reconfigurable buses. *5th MIT Conf. on Advanced Research in VLSI* (J. Allen and F.T. Leighton, eds.) MIT Press, Cambridge, MA, 163-178.

[17] T.D. de Rose, L. Snyder, C. Yang (1987): Near-optimal speedup of graphics algorithms using multigauge parallel computers. *Intl. Conf. on Parallel Processing*, 289-294.

[18] A.L. Rosenberg (1991): Product-shuffle networks: toward reconciling shuffles and butterflies. *Discr. Appl. Math.*, to appear.

[19] Y. Saad and M.H. Schultz (1989): Data communication in hypercubes. *J. Parallel and Distr. Computing 6*, 115-135.

[20] M.R. Samatham and D.K. Pradhan (1989): The deBruijn multiprocessor network: a versatile parallel processing and sorting network for VLSI. *IEEE Trans. Comp. 38*, 567-581.

[21] J.T. Schwartz (1980): Ultracomputers. *ACM Trans. Prog. Lang. 2*, 484-521.

[22] L. Snyder (1982): Introduction to the Configurable Highly Parallel computer. *Computer 15* (1) 47-56.

[23] L. Snyder (1985): An inquiry into the benefits of multigauge parallel computation. *Intl. Conf. on Parallel Processing*, 488-492.

[24] L. Snyder and C. Yang (1988): The principles of multigauging architectures. Typescript, Univ. Washington.

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

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

[27] V.G. Vizing (1964): On an estimate of the chromatic class of a p-graph (in Russian). *Diskret. Analiz 3*, 25-30.

[28] C.C. Weems, S.P. Levitan, A.R. Hanson, E.M. Riseman, D.B. Shu, J.G. Nash (1987): The Image Understand Architecture. *Intl. J. Computer Vision 2*, 251-282.